Computer Science Curriculum for the Year 2000

Department of Computer Science
Brown University
Providence, Rhode Island 02912
(401) 863-7600

Technical Report No. CS-90-28
November 1, 1990

Brown University

Computer Science Curriculum for the Year 2000

Department of Computer Science
Brown University
Providence. RI 02912-1910
November 4. 1990
(401) 863-7600

Contents

1 Introduction

This report documents the initial phases of a study by members of the Brown department of computer science for revision of its computer science curriculum. It is a working paper designed to stimulate discussion, both within the department and externally, prior to committing to a specific course of action. It is the basis for a departmental proposal to the National Science Foundation. By making these preliminary ideas available to a wider audience, we hope to contribute to the national debate concerning curriculum revision, and to obtain feedback for our own curriculum development. Comments on these ideas will be appreciated.

2 Scope of Proposal

We propose to develop a comprehensive introductory freshman-sophomore computer science curriculum that balances theory and practice and makes use of modern technology and modern teaching methods. Our curriculum work has the following goals:

- To develop a two-year introductory core curriculum that balances understanding of fundamental concepts with skills in programming, design, and analysis.
- To disseminate our curriculum and associated instructional software through distribution of our course materials and through faculty training workshops.
- To evaluate the effectiveness of courses through questionnaires, a panel of experts, and an annual review workshop.
- To analyze what can be done to attract students and keep them in the program, and to ease the transition from precollege to college and encourage entry by women and minorities.

Computer science is changing the way we work and the way we think. It has given rise to a new form of conceptual engineering associated with the design and analysis of programs. It is having a profound impact on traditional disciplines such as mathematics and physics, and on broad areas of knowledge like the humanities, the social sciences, and the life sciences. Our work will center on the development of a six-course core curriculum that provides a foundation for a professional or academic career in computer science, and will also serve as the core of a liberal education.

3 Curriculum Overview

We propose to develop a new core curriculum for computer science consisting of a two-course introductory sequence followed by four intermediate-level courses. This core will be required by our AB and ScB degrees. The two-course introductory sequence is designed to convey a concrete understanding of concepts in programming, problem solving, data structures, algorithms, and mathematics relevant to computer science. The remaining four

intermediate-level courses consist of two theoretical courses and two practical courses. The theoretical courses cover algorithms and models of computation and the two practical courses cover computer systems and software engineering.

These six courses provide a foundation for a professional or academic career in computer science, and will also serve as the core of a liberal education for those earning a BA degree. The two-course introductory course sequence can also be used in other degree programs, such as electrical engineering and applied mathematics.

In addition, to make computers and computer science more accessible at the introductory level, we are considering several other computer-related courses:

- A basic (four-week) programming minicourse for those who have not previously encountered computers, to teach basic programming and the use of workstations. This course will be offered in the summer prior to the beginning of the academic year or during the mid-year break, and is designed to to dispel the fear of computers for those who have not encountered them before. Although this course is not part of the regular academic curriculum, it will be critical in determining attitudes to computers and must be carefully designed.

- A computer literacy course refocused on a conceptual understanding of computers and using modern workstation-based tools.

- A course in scientific literacy designed in collaboration with scientists in other disciplines. This course will cover basic ideas in scientific method, physics, mathematics, computer science, and the life sciences. If designed well, it could play a role similar to that played by courses on great books or great ideas in the humanities.

The department is actively participating in discussions of interdisciplinary initiatives with other science departments to make the study of science more attractive, and in university-wide discussions on the overall restructuring of academic units and on the changing role of universities as an increasingly important mechanism for shaping social attitudes and nurturing technology in the 21st century.

4 Human Support Environment for Students

Learning takes place in large measure outside the classroom. We provide several mechanisms for supporting such learning, including interaction with undergraduate teaching assistants and a departmental undergraduate group. Despite these mechanisms we, like most computer science departments in North America, have experienced a sharp decline in undergraduate enrollments since the mid-1980s. In addition, we have experienced an alarming decline in the percentage of women in our courses. We are following with interest the work of the ACM committee on the Status of Women [2], and the efforts of the Computer Research Association to address the issue of attracting women and minorities [28], and are exploring methods to increase the attractiveness of computer science to women and minorities. Below we describe our undergraduate teaching assistantship program, our department undergraduate group, the continuing efforts to make our courses appealing to all students and especially to women and minorities, and our experiments in cooperative learning.

4.1 Undergraduate Teaching Assistants

Brown has employed for the past twenty-five years a very effective system of using undergraduate teaching assistants (UTA's) for support of the introductory courses. These UTA's have taken the course previously and are therefore fully familiar with the content, methodology, assignments and, perhaps most importantly, difficulties with learning new concepts or software. Their most important function is to provide one-on-one tutoring, counseling, debugging help, and moral support during their office hours, which are held three to four hours a night, six days a week. The UTA's also conduct "help sessions" to reinforce especially difficult concepts covered in lecture, or to provide additional information about the programming environment and assignments. Finally, the UTA's also grade the programming assignments and provide the students with extensive written feedback on program design, structure, documentation, efficiency of the algorithm, error conditions not dealt with, and so on. We typically employ one UTA for every 10 students so as not to place too great a burden on the UTA. UTA's deepen their understanding of the subject matter, learn valuable communication skills, and often discover that they have an interest in teaching, which of course we encourage. Over half of our majors grade at least one course in their undergraduate career, and the majority of UTA's do more than one semester of TAing. In short, the UTA's provide a well-functioning and much-appreciated support system for students in the introductory courses.

4.2 Departmental Undergraduate Group

Our department's unusually active "departmental undergraduate group" (DUG) holds weekly social or technical meetings, recruits among the freshmen, and tries to draw students into the department, as well as providing support and advice to majors. More than half of the computer science majors attend meetings regularly.

4.3 Attracting and Retaining Women and Minorities

We have a long tradition of affirmative action in recruiting and hiring UTA's. For example, in the introductory course, among the three head TA's there is always at least one woman and, whenever possible, a minority member as well, to serve as role models. This has not proved sufficient to reverse the declining enrollments of computer science students in general and of women and most minorities in particular. To further increase our effectiveness, we are considering augmenting our UTA system and DUG with interpersonal support in individual courses through cooperative learning.

4.4 Cooperative Learning

Cooperative learning [3,10] is a methodology based on teamwork that shows considerable promise to make computer science education more appealing [13]. It provides a framework for students to work together in small groups and take responsibility for one another's progress, though their efforts are still graded on an individual basis. A current experiment with this technique in a graduate level computer science course is encouraging. Cooperative learning,

as currently practiced, is used primarily in grammar schools, but appears to be effective at the university level in building student confidence and self-esteem through student-to-student learning experiences. We will examine the techniques of cooperative learning more closely and experiment with incorporating them in our new introductory two-semester sequence.

5 Educational Technology

The effective use of up-to-date hardware and software technology for research and instruction has been a hallmark of this department. We pioneered the first workstation-based classroom in 1982 and have used it in our introductory courses since 1983. Our first such classroom was populated with 55 Apollo workstations, each a 0.5-MIPS, 4Mbyte machine with a high-resolution monochrome display and a mouse. In 1989 we installed 65 Sun Sparcstations in a new auditorium. Each of these machines has a 12-MIPS processor, 12M of memory, a high-resolution color display, hardware graphics accelerator, 100M local disk, and a mouse. This facility is still unique, as far as we know.

