Foundations of Computer Science: What are they and how do we teach them?

Viera K. Proulx, Richard Rasala and Harriet Fell
Northeastern University
Boston
MA 02115, USA
{vkp, rasala, fel}@ccs.neu.edu

Abstract

Computer science as a discipline is changing rapidly. New developments in software and hardware are changing the way we write programs, design systems, and create applications. The role of the first year curriculum in computer science is to lay the foundations for becoming a professional in the field. We examine the ways in which the changing nature of computer science influences our teaching methods, our view of which concepts are fundamental, and the overall sense of what it takes to become a successful computer scientist. We propose a first year curriculum model that has a strong emphasis on design, on programming in a structured project based environment, and on the extensive use of tools, libraries, and templates. We illustrate this model by describing a collection of graphics-based exercises that apply computing across the disciplines.

1 Curriculum trends in computer science

1.1 Depth first versus breadth first

Unlike mathematics where the freshman curriculum has consisted of calculus for generations, there is a significant debate among computer science educators about what to teach to freshman computer science majors. The two main views have been described as 'breadth first' and 'depth first' ACM/IEEE Curriculum 91 [21]. The proponents of the 'breadth first' view recommend that students receive a broad introduction to computer science including topics such as algorithmics, logic, computer architecture, machine languages, high level languages, compilation, the elements of programming, artificial intelligence, etc. The proponents of the 'depth first' view recommend that programming be the focal topic for the freshman year and that related topics such as algorithms, data structures, and design be motivated as providing a more powerful perspective on the programming process. Both views have merit which is why the debate has not and probably cannot be entirely resolved.

At Northeastern University, we organize the freshman program on the 'depth first' model. We do this from a belief that what brings a student into computer science in the first place is the fact that a computer program designed from pure thought can make the computer do wonderful things. We want to build on this excitement. We also have a practical reason. After the freshman year, our students follow a cooperative education plan in which they alternate academic work with work in industry every three months. It is only by following a 'depth first' model that we can prepare our students to be ready for their first job assignments in the software industry.

Integrating Tech. into C.S.E. 6/96 Barcelona, Spain
© 1996 ACM 0-89791-844-4/96/0009...$3.50

1.2 The importance of design

The rapid developments in both hardware and software that have taken place in the past few years call into question the traditional approaches to computer science education for undergraduates. In 1976, Niklaus Wirth could summarize the essence of software design in the title of his book *Algorithms + Data Structures = Programs.* The philosophy of this text is that a knowledge of the classic algorithms and data structures is the critical component in the education of a computer scientist. The development of complete programs is seen as a relatively simple top-down process in which the functional decomposition of a design problem leads naturally to the use of appropriate algorithms and data structures.

Today, the design and development of programs is more complex and more subtle than the simple model suggested by Wirth's title. To train students to work in a modern software development environment, it is essential to recognize that classic algorithms and data structures are just the starting point. *The design process itself must be a major focus throughout the curriculum.* Traditionally, computer science education has been most concerned with the design and performance of individual algorithms on individual data structures. *It is now imperative that the design of large program structures with elaborate functionalities and elegant interfaces be treated as one of the central problems of computer science education.*

To build an innovative curriculum with a focus on issues of design, it is important to recognize certain critical realities:

- The object-oriented paradigm has become central to the design process [2, 10, 16]. The concept of object combines a data structure with the algorithms that operate on it. Although this combination may at first seem merely convenient, it changes the entire design process from a top-down functional model to a model of objects that interact with one another. The bells and whistles of object-oriented design such as inheritance, function and operator overloading, and templates also make the use of objects more natural and more powerful than classic data structures.

- Graphics and user interface design are now as fundamental to programming as text [1,5,6,9,10,13,14,18,19]. Due to the needs of graphics and user interface design, modern operating systems provide a much richer and more complex set of tools than do classic operating systems. Many algorithms are incorporated into the operating system as standard tools.

- Modern software development environments provide algorithm toolkits and class libraries which extend the underlying operating system toolkits. These proprietary toolkits speed the development process on a particular system but are not as portable as the underlying object-oriented programming language. Extra design effort is required to prepare an application for deployment on multiple systems.

- In some development environments, the starting point for the development of large programs is an application framework (based on toolkits and a class library) which defines the generic behavior of an application without any of its specific details. The programmer's task is to find the critical points in the middle of the framework where new or modified classes can be used to instantiate the desired functionality of the specific application being created.

- Program extensibility is carried one step further by preparing an application to accept software plug-in's which can be inserted into the application after it has been compiled and shipped.

Since the learning curve of objects, graphics, user interfaces, toolkits, class libraries, application frameworks, and plug-in's is formidable, *it is necessary to plan a multi-year curriculum starting in the freshman year* so that undergraduate students can become familiar with the critical ideas and the numerous details at a reasonable pace. To be successful, this curriculum must be based on laboratory experiences not just on classroom teaching and textbooks [1, 6 17, 19, 20].

At the freshman level, an extensive collection of laboratory exercises must be developed which illustrate design issues, explore algorithms, and encapsulate using class structures. These exercises must also be rich in graphics, have high quality user interfaces, and be excellent models of computing applications. These laboratory projects should enable students to experience design-in-context, that is, reaching a software design goal by using a combination of existing code, toolkits, class libraries, and new code developed specifically to meet the particular design challenge.

2 Principles of apprentice based learning

At Northeastern, we have developed a teaching paradigm which integrates a number of ideas in current science curriculum reform with some approaches that are unique to our institution. Our model is an apprentice based approach in which students make meaningful contributions to interesting software products but are not required to program every detail [4]. We describe here the basic principles we have followed in creating an environment to support this approach.

2.1 Robust programming environment

At the foundation of our programming environment are the fundamental toolkits we provide. At present, we have already implemented in C++ toolkits for windows, color graphics, graphics text, convenient and robust IO, and various systems utilities. Starting from this foundation, students can generate graphics on their first day of programming and produce interesting programs within a couple of weeks. We believe that *students must begin programming at a level where concepts are what really matters* rather than on a lower level where language technicalities become the main issue. Using the toolkits, students can get going quickly and then pick up the technical details of the programming language in a gradual and systematic manner.

2.2 Shells for focus

We believe that students should focus on mastering one or two concepts at a time, leaving the remaining tasks to shells and components provided by the instructors. Laboratory projects should enable students to experience design-in-context, that is, reaching a software design goal by using a combination of existing code, toolkits, class libraries, and new code that students develop specifically to meet the particular design challenge. In this way, students spend their time working of the key parts of a project that illustrate the concepts under study.

2.3 Design examples

It is very important for students to be able to read and study substantial bodies of high quality code that solve real problems. Design cannot be learned simply by statements of principles. Students need to examine actual designs in which the general principles are illustrated by solutions to specific design problems. All of the software presented to the students (toolkits, shells, components) should serve as models of excellent design.

2.4 Graphics

Most of today's students have grown up with computers. They have used application programs with sophisticated graphical user interfaces, they have played computer games, and they have explored the World Wide Web. They expect more from a computer than text-in/text-out programs. The student exercises must be rich in graphics, have high quality user interfaces, and be good models of computing applications. However, to be most effective, graphics should not be used simply to get the student's attention or provide motivation. Graphics should be used as an exploration tool, as a means of visualization, and as an aid to debugging.

2.5 Animation

User controlled animation is an excellent method for presenting dynamic processes such as algorithm behavior. Algorithm animations have been created by a number of computer science educators including ourselves [3, 5, 11, 15]. We use such animations for demos in our lectures but also make them available to students who can run them to get an overall gestalt of an algorithm or examine an algorithmic process in a step-by-step fashion. Since all toolkits are open, we also have students create their own algorithm animations.

2.6 Experimental analysis

The theoretical principles of algorithm analysis are difficult for freshman students because of their limited experience with both algorithms and the mathematics required for analysis. We believe that students must acquire actual performance data by executing timed algorithms. To examine this experimental data, we encourage the use of spreadsheets for both numerical and graphical analysis. Using these techniques, students get a concrete understanding of order of magnitude estimates that a mere statement of algebraic formulas cannot provide [15].

2.7 A variety of applications

In addition to exercises that focus directly on computer science issues, students need to experience the variety of ways that computing can be applied to different disciplines. An important component of the projects in an apprentice based curriculum must be exercises that illustrate how the computer can be used to solve real life problems.

3 The software environment

We have built extensive software to support an apprentice based style of learning. This software serves several learning goals. First, it provides a robust infrastructure in the form of toolkits, shells, and other software components. Second, it supports an easily accessible use of graphics and provides examples and models in different settings. Finally, all of the software components may be used as resources for learning about design and for learning how to read programs.