Our software technology is also world-class. Over a period of years in the early 1980s we developed the BALSA system to animate algorithms [5]. BALSA was originally conceived as a research vehicle to investigate the behavior of complex algorithms through animation, but it was soon adapted for instruction and has been used heavily since 1985 for sophomore-level courses. BALSA is typically used to broadcast demonstrations of algorithms from one workstation to all workstations in a classroom; dozens of demonstrations have been developed. BALSA has established that dynamic concepts are much more rapidly understood if shown dynamically rather than with the static means previously available: dry code and mathematics come alive in a BALSA demo.

The task of producing BALSA demonstrations, however, is daunting, since programs must be written at a very low level. Recently, PhD student J. Stasko developed a more advanced algorithm animation tool. TANGO [25], that significantly reduces the time required to produce animations by providing the user with key animation building blocks. TANGO is now being used in courses at the undergraduate and graduate levels. Both BALSA and TANGO illustrate our experience in and commitment to migrating research systems to instruction, a commitment that has been institutionalized by creation of the position of Manager of Research and Educational Software.

We also have a tradition of placing the latest software technology in the hands of undergraduates in our algorithms, compiler, database, computer graphics and software engineering courses. For example, students have used for their assignments and projects advanced compiler generation tools, object-oriented databases, 3D animation generators, user interface toolkits, and other software technology.

Another example of migration of a research tool to instruction is the FIELD programming environment [20] which is now used throughout our curriculum in courses using Pascal, C and C++ at introductory and advanced levels. FIELD provides a framework for workstation-based program development under UNIX. Using a centralized message server and high-quality visual interfaces, it integrates all the standard UNIX programming tools including debuggers, editors, and make and profiling facilities. In addition, it provides a variety of program visualization tools to make possible the automatic and dynamic visualization of a

large variety of data structures, the display of call graphs as well as class hierarchy browsers.

Computer graphics adds excitement to introductory computer science courses. Since the media expose us all to sophisticated computer graphics, it is desirable to incorporate this technology in instruction. Workstations provide the means to do this and students are excited by the prospect of working with graphics. Our introductory-level students build relatively sophisticated 2D graphics applications in their first courses and we provide interactive 3D capabilities in our undergraduate graphics course.

To support visualization, we will continue to expand the development of tools for building animations. The new tools will be aimed at making more sophisticated animations possible and making such animations much easier to produce. We expect to develop a successor to TANGO over the next two years that will make it possible to design high-quality 2D animations interactively by demonstration in an hour or less. We also expect to greatly reduce the time required to produce 3D animations, a notoriously difficult task. To simplify the visualization of complex programs, we plan to extend the FIELD software development environment to provide displays for abstraction-based program viewing and more intelligent display of complex data structures.

We also intend to exploit parallelism as it becomes more readily available. Parallel machines will become much more common as the demand for computational power exceeds what can be provided at reasonable cost with conventional architectures. We plan to introduce our first-year students to simple examples of parallel algorithms.

All computer science curricula today should be supported by modern technology. We are fortunate in having a state-of-the-art instructional environment. However, our experience with this technology has taught us that we must upgrade our environment every three to four years to take advantage of advances in computing technology. We therefore are requesting funds in the second year of this project to purchase new equipment (see budget justification).

6 Curriculum Options

6.1 Literature on Computer Science Curricula

We briefly review the literature on computer science curricula to provide a yardstick against which our curriculum proposal can be measured. The first computer science departments emerged in the 1960s, and the first comprehensive curriculum proposal, "Curriculum '68" [22] provided a framework for the uniform and widespread development of computer science programs in the 1970s. "Curriculum '78" [23] presented a pragmatic, professionally oriented framework whose introductory courses were defined in greater detail in [14,15]. This curriculum was criticized, however, on the grounds that it emphasized short-term goals at the expense of long-term foundations. The role of mathematics in computer science was examined in [19], and a liberal arts curriculum emphasizing principles and foundations was developed in 1986 [11].

The late 1980s have spurred renewed interest in computer science education, in part because of declining enrollments. The report of a task force on core computer science [7], known as the Denning Report, divides computer science into nine areas and suggests that each area be examined from the viewpoint of theory, abstraction, and design. A controversial

article by Dijkstra on teaching computer science [9], which advocated that the first course teach reasoning about programs without using computers, spurred responses by a number of eminent computer scientists suggesting alternative introductory approaches. Papers by Loui [17] and Parnas [18] advocate an engineering-oriented approach that uses rigorous engineering principles in the teaching of computer science. The joint ACM/IEEE curriculum task force, which has worked for several years on a successor to Curriculum '78, will issue a report in 1991 [1,6] presenting a flexible curriculum specified in terms of modules and following the Denning Report in its general outlines.

Educators are on the defensive because enrollments are declining, and are divided in the educational strategies they advocate. Dijkstra and Parnas advocate strategies driven by a single principle, while Denning advocates a balanced approach that entails exploration of each subfield from the point of view of theory, abstraction, and design. Our department's tradition of balancing theory and practice, which is a guiding principle of our proposed curriculum, is close to that of the Denning Report.

6.2 Introductory Course: Alternative Approaches

We contrast the traditional Pascal-based approach to a newer Lisp-based approach, and then examine the Curriculum '78 approach to a two-course introductory sequence. The goals of CS1 and CS2 in Curriculum '78 are similar to those of our proposed two-course introductory sequence in their objective of combining practical programming with an introduction to foundations.

6.2.1 Our Current Introductory Course

Our current introductory course, developed by Andy van Dam, is a fast-paced introduction to top-down design and structured programming using Pascal on high-performance workstations. It also introduces interactive graphics using a subset of QuickDraw. The course is driven by a sequence of nine progressively more complex programming assignments culminating in an independently chosen final project. It culminates in a substantial final project often requiring more than 1000 lines of Pascal code. Examples of popular final projects are mate-in-one, an adventure game, and a MacDraw-style drawing editor. Assignments are carefully graded with copious written feedback by undergraduate teaching assistants; grading includes marks for program structure and for documentation. The course includes an introduction to searching, sorting, graph algorithms, and geometric algorithms such as line clipping. In moving to a two-semester introductory course, we will shift the emphasis from procedure-oriented programming to data abstraction and object-oriented programming, provide more effective software and visualization tools, and informally introduce fundamental concepts that facilitate a smooth transition to the second introductory course.

6.2.2 The Abelson-Sussman Approach

Courses using the Lisp-based text *The Structure and Interpretation of Computer Programs* [4], which was developed for the introductory course at MIT, are becoming an increasingly popular alternative to the traditional approach. We have taught this course experimentally

and are in close touch with Hal Abelson, one of the authors of [4], concerning how it is taught at MIT. The authors' plan to revise the book by introducing concurrency at an early stage and adding more material on object-oriented programming is similar to our plans for revising our first course to match the requirements of our proposed curriculum.

The Abelson-Sussman text [4] starts directly with interactive expression evaluation, definitions that enrich the environment of expression evaluation, and the substitution model for function evaluation. These fundamental computational ideas can be introduced directly without appearing to introduce a language because LISP expressions and definitions have such a simple syntax. Recursion, language interpretation, and even modularity, data abstraction, and object-oriented programming can be developed simply and naturally.