3.1 The toolkits

We describe here the three most basic toolkits we have developed to provide a robust programming environment for ourselves and our students.

3.1.1 IO Toolkit

Robust text-based input is supported by the IOTools package. The input tools in this package will catch every possible IO error, print a polite error message, and then permit the user to input the data again. A typical call to read a double precision number `radius` has the simple form:

```
radius = RequestDouble("Enter radius", 10);
```

Notice that `RequestDouble` handles a prompt and a default value automatically. It is also possible to omit the default and require the user to supply input. A third variation is illustrated by the call:

```
while (ReadingDouble("Enter radius", radius))

        { ... }
```

The function `ReadingDouble` returns `true` if the user provides data and `false` if the user declines to provide data. This allows the user to control whether or not to *execute a loop* based on whether or not the data needed in the loop is provided. It is also possible to control a loop based on multiple inputs:

```
while (ReadingDouble("Enter x", x)

    && ReadingDouble("Enter y", y)) { ... }
```

The point of these examples is that, by using IOTools, input integrates elegantly into the control structures and flow of the C++ language. The philosophy is that input is not just data, it is also control.

3.1.2 Windows toolkit

The windows package supports a model in which text-based interactions occur in a console window and graphics and mouse-based interactions occur in a graphics window. Although these windows can be configured in arbitrary sizes and locations, the ease of use of the package comes from the fact that there are 16 particular named configurations that are designed to handle most of the windows needs for the entire suite of freshman laboratories. Students can quickly set up the windows they need for their projects without reference to system calls or numerical parameters.

3.1.3 Graphics toolkit

The graphics package supports pixel based drawing with a full complement of lines and shapes. The package also supports 24-bit color with facilities to set common colors by name and common shades with a single level parameter. Text in the graphics window and mouse interaction are handled by related packages. All graphics and animations are based on a few basic commands:

- moveto, lineto, and drawline style commands
- paint, frame, erase, and invert commands for rectangles, ovals, and circles
- commands to set foreground and background color

The graphics toolkit assures a uniform user interface as well as orthogonality of the types of arguments that are accepted by the different graphics functions.

3.2 The use of graphics

Graphics plays a very important role in our curriculum. From the beginning, students participate fully in the design of graphic images and learn how to create drawings, models, and animations. Graphics are used for motivation, to assist in the understanding of algorithms, to provide feedback during debugging, to explore user interface design, and to visualize concepts in other disciplines.

3.2.1 Motivation

Graphics is fun. A well designed demonstration or laboratory assignment using graphics can create intense excitement in a class. Students beome interested in learning the design principles that underlie visual effects. They are also pleased to produce products they can show off to friends and family.

3.2.2 Visual feedback

Many projects use graphics to give students visual feedback. In some assignments, the graphics image is the end product of the project and successful completion of an algorithm is demonstrated by the correctness of a picture or an animation. For example, loop animations and recursive fractals provide testbeds for the understanding of key concepts. Graphics output may also represent the evolution of an algorithmic process, for example, progress in a maze search or changes in a queue over a period of time.

3.2.3 User interfaces

By creating graphic images students become aware of the esthetic and functional appeal of their programs. Some of the assignments further explore these issues. Students draw a piano keyboard that is played by mouse clicks and they design a primitive version of a paint program. After completing these assignments, they discuss the design of the icons, their functionality, and the need for error checking. Throughout the year, students discuss the design of the graphics features that are part of their assignments.

3.2.4 Modeling and visualization

A large number of laboratory projects are concerned with modeling and visualization. Some projects explore data and function plotting, others simulation, and others algorithm animation. Students learn techniques for building such models and animations. They see the enormous power of visualizing abstract results and observing dynamic representations of time dependent events.

3.3 Models of good design

All of our software is intended to serve as a model of good design. It is used in lectures, in laboratory projects, and in independent reading as a source of examples, design ideas, and models to emulate. Exploring models of good design is an integral part of our student's learning process.

3.3.1 Reading Well-designed Source Code

In most of our projects, students start by reading some existing code. In the earliest laboratories, they read model programs that they can imitate or modify. As they learn to use functions provided by the instructors, they examine the code available in the main shell and the interfaces to the routines they will use. Later on, they study the standard toolkits and learn to design toolkits of their own.