It is difficult to compare the educational value of good Pascal-based courses like that at Brown with good LISP-based courses like that at MIT. Neither appears to dominate the other. Our two-semester course will experimentally combine the software engineering advantages of our Pascal-based course with the appreciation of structure and interpretation of computer programs of the MIT-based course, and a grounding in the analysis of algorithms.

6.2.3 The Curriculum '78 CS1 and CS2 Approach

The courses CS1 and CS2 in Curriculum '78 and revised in [14,15] are similar in spirit to our introductory sequence. The goals of CS1 [14] are as follows:

1. to introduce a disciplined approach to problem-solving and algorithm development
2. to introduce procedural and data abstraction
3. to teach program design, coding, debugging, testing, documentation, programming style
4. to teach a block-structured, high-level programming language
5. to provide familiarity with the evolution of computer hardware and software technology
6. to provide a foundation for further studies in computer science

Entering students should have completed four years of high-school mathematics including algebra 2 and trigonometry. Most computer science majors are expected to have taken one or more programming courses in high school, and should expect to spend time becoming familiar with terminals and editors at the beginning of the course if they have not.

CS2 [15] has the following goals:

1. To reinforce disciplined design, coding, and testing of programs
2. To teach the use of data abstraction using as an example data structures other than those normally provided as basic types in current programming languages, for example, linked lists, stacks, queues, and trees
3. To provide understanding of the different implementations of these data structures

4. To introduce searching and sorting algorithms and their analysis

5. To provide a foundation for further studies in computer science

It is expected that the student taking CS2 will have completed CS1 and a discrete mathematics course. The student completing CS2 will be able to successfully design and implement large programming systems involving multiple modules using good programming style.

In the liberal arts curriculum [11] CS1 has similar objectives to those of CS1:84, but CS2 has the following, more ambitious, objectives.

1. to consolidate the knowledge of algorithm design and programming gained in CS1, especially emphasizing design and implementation of large programs.

2. to begin a detailed study of data structures and data abstraction as exemplified by data structures or modules

3. to introduce the uses of mathematical tools such as complexity (O-notation) and program verification

4. to provide an overview of computer science

Our introductory sequence may be viewed as a modernized version of CS1–CS2 that takes advantages of workstations and builds on our better understanding of software technology and fundamental concepts that have appeared since CS1 and CS2 were developed.

7 Introductory Sequence

7.1 General Approach

Following the general guidelines expressed in the "Denning report" [7], we intend to provide an overview of the fundamental concepts of computer science and their application to the development of programs. Thus our approach attempts to combine theory and practice. It shows how problem solving with a computer involves both conceptual modeling and programming methodology. The proposed courses are organized around a graduated sequence of programming projects, which develop progressively more advanced concepts in programming, data structures, algorithms. and discrete mathematics.

In short, our goals are:

- To develop basic programming and problem-solving skills, and lay the foundation for good programming style.

- To provide data and procedure abstraction through object-oriented programming.

- To provide an early introduction to fundamental concepts in computer science, and to integrate them into the assignments.

- To accommodate both those who want only a one-semester terminal course, and those who want a deeper understanding of the subject, including prospective majors. We plan to make our first course so attractive that it will draw students into the field who previously had no interest in computer science.

In achieving these goals, we will build on several traditions that make Brown's introductory computer science course especially accessible and attractive to novices.

- The course does not presume any prior background. Students who have been previously unable or reluctant to take a computer course are encouraged to do so, because of the lack of prerequisite. We reassure such novices that, while prior experience may help, it is also often a handicap in that bad programming habits have to be unlearned; thus novices are not at any great disadvantage.

- Assignments are primarily non-numerical, so that only 9th grade algebra is assumed; this helps draw students who feel that they are not mathematically inclined or not very interested in science

- We introduce (mathematical) formalisms gradually and with considerable motivation.

- Because of our visual programming environment and algorithm animation facilities, we appeal to visual intuition rather than mathematical intuition.

Appendix B contains three figures that illustrate the visual programming environments for our current introductory courses. The first figure shows the debugging of a simple linked list program using FIELD. The second figure illustrates a BALSA demo that compares three simple sorting methods: selection, insertion, and bubble sort. The third figure shows a geometric algorithm for the construction of Voronoi diagrams animated with TANGO.

Our examples are primarily two-dimensional, with little need for three-dimensional reasoning. Computer-aided visualization allows students to understand relatively sophisticated concepts far more rapidly than they could with traditional pencil and paper, whiteboard, or viewgraph technology. For example, our algorithm animation system BALSA has been used successfully to teach graduate level material at the sophomore level. Also, our sophisticated software development environment, with its visualization of data structures, is a powerful tool that alleviates much of the drudgery involved in entering and debugging programs. In short, our use of advanced technology paradoxically helps to make computer science more accessible.

An innovative aspect of our proposed introductory sequence is our emphasis on modern technology, both in the course contents and organization. The concepts of parallel computation will be introduced and made concrete through programming assignments; thus students will be exposed to the rudiments of "parallel thinking" early in the curriculum. (Our preliminary experimentation on teaching parallel computation within a sophomore-level algorithms course has been successful.) Computer graphics will be used to support the visualization of algorithms, programs, and mathematical structures, thereby making computer science concepts more readily accessible to students. We can build on our considerable experience in using algorithm and program animation in the classroom. Finally, an integrated environment

will be provided to support all the software development activities, the same environment as will be used in more advanced courses in the curriculum.

In the remainder of this section, we provide one possible scenario for our proposed introductory course sequence. It is derived from the existing introductory courses (the first and third), so that we are confident this type of construction can be made to work.

7.2 First Course

The first part of the course sequence will be devoted to providing the students with basic programming knowledge. This will include variables, control flow constructs, procedures, and parameter passing. Evaluation of Boolean expressions will lead to a discussion of logic and Boolean algebra. Programming shall be taught as a design activity targeted to problem solving. There will be a strong emphasis on modularity and abstraction as well as on elements of programming methodology such as good documentation and readable code. Elementary concepts of computer graphics will be covered and exercised in a programming assignment to design a menu-driven graphical editor.

Once the students have a basic programming knowledge, we will begin to teach some elementary data structures and algorithms. The next portion of the course will teach arrays and array-based data structures including stacks, queues and sets. As part of this discussion we will introduce the basic mathematical concept of sets. These data structures will be used to introduce the notion of algorithms using searching and sorting as an example. We will cover elementary searching and sorting methods, such as binary search, selection sort, and insertion-sort. An especially helpful visualization of sorting algorithms is obtained by showing their execution by means of a matrix of colored dots [5]. The rows of the matrix are associated with the contents of the array at successive stages of the computation, and the color of each dot is proportional to the value of the corresponding array element. The color patterns so obtained are both instructive and fun to observe. The discussion of sorting methods will introduce the concepts of running time and the growth rate of functions (including big-O notation). To reinforce these concepts, the students will do a programming project that involves sorting, in which the running times of sequential and binary search will be compared; here we will introduce the use of very simple recurrence relations in running time analysis.

To introduce the notion of parallel computation, we study the problem of finding the maximum element in an array. Students will implement a simple parallel solution that draws motivation from a sports tournament. In this context, students are first exposed to the issues of processor utilization and communication, and the usefulness of the divide-and-conquer paradigm. This initial exposure to concepts in parallel computation will give the students a taste of "thinking in parallel" and will prepare the way for further study of parallelism later in the course sequence.

Recursion is a key concept and will be used in a graphics painting program to fill a selected region bounded by black pixels. A recursive algorithm will be discussed and compared with nonrecursive stack-based or queue-based algorithms by means of graphic animation.

The final segment of the first course will introduce the students to pointers and some elementary data structures that use them. The notion of a data abstraction that includes both the data and the operations on the data will be emphasized. This will lead naturally

to the notion of object-oriented programming and the use of methods and inheritance. The programming project here will have the students use simple linked list and queue objects in implementing a simple database system.

A capstone final project of approximately one-thousand lines of code, chosen from half a dozen standard ones or devised independently, will provide the student a chance to put many techniques learned in the course together.

7.3 Second Course

The second course sharpens programming skills and provides a deeper coverage of algorithms and problem-solving techniques. By the end of the course, students should be able to write well-structured programs of significant size and complexity that include nontrivial algorithms such as maximum flow computations.

The course will revisit sorting and searching, exploring more sophisticated approaches to sorting such as heapsort, mergesort, and quicksort. We shall discuss the divide-and-conquer paradigm more deeply and analyze running times. A project on implementing mergesort in parallel will reinforce understanding of binary search and recursion, two concepts covered in the first semester.

Building on the students' experience with linked lists, we will show them how to use pointers in maintaining more sophisticated data structures based on trees. Students will develop a program for maintaining a family tree. We will introduce balanced search trees, data structures for sets that support fast search, insertion, and deletion, and hash tables and their use in representing sets. These data structures will be used in implementing an interpreter for a simple graphic language.

Geometric algorithms, such as detecting if two segments intersect, or testing if a point is inside or outside a polygon, will be developed and used to implement a convex-hull algorithm. We shall motivate the study of convex hulls with concrete problems, such as enclosing a set of locations with the shortest possible boundary, or identifying clusters of points in an image. Constructing a convex hull provides a first example of how to combine several algorithms in the efficient solution of a new problem.

In the last part of the course, we will take the use of pointers one step further, and introduce the concept of graphs and graph algorithms. Students will implement a maze-search program and learn about connectivity and transitive closure, and the related mathematical concepts of equivalence relations and partial orders. We will explore the use of graphs in modeling networks, and we will teach associated graph algorithms: minimum spanning tree, shortest path, and maximum flow. These algorithms will be applied in the final programming project.

The introductory sequence culminates in a relatively large programming project involving an interactive airline query system. This project illustrates the application of a variety of algorithms and data structures to a real-life problem that can be modeled by a dynamically changing network structure. Algorithms used in this project include shortest path, connectivity, maximum flow, and dictionary maintenance.

7.4 Tools

Programming language: Ideally, the programming language we use should combine many virtues: it should be procedural, yet also support object-oriented programming. It should be simple enough for beginners to use, yet support extensions for SIMD parallelism and libraries for graphics and user interfaces. We will begin by using an extended version of Pascal, but hope eventually to use a programming language that better fulfills all these requirements and may become widely available.

Programming environment: The FIELD visual software design environment integrates a large number of programming tools, including debuggers, profilers, and a layout facility that displays subroutine-call graphs and data structures. The environment is easily configurable, and can be adapted both for the simplicity required in an introductory course, and for the sophistication required for advanced programming courses. We are already using this environment throughout our curriculum. As part of this proposal, we plan to extend the FIELD environment to support enhanced program visualization and additional tools. In particular, we will provide the students with debugging and visualization tools for SIMD parallelism.

Visualization environment: We plan to extend the graphic library used in the programming assignments of the current third-semester course and the "active" features of the TANGO algorithm animation system by developing more sophisticated interaction capabilities.

The primary problem in using TANGO or BALSA today is the difficulty of creating new animations and of making these animations visually appealing and efficient enough to be meaningful. We plan to address these problems by developing an extension of or replacement for these systems that is based on a visual, programming-by-demonstration interface for creating animations, and an advanced graphics library that will provide an efficient backend for this interface. At the same time, we will begin to look into providing higher quality animations including the use of 3-D graphics.

Another extension of our visualization tools will involve their use in an interactive environment where users wish to experiment with own input and algorithms. The animation of graph and geometric algorithms should allow the user to interactively construct a graph (or a point-set) on the screen, and experiment with various user-generated or library-based algorithmic modules. Current state-of-the art tools do not adequately support this interaction. In TANGO, for instance, the user has to create a separate animation code for each algorithm, and it is difficult to apply two different algorithms to the same animation window.

We plan to overcome these limitations by developing an integrated tool for editing and animating graph and geometric structures which communicates with external programs through a message-passing scheme. This tool would also provide random-graph generators and automatic layout algorithms. Our previous research work on graph drawing systems suggests the feasibility of this approach [8,26].

8 The Systems Core

The system core has two components: system architecture and software engineering. The system architecture course will provide an understanding of computer architecture and operating systems, while the software engineering course will provide a foundation for the design and implementation of large programs.

8.1 System Architecture

Computer architecture and operating systems are core areas of computer science that have been in our curricula for decades. Numerous high-quality textbooks have been written for both subjects. Any well-rounded graduate of a computer science department is expected to be familiar with the concepts of both areas. However, the very success of computer science as a discipline has reduced the prominence of these two courses in our curricula. These two fundamental courses have become electives and as electives their subject matter is not being learned by all of our students.

We will develop a single required "systems architecture" course that includes fundamental concepts of both architecture and operating systems. This will allow us to teach in a unified framework concepts (such as virtual memory) that are traditionally addressed in the separate courses from different points of view.

This course will serve two purposes: it will expose students who go no further in either architecture or operating systems to these areas, and it will provide the foundation for more advanced study. The content of this course, given its importance, must be determined with care. It must be broad enough that students who do not take further courses in the area have a basic understanding of how computer systems work. At the same time, the course should cover new and emerging technologies so that our graduates will be able to keep up with the field.

To give the course sufficient breadth, a single, complete system will be studied in depth and will form the basis for analysis and assignments. The architecture and operating system of a specific computer will provide students with a thorough understanding of its structure and with an appreciation of important system concepts.

A problem with this sort of approach is chosing the appropriate computer: it is difficult to find one computer that is sufficiently representative to be our sole object of study. Furthermore, as the field evolves, the computer of study must evolve as well. Our approach will be to use a simulated computer, sufficiently realistic to be of interest, but simplified to the extent necessary to allow us to cover it in a sophomore course. This approach will allow us to evolve the architecture without having to make major purchases of new hardware.

Our simulated computer will possess enough functionality to support a realistic operating system, so that students will develop an appreciation of the complexity of real systems without being overwhelmed. To give students a sufficiently real exposure to operating systems, we will use a UNIX variant, such as Tannenbaum's Minix or Comer's Xinu.

The course will consist of lectures and laboratory sessions. In the former we will introduce concepts to students, showing the "big picture" of how these concepts are important to the complete design. In the laboratory sessions the students will investigate the details of how these concepts are put into practice and, in some cases, study alternative approaches.

We will employ hypertext techniques to enhance the educational value of our simulator. Students looking at an I/O driver, for example, will be able to follow "links" to descriptions of I/O architectures and to comment on the design of the driver being investigated.