Some of the toolkits or projects serve as examples that are studied and examined during lectures. The IOTools package serves as an example of input verification and expression parsing. The CharBuffer class which supports safe strings is used as the first example of templates. The Swimming Fish laboratory contains several simple yet non-trivial classes that interact nicely with one another. Many of the early projects are revisited later in the year for an examination of the design decisions that were made in building the project.

3.3.2 Exploring the design alternatives

Many projects are miniature versions of similar applications in real life. The goal of such projects is to make the underlying ideas accessible to the students and to provide a framework for exploring specific concepts. The minipaint project, the piano keyboard, the traffic simulation, and the Game of Life are all examples of such projects. In the classroom or as a post-lab exercises, we challenge students to think about the limitations of such simple designs. We look for features we may wish to add, discuss the ways in which this may be accomplished, observe the choices that were made originally, and consider the constraints these design choices impose on possible future changes.

3.3.3 The debugger

Students should learn to use resources available on the system they work with. This establishes a pattern of lifelong learning and good practice. We use the debugger to explain the difference between value and reference parameters and between pointers and regular variables. We trace function calls and we track pointer traversal of character strings. Many of these concepts are hard to observe directly and the debugger allows us to 'see inside'. Other similar tools or environments may be used for the same purpose.

4 Typical projects

The freshman laboratory projects are designed to focus on one or two main themes while using additional concepts and threads in an auxiliary manner. The permits a student to concentrate on the central issues of the project while integrating material learned earlier and utilizing material that will not be taught formally until a later date. Each project has a different level of emphasis on control structures, data structures, classes, toolkit building, program design, user interface design, and algorithms. It is essential for an apprentice-based approach to learning that over the course of a term students experience all aspects of the software design process.

We will describe some typical laboratory projects and discuss the themes empahsized by these projects.

4.1 Introductory projects

The first programming assignment is to write a C++ program to generate a picture of their own design. During the first closed laboratory session students examine and modify an existing scalable drawing and use that program as a model for their assignment. The requirement that their drawing must scale to multiple sizes forces students to think more abstractly than if they were drawing to absolute pixel positions. In this lab, students use the basic toolkits and learn that code can be separated into functions but they need only write inline code to solve the assignment. Learning by imitation and code reuse are key themes at this stage.

Despite the fact that students know almost nothing about C++ in depth, they produce spectacular drawings. Figure 1 is an example of student work (Ryan Mitchell The Drum Set).

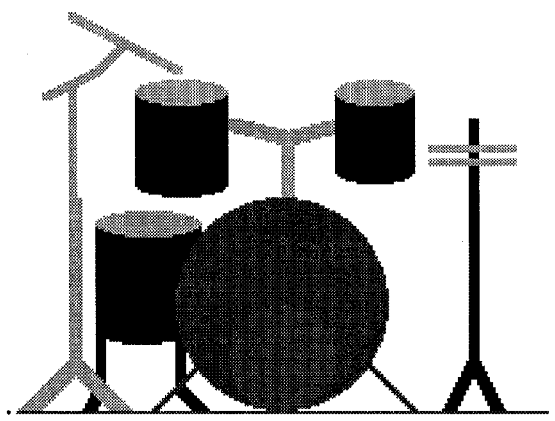

Figure 1 The Drum Set

In other early assignments, students learn about loops and decision statements by animating a rolling or bouncing ball, a floating balloon, or a roulette wheel. They also work with sound by playing a siren, drawing a working piano keyboard, or creating sound effects for the moving balls. The graphics and sound provide tangible feedback in these early assignments and help in the debugging stages.

4.2 Scientific problem solving

These projects examine themes in mathematics and the sciences that can be illuminated by visualization and simulation.

4.2.1 Data visualization

A computer generated contour map allows us to focus our attention at that area of data values that we find most interesting. In a simple exercise, students plot mathematically defined functions of two variables using different colors to represent function values. They vary the ranges for different colors to be able to see better the points near zero, or to see the shapes of the peaks. This technique is the basis of many interesting applications such as the analysis of CAT scans or satellite images of the Earth.

4.2.2 Mathematical functions

In the winter term, students do a pair of labs that use scaled graphics of mathematical functions. These labs also elaborate on the idea of classes. In the first lab, students build a complex number class and test this class with both numerical and graphical feedback. In the second lab, students build a complex polynomial class. In order to test this class, students view a polynomial P as defining a transformation z to P(z) on the complex plane. To visualize this transformation, they plot the image of a circle of a given radius under the action of P.