Despite the educational value of gaining a complete understanding of one representative system, we must also introduce students to concepts and approaches either absent from or not apparent in our simulator. For example, students should be aware of non-von Neumann architectures such as hypercubes and SIMD designs; students should appreciate the differences between RISC and CISC architectures. Even though we cannot cover them in detail, students should be aware of issues in distributed computing and security. Where appropriate, these concepts will be discussed throughout the course as we talk about alternative designs.

Combining architecture and operating systems into one course not only forces us to rethink what the most important concepts are, but also lets us economize by avoiding duplicate discussions of many concepts currently dealt with in both architecture and operating systems courses. A systems architecture course allows us to give a more natural treatment of a number of subjects (such as I/O, context switching, virtual memory) that involve both architecture and operating systems issues. Our use of a simulator (whose development is already underway – a prototype is in classroom use this year) will make the learning experience more efficient and interesting for students.

8.2 Software Engineering

Software engineering is concerned with the management of complexity in programs that are large, long-lived, and involve many people [27]. The management of abstraction and the effective use of available tools lie at the heart of software engineering. Our software engineering course will develop programming skills through advanced programming and design in an object-oriented programming language, software management skills through team programming projects that address the specification, design, implementation, and maintenance phases of the software life cycle, and tool usage through the use of a parser generator such as yacc, a database system such as Ingres, and a user interface toolkit such as Motif.

Our current undergraduate software engineering course systematically develops the notion of abstraction, starting with procedure abstraction and continuing with data abstraction and iteration abstraction. It emphasizes specification and design, which may also be viewed as forms of abstraction, and explores the programming methodology for testing and debugging. It also includes an introduction to formal specification and verification. It has three progressively more difficult assignments, the third being a major project involving team programming.

The new course will be organized as a set of individual and team programming projects. Each project will involve developing appropriate specification and design documentation as well as a working implementation. The projects will be chosen to build on top of one another both directly and through modification. In addition, the projects will require the students to understand the basic principles behind and the simplified use of various available programming tools such as yacc, Ingres and Motif, The course will culminate with a large final team project that attempts to integrate the overall material in the course.

One of the difficulties with current software engineering courses is the choice of a language

that provides a good foundation for software engineering issues. The book by Guttag and Liskov [16], which presents the issues more clearly than traditional software engineering texts, uses CLU which is not a mainstream language. C++ has been the choice at Brown, since C is the preferred heavy duty programming language for advanced computing courses at Brown and other academic institutions. But C++ does not have a good typing system and is flawed as a basis for modular programming. Ada, another possible choice for a base language, is so complex that it gets in the way of clear presentation of concepts of modularity. We hope that a good candidate language for software engineering will emerge in the next few years. Brown's active research program in object-oriented programming has as one of its goals the development of such a language [27].

The emphasis throughout the course will be on software design. We will continue to stress various forms of abstraction, including procedure abstraction, data abstraction, and iteration abstraction. We will also teach the students the basic techniques of object-oriented design. The course will help the students to understand the variety of different design philosophies and when they should be applied. It will also teach analysis of the design, including the proper selection of algorithms and data structures, and the use of elementary complexity analysis.

The course will make use of visualization and programming tools being developed at Brown and elsewhere. Students will use sophisticated programming tools such as the FIELD programming environment, upgraded with appropriate visualizations. Research projects at Brown include tools for visualizing program designs and abstractions. We will migrate such research tools into a classroom environment for this course.

9 The Theory Core

We propose a new two-course theory core sequence that would normally be taken during a student's second year of study in computer science. The first deals with the design and analysis of algorithms and data structures from several novel perspectives. The second studies issues of logic, computability, and complexity; several concrete computer languages are studied for illustration. These courses are discussed in the following two subsections.

The motivation for the theory curriculum is to provide the students with knowledge and critical methods of analysis and intuitions. Theoretical and analytical skills are necessary to adapt to rapidly changing technologies and can protect students against obsolescence.

9.1 Algorithms and Analysis

The first of the two-course theory core deals with the design and analysis of algorithms and data structures. We envision a novel approach in which the class would cover four algorithmic modules from several different perspectives.

The first perspective that permeates the course deals with the machine models used. We plan to treat each module simultaneously from both sequential and parallel points of view. When relevant, communication issues will be covered, such as I/O between internal and external memory, which are often a bottleneck in large scale computations.

The second perspective concerns deterministic versus randomized computation: randomized algorithms can often yield significant simplicity of design and reduced processing speed. We also deal with exact versus approximate algorithms and heuristics: in many cases, such as NP-complete problems, approximation algorithms are the only feasible solution methods available.

The third perspective concerns the type of analysis used. We will not only cover the traditional worst-case, big-oh analysis, but also emphasize the important constants hidden in the big-oh notation. We will also discuss techniques for average-case analysis. The student will be well prepared to explore these perspectives further in later courses if desired.

Assignments will consist of both written analytical questions and two or three programming assignments.

The algorithms content of the course is reflected in four basic modules designed to provide a more advanced appreciation of efficiency issues.

1. The first module treats sorting and order statistics, key software building blocks as well as excellent illustrations of the use of analysis. Parallel sorting will concentrate on parallel processors and special-purpose sorting networks. We will also cover external sorting, in which the input is too large to fit in internal memory and the bottleneck of the computation is the I/O; such applications make up roughly one-fifth of computing resources on large-scale business systems.

2. The second module deals with searching and database operations. It is a natural successor to the first module and includes hashing, balanced and unbalanced binary trees, external tries and database access, and priority queues.

3. The third module presents graph algorithms at a higher level than in the first-year introductory sequence and reviews graph models and basic algorithms. It also covers important applications, such as graph matching, maximum flow, stable marriages, and graph coloring.

4. The fourth module is an extension of the first three in the area of computational geometry and graphics. We will discuss topics such as geometric intersection, multidimensional searching (range trees, segment trees, priority-search trees), Voronoi diagrams, rendering, hidden-surface elimination, and point location.

More advanced topics, which might be covered if time permits, include Gaussian elimination and linear programming (particularly the simplex method), the fast Fourier transform, and cryptography.

9.2 Theory of Computation

Theoretical computer science may be divided into two areas: design and analysis of algorithms and theory of computation. The former is concerned with techniques for designing algorithms and their performance, as described above, while the latter is concerned with analyzing the inherent complexity of problems using suitable abstract models.

The proposed course on models of computation deals with the second of the topics in the concrete manner of a recent book on concrete mathematics [12]. One way of making

models concrete is through programming languages that embody or illustrate specific models of computation. For example, computability will be explored not only through Turing machines, but also through the LISP Apply function, whose ability to execute any LISP program precisely parallels the ability of a universal Turing machine to mimic any specific Turing machine. LISP may also be used to explore formal properties of recursion.

Undecidability results, such as the undecidability of the halting problem for Turing machines, will be presented. The close relation between noncomputability and uncountability in mathematics will be illustrated by showing the similarity of the diagonalization proofs used in demonstrating these results.

Logic will be introduced through Prolog, which provides computational motivation for the study of the propositional and predicate calculus and of the analogy between logic and computation. Formal properties of programming language type systems, including type checking and type inference, will be motivated by examples from the language ML. This course will also include exploration of models of operational semantics, which lies at the heart of any model of computation, and an introduction to notions of denotational semantics.