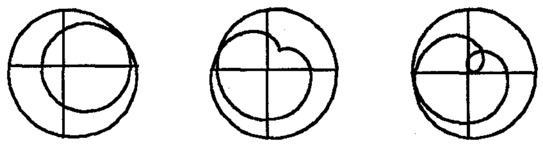

Figure 2 Polynomial Images at Radii 1, 2, 3

Figure 2 illustrates the image of a particular cubic polynomial on circles of radii 1, 2, and 3. In the image of a circle of radius 3, it is clear that the cubic term in the polynomial dominates since the image does a triple loop-the-loop. Similarly, in the image of a circle of radius 1, it is clear from the double loop-the-loop that the quadratic term dominates. The image of a circle of radius 2 shows what happen in the transition zone between cubic and quadratic dominance.

These images provide a visual foundation for discussing the issues of order of magnitude estimates and big-O style notation which are at the heart of later studies in analysis of algorithms. The images also provide an explanation of the fundamental theorem of algebra which states that a polynomial of degree N has N roots. Since a polynomial of degree N loops N times for large radii and contracts to a point as the radius tends to zero, there must be a time when each large loop crosses the origin to yield a root of the polynomial.

4.2.3 Simulations

To learn about queues, students model the movement of traffic through an intersection. They implement the functionality of a traffic signal, the queue that manages the incoming and outgoing cars, and most of the functionality of the traffic lane class which organizes the traffic flow in a lane.

At this stage of their studies the design of the shell is too complex for students to do on their own but they are ready to study the code as an example of classes and abstractions. During the lectures, we explain the graphic design and implementation, the design of the classes for cars, queues, traffic signals, traffic lanes, and the way these classes are interconnected. We then challenge students to think about how the model needs to be modified to simulate four-way traffic or even several road intersections.

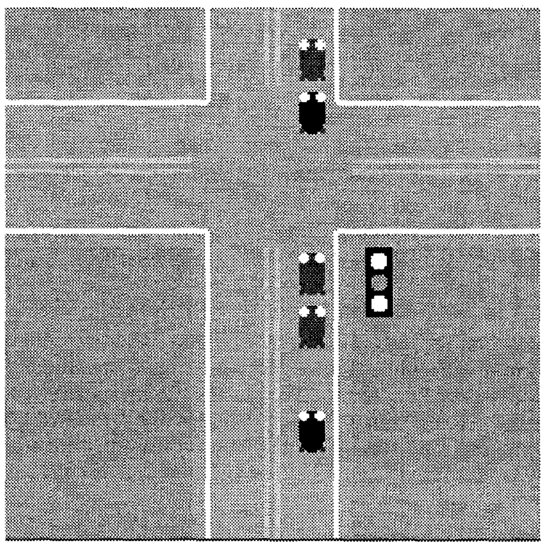

Figure 3 Traffic Simulation

In another project, students simulate a hospital emergency room with a certain number of beds, arrival times, treatment priorities, and expected length of stay. Both of these projects illustrate the use of simulations in resource allocation and planning.

4.3 Algorithm animation and visualization

In these projects, animation is used to visualize the dynamic behavior of algorithms.

4.3.1 Algorithm design: Swimming Fish

The Swimming Fish laboratory is given midway through the first term. Students design an algorithm to move a fish through an underwater cave (maze) to find food. Students program the search and can see immediately whether it works as expected.

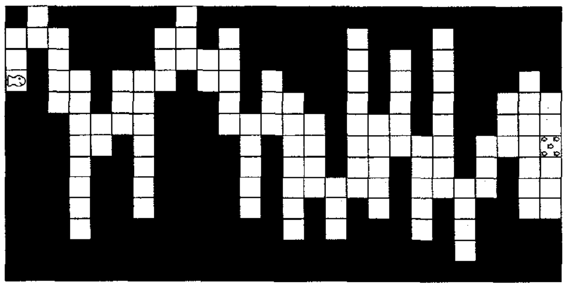

Figure 4 Swimming Fish

The basis for the search algorithm designed by the students are three functions: MoveFish(direction), FreeToMove(direction), and FoundFood(). Students learn the power of abstraction when they realize that they can solve the problem without referring to lower level details about the geomety of the maze and the location of the fish.

The foundations for the Swimming Fish lab are three classes Maze. Fish, and Spot. Later in the course, we use this design to illustrate interacting classes.

4.3.2 Sorting

The first animated sorting movie is 15 years old [3] (Sorting Out Sorting) and many other such animations have been built. We built our own sorting animation to give students maximum control of the learning process. Students can control the size and structure of the data set and the timing (single step, slow motion, or full speed). In addition, the representation of each algorithm is customized to highlight the critical aspects of that algorithm [5,6]. As a result, students can understand the simpler algorithms within a few minutes.

At the completion of each algorithm, the application shows the total number of moves, comparisons, and time ticks. This data can be used to compare different algorithms and to illustrate the best case or worst case examples.

4.3.3 Algorithm analysis

To follow up on the sorting animations, students use a program, Time Trials, that collects timing data for all sorting algorithms. It is an important experience for the students to collect this data since they learn that some algorithms can take a half hour to accomplish what quicksort does in a few seconds.

The data files generated by this program can be imported into a spreadsheet. The students can perform real data analysis and experiments [15]. The spreadsheet charts permit the students to see the gross differences between $O(N^2)$ and $O(N \cdot \log N)$ sorts and the fine details in various implementations of quicksort and heapsort.

In another project, students write their own data collection program for several variants of the Union/Find algorithm and analyze the collected data. This experiment is very impressive. Students learn that by adding path splitting at a cost of one line of code, there is a 100-fold improvement in performance. They see that even a simple optimization may be very effective and is worth investigating.

4.3.4 Recursion

Recursion can be represented at several levels. We hope that by seeing different representations of recursive processes, students will gain deeper insight into this problem solving technique. We use exercises where the goal is to draw simple fractal curves such as the Mandelbrot snowflake or trees with a given number of branches. These fractals are based on a simple abstraction of a Logo-type turtle.

A Towers of Hanoi animation is used in class. The animation illustrates graphically the progress towards the solution and the current contents of the call-return stack [5].

4.3.5 Morphing

In the morphing exercise, students transform a figure drawn by connecting twenty points into a new image using linear interpolation (morphing). In terms of programming, it is a simple exercise on using arrays. The visual impact is very powerful and students learn how morphing animations are created.

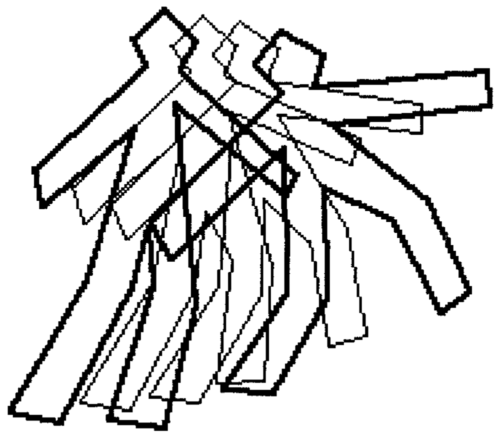

Figure 5 A Morphed Person

Curiously enough, we also implemented this exercise using a spreadsheet where the morphing is controlled by a slider. A similar spreadsheet-based application shows how morphing from an average human face to one's picture and extrapolating beyond can be used to draw a caricature [12].

5 Future development

Our plans for the future development include:

- converting most of the existing Pascal based labs and projects to C++
- creating additional projects that utilize the object-oriented capabilities of C++
- porting the existing toolkits and projects to a Windows based environment
- writing additional support materials such as an instructor's guide and supplementary material related to the computer science topic that is the theme of a particular project
- making all the materials available electronically
- designing a process for collecting data and feedback from those that use our materials

6 Conclusion

We believe that the ability to handle complexity is one of the fundamental requirements for success in computer science. It is not sufficient for students to learn classic algorithms and data structures. Students should learn how to invent and encapsulate new algorithms and data structures. Along with a mastery of building such components of a program, students must learn how to design programs in the large.

In order to design large programs, students need to work with standard toolkits, libraries, and templates and must learn how to create new ones. In the first year of computer science, students must experience numerous examples of good design with well organized internal structures and pleasing graphical user interfaces. It is important for students to see a variety of applications so that they understand the breadth of problems to be solved by computing techniques.

It is our hope that the toolkits and laboratories described in this article will inspire similar efforts at other universities since we believe that computer science students need to be involved in the design process as soon as possible.

The existing C++ materials are currently available on the Web at the URL: http://www.ccs.neu.edu/home/rasala/cpp.html.

References

1 Abernethy K., Allen, J. T. Experiments in Computing: Laboratories for Introductory Computer Science in THINK Pascal, Brooks/Cole, Pacific Grove, CA, 1992.