The models of computation course will borrow ideas from a number of existing courses, including our current courses on theory of computation, programming languages, and logic. Presenting the fundamental ideas concerning models of computation in a concrete but rigorous manner is a challenging task that will require considerable thought and careful organization of the subject matter.

This course will examine the relation between deterministic and nondeterministic models of computation as a motivation for the study of whether P = NP. It will consider reducibility of one problem to another and will motivate the study of complexity classes and of mappings between algorithms in the same complexity class.

10 Related Early Courses

Two areas in which we currently have courses, but which need revision in light of the changes described above, are computer literacy and computer programming in Pascal.

10.1 Computer Literacy

Computer literacy courses today tend to emphasize understanding of software tools rather than principles. However, as software technology finds its way into curricula around the country, it is no longer necessary that departments of computer science offer an extensive treatment of topics such as spreadsheets. Such topics are now being taught by the departments that need them, and non-credit courses in useful software are offered by most institutions' computer centers.

We can now refocus courses in computer literacy to stress the conceptual understanding of computers. Our goal in this course will thus be to give a concrete understanding at an elementary level of the conceptual underpinnings of computer science. A computer-literate person should have a rudimentary understanding of the architecture of a simple computer and knowledge of an introductory programming language and of techniques for developing programming solutions. In addition, this person should understand the principles involved in

design and implementation of certain important types of software, such as word processors, operating systems, spreadsheets, databases, and computer graphics. It is also important to understand the many types of applications of computers in such disparate areas as education and manufacturing and to consider the social and ethical implications of computers. Finally, the computer-literate student should be able to put the computer into historical perspective.

A course of this kind was taught several years ago and the textbook written to support it by a member of this department and two of his undergraduate teaching assistants [21] is still current. Today the course should avail itself of the superior software tools developed at Brown. An excellent new tool called Passé [24], designed to simplify instruction in Pascal programming, is currently being used for this course. In addition, we will use new hypertext-type tools to illustrate the history of computers, as well as the TANGO animation facility to convey an understanding of the principles of word processors, operating systems and databases. Using these tools will reduce the time spent on the technology and leave more time for discussion of principles.

10.2 Introduction to Programming

We currently provide a introductory course in programming for students wish to apply their knowledge to areas other than computer science. We will modify this course to take into account changes in our first programming course as reflected in the two-semester sequence described above. We also will modify it to take advantage of new technology that is available to us here at Brown, in particular, the FIELD software development environment.

Our new two-semester introductory course described above has an orientation which is more appropriate for computer scientists. The concepts taught and examples illustrating them are drawn from those key to computer science. As a consequence, the needs of others may not be as well served by such a course. A better course for them emphasizes data structures built from arrays, avoids pointers, and employs well-chosen examples from the areas other than computer science.

Accordingly, we propose to modify our existing course to cover simple data types, simple data structures based on arrays, and procedural abstraction. We will stress concepts important to numerical computation and introduce computer graphics as a visualization aid in scientific computation. Given the need to manipulate large quantities of data in most professions today, we will introduce set manipulation techniques such as searching and sorting. We will also develop an appreciation for the importance of analysis as a guide to the writing of efficient programs and will continue to insist on good programming methodology.

Instruction in this course will profit from access to some of the advanced tools mentioned above. We will use the FIELD software development environment, thereby developing an appreciation for the importance of good tools, such as debuggers, profilers and make facilities, tools which all serious software developers should be using. Coupled with high performance graphical workstations, such tools will make this course a very attractive one.

11 Undergraduate AB and ScB Degree Programs

Majors will require two additional courses in computer science to earn an AB degree, and seven additional courses to earn an ScB degree. We provide a wide selection of junior-senior level courses in artificial intelligence, algorithms and complexity, computational geometry, computer architecture, computer graphics, databases, languages and compilers, logic and logic programming, operating systems, theory of computation, VLSI design, and several other areas.

Subject to the availability of faculty, we plan to add two additional capstone courses that will augment our project-oriented senior seminar and make our curriculum more application oriented. The first would be a senior seminar in scientific computation and visualization oriented towards the use of computers in large science and engineering applications, while the second would be a senior seminar in non-numeric computation oriented towards the use of computers in the humanities, arts, and business. While the course in scientific computation would require a sophomore-level mathematical and science or engineering background, the course in non-numeric computation would require maturity in a non-scientific application area. These capstone courses would encourage joint majors combining computer science with another discipline.

12 Evaluation Procedures

We will undertake the following evaluation and review activities:

- Carefully designed questionnaires to elicit feedback from students on their understanding of material and on the evolution of their motivations and attitudes towards the course and the discipline.
- Annual written reports by a panel of three outside reviewers who will evaluate course proposals and meet with course instructors at the end of each academic year to evaluate progress for each course and for the curriculum as a whole.
- An annual workshop at which courses will be described and discussed at greater leisure. Among the participants in the workshop will be faculty from other institutions interested in curriculum development, including faculty who wish to implement our courses and faculty involved in other curriculum development projects.
- Written course materials for each course in both hard-copy and soft-copy form. Soft copy can include assignments, self-assessment materials, examination questions and answers, hypertext, and other educational courseware.
- Export of courses to other institutions for teaching on a trial basis, with feedback on both the curriculum and the improvement of course materials to facilitate exportability.

There will be an annual review cycle that carefully documents the progress in each of the six core courses, so that we can systematically build on the experiences of each year in subsequent years of the project. By the end of the third year each of the courses will be

debugged and sufficient public domain documentation will be available for other institutions to adopt the curriculum.

13 Bibliography

[1] "Computing Curricula 1990. ACM/IEEE Curriculum. Draft." Complete document scheduled to be available in November 1990 and to be published in 1991.

[2] "Becoming a Computer Scientist. A Report of the ACM Committee on the Status of Women in Computer Science," *Communications of the ACM* (November 1990).

[3] "Cooperative Learning: Making It Work," *The Harvard Education Letter* 5 (November/December 1989).

[4] H. Abelson, G.J. Sussman and J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill, 1985.

[5] M.H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software* 2 (January 1985), 28–39.

[6] K. Bruce, "Creating a New Curriculum. A Rationale for Computing Curricula." Report for the IFIP WG3.2 Workshop on Informatics Curricula for the 90's, Brown University, April 1990.

[7] P. Denning (Chairman), D. Comer, D. Gries, M. Mulder, A. Tucker, J. Turner and P. Young, "Computing as a Discipline, Final Report of the Task Force on the Core of Computer Science," *Communications of the ACM* (January 1989), known as The Denning Report.

[8] G. Di Battista, A. Giammarco, G. Santucci and R. Tamassia, "The Architecture of Diagram Server," *Proc. IEEE Workshop on Visual Languages (VL'90)* (1990).

[9] E.W. Dijkstra. "On the Cruelty of Really Teaching Computer Science," *Communications of the ACM* 32 (December 1989), 1398–1404.

[10] A.L. Gardner. C.L. Mason and M.L. Matyas. "Equity, Excellence & 'Just Plain Teaching'," *The American Biology Teacher* 51 (February 1989).

[11] N. Gibbs and A. Tucker. "A Model Curriculum for a Liberal Arts Degree in Computer Science." *Communications of the ACM* (March 1986).

[12] R.L. Graham. D.E. Knuth and O. Patashnik. *Concrete Mathematics*. Addison Wesley. 1989.

[13] D.W. Johnson. R.T. Johnson and E.J. Holubec. in *Circles of Learning: Cooperation in the Classroom*. Interaction Book Company, Edina, Minnesota, 1986.

[14] E. Koffman, P. Miller and C. Wardle, "Recommended Curriculum for CS1: 1984." *Communications of the ACM* (October 1984).

[15] E. Koffman, D. Stemple and C. Wardle, "Recommended Curriculum for CS2: 1984," *Communications of the ACM* (August 1985).

[16] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill, 1986.

[17] M.C. Loui, "Computer Science is an Engineering Discipline," *Engineering Education* (1987).

[18] D. Parnas. "Education for Computing Professionals," *IEEE Computer* (January 1990).

[19] A. Ralston, "The First Course in Computer Science Needs a Mathematics Corequisite." *Communications of the ACM* (October 1984).

[20] S.P. Reiss, "Connecting Tools Using Message Passing in the FIELD Program Development Environment," *IEEE Software* 7 (July 1990).

[21] J.E. Savage, S. Magidson and A. Stein, in *The Mystical Machine*, Addison-Wesley Publishing Company, 1986.

[22] ACM Curriculum Committee on Computer Science. "Curriculum 68," *Communications of the ACM* (March 1968).

[23] ACM Curriculum Committee on Computer Science, "Curriculum 78," *Communications of the ACM* (March 1979.).

[24] D. Sklar and J. Vogel, "Passé: A Pascal Animation and Simulation Structured Environment," Dept. of Computer Science, Brown Univ., Technical Report, 1990, in preparation.

[25] J.T. Stasko, "Simplifying Algorithm Animation with TANGO," *Proc. IEEE Workshop on Visual Languages (VL'90)* (1990), 1–6.

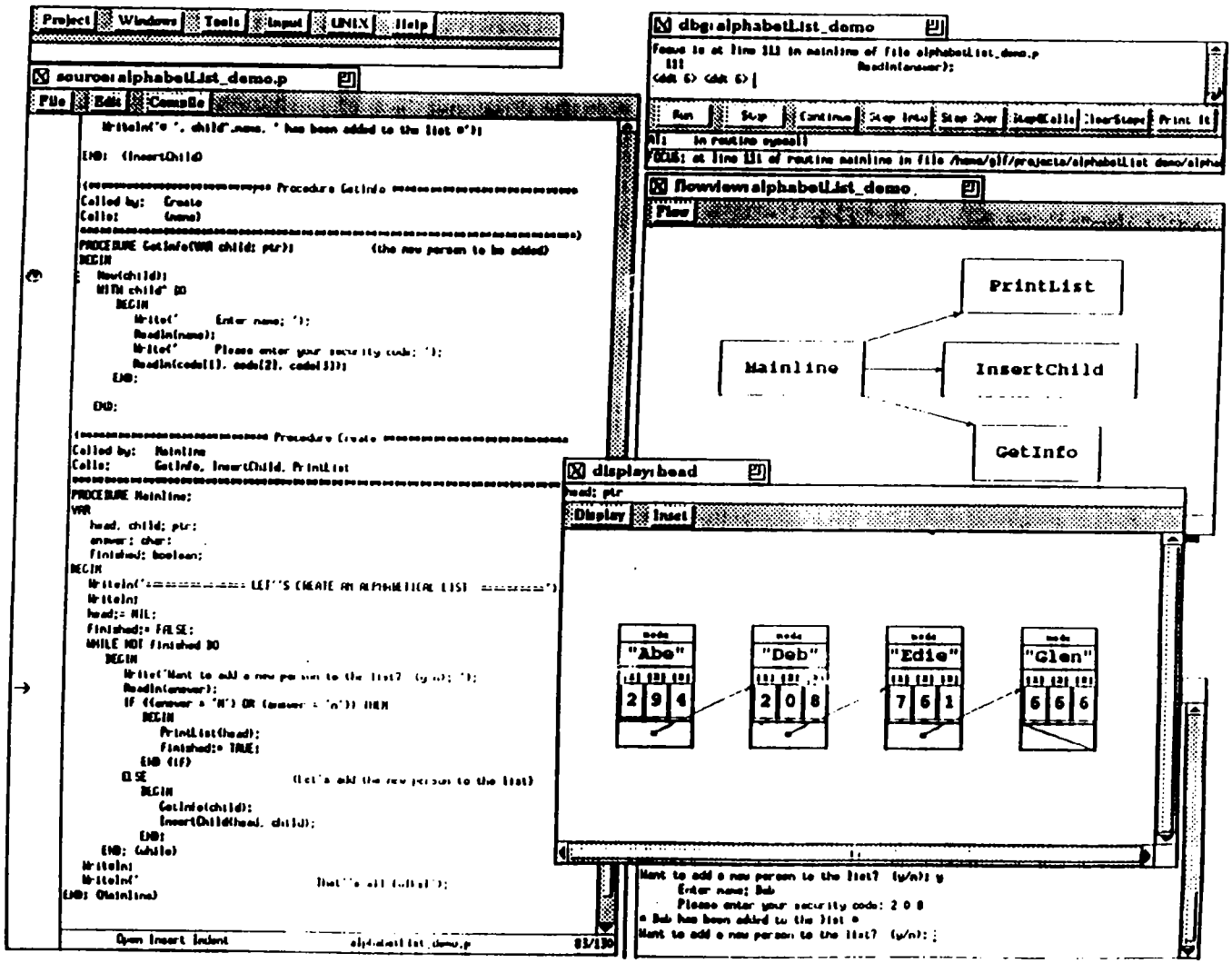
[26] R. Tamassia, G. Di Battista and C. Batini, "Automatic Graph Drawing and Readability of Diagrams," *IEEE Transactions on Systems, Man and Cybernetics* SMC-18 (1988), 61–79.

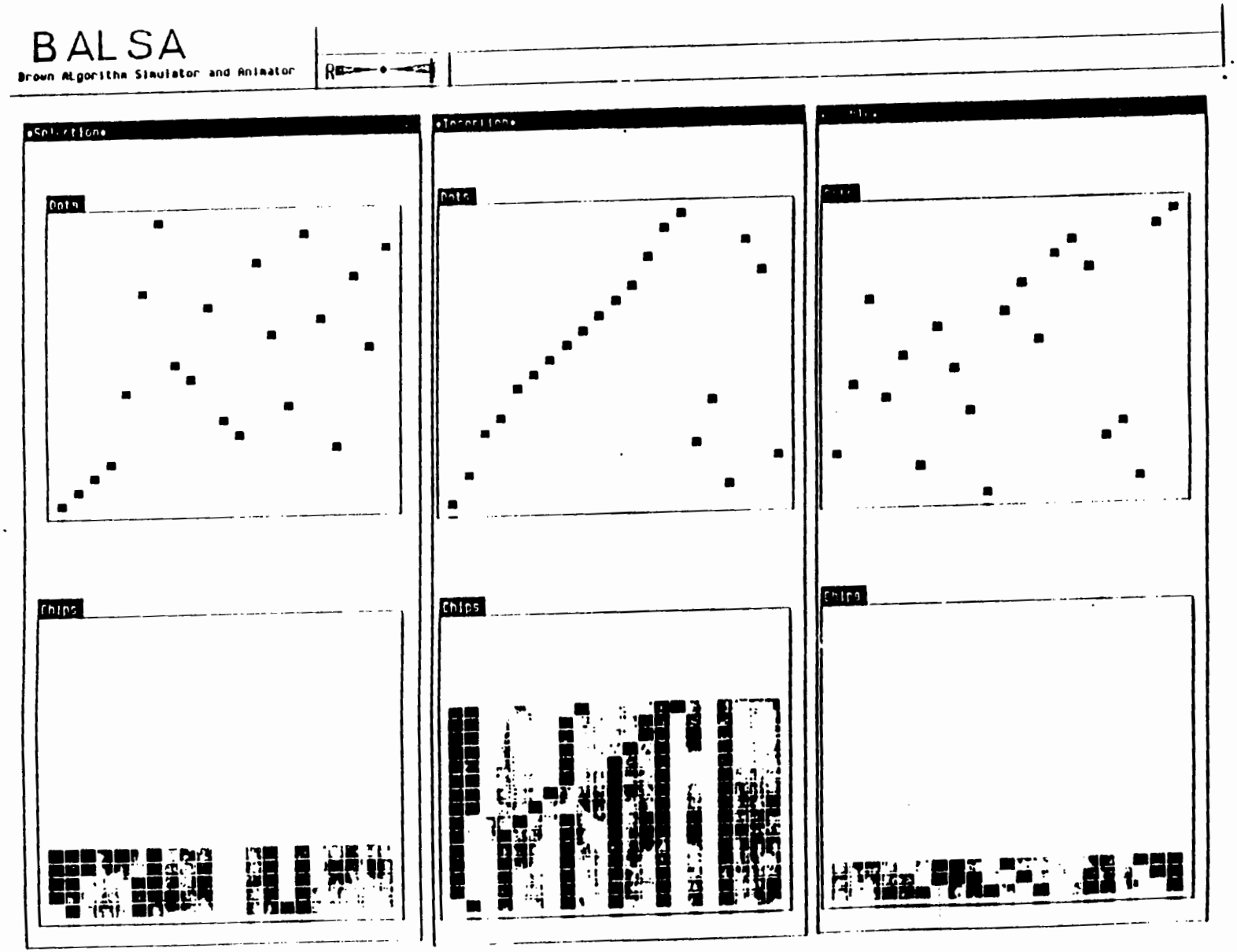
[27] P. Wegner, "Concepts and Paradigms of Object Oriented Programs," *OOPS Messenger* 1 (August 1990).

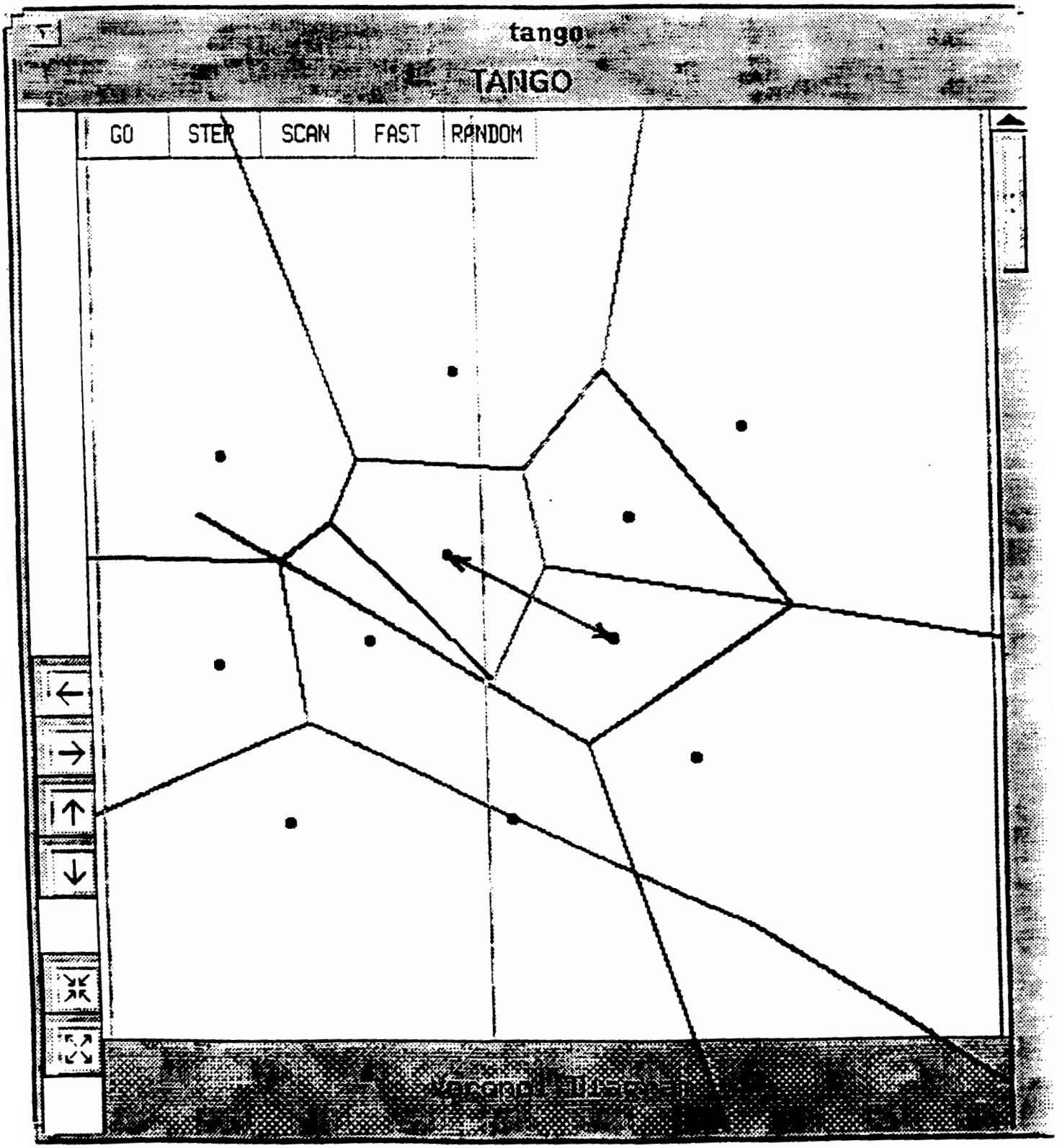
[28] P. Wegner, ed., "Strategic Directions in Computing Research," Joint Report by the Association for Computing Machinery (ACM) and the Computing Research Association (CRA), October 1990.

B. Visualization Examples

1. Debugging a simple linked list program using FIELD.
2. BALSA demo comparing three simple sorting methods: selection, insertion, and bubble sort.
3. A geometric algorithm for the construction of Voronoi diagrams animated with TANGO.


Project
Windows
Tools
Input
UNIX
Help
source: alphabetList_demo.p
File
Edit
Compile
dbg: alphabetList_demo
flowview alphabetList_demo
Flow
PrintList
Mainline
InsertChild
GetInfo
display: head
Display
Inset
"Abe"
"Deb"
"Edie"
"Glen"
Want to add a new person to the list? (y/n): y
Enter name: Bob
Please enter your security code: 2 0 8
Bob has been added to the list
Want to add a new person to the list? (y/n):


BALSA
Brown ALgoritha Simulator and Animator
Data
Chips
Data
Chips
Chips


tango
TANGO
GO
STEP
SCAN
FAST
RANDOM