2 Astrachan, O., Reed, D. AAA and CS1: The Applied Apprenticeship Approach to CS1, *SIGCSE Bulletin,* Vol. 27, No. 1, March 1995, pp. 1-5.

3 Baecker, R. M. and Sherman, D. Sorting Out Sorting, 16mm color sound film, 30 minutes, 1981. (Shown at ACM SIGGRAPH '81 in Dallas, TX and excerpted in *ACM SIGGRAPH Video Review* No. 7, 1983.)

4 Brown, C., Fell, H. J., Proulx, V. K., and Rasala, R. Instructional Frameworks: Toolkits and Abstractions in Introductory Computer Science, *Proceedings of ACM Computer Science Conference*, Indianapolis, IN, February 1993.

5 Brown, C., Fell, H. J., Proulx, V. K., and Rasala, R. Using Visual Feedback and Model Programs in Introductory Computer Science, *Journal of Computing in Higher Education*, Vol. 4, No. 1, Fall 1992, pp. 3-26.

6 Brown, C., Fell, H. J., Proulx, V. K., and Rasala, R. Programming by Example and Experimentation, *Proceedings of the Fourth International Conference on Computers and Learning* (4th ICCAL), Acadia University, Wolfville, Nova Scotia, June 1992.

7 Feldman, T. J., Zelenski, J. D. The Quest for Excellence in Designing CS1/CS2 Assignments, *SIGCSE Bulletin,* Vol. 28, No. 1, February 1996, pp. 319-323.

8 Freund, S. N., Roberts, E. S. THETIS: An ANSI C Programming Environment for Introductory Use, *SIGCSE Bulletin,* Vol. 28, No. 1, February 1996, pp. 300-304.

9 Kurtz, B. L., Mayekar, U. S., and O'Neal, M. B. Design and Implementation of a Generalized Problem Solving Assistants for Algorithm Development, *SIGCSE Bulletin,* Vol. 27, No. 1, March 1995, pp. 97-101.

10 Naps, T. L., Swander, B. An Object-Oriented Approach to Algorithm Visualization - Easy, Extensible, and Dynamic, *SIGCSE Bulletin,* Vol. 26, No. 1, March 1994, pp. 46-50.

11 Naps, T. L., Stenglein, J. Tools for Visual Exploration of Scope and Parameter Passing in a Programming Languages Course, *SIGCSE Bulletin,* Vol. 28, No. 1, February 1996, pp. 305-309.

12 Neuwirth, E. Private Communication.

13 O'Neal, M. B., Kurtz, and Watson, B. L. A Modular Software Environment for Introductory Computer Science Education, *SIGCSE Bulletin,* Vol. 27, No. 1, March 1995, pp. 87-91.

14 Proulx, V. K., Fell, H. J., and Rasala, R. Interactive Animations in Computer Science, *Proceedings of Frontiers in Education 93* (23rd Annual Conference: Engineering Education: Renewing America's Technology), IEEE Press, November 1993, 786-790.

15 Rasala, R., Proulx, V. K., Fell, H. J. From Animation to Analysis in Introductory Computer Science, in *Proceedings of ACM Computer Science Conference*, Phoenix, AZ, March 1994, pp. 61-65.

16 Reid, R J. The Object-Oriented Paradigm in CS1, *SIGCSE Bulletin,* Vol. 25, No. 1, March 1993, pp. 265-269.

17 Robergé, J., Suriano, C. Using Laboratories to Teach Software Engineering Principles in the Introductory Computer Science Curriculum, *SIGCSE Bulletin,* Vol. 26, No. 1, March 1994, pp. 106-110.

18 Roberts, E. S. A C-Based graphics Library for CS1, *SIGCSE Bulletin,* Vol. 27, No. 1, February 1995, pp. 163-167.

19 Scragg, G., BaldwinD., and Koomen, J. Computer Science Needs an Insight-Based Curriculum, *SIGCSE Bulletin,* Vol. 26, No. 1, March 1994, pp. 150-154.

20 Thweatt, M. CS1 Closed lab vs. Open Lab Experiment, *SIGCSE Bulletin,* Vol. 26, No. 1, March 1994, pp. 80-82.

21 Tucker, A. B. et. al. (ed.), Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, 1991.

22 Wallace, S. R. and Wallace, F. J. Two Neural Network Programming Assignments Using Arrays, SIGCSE Bulletin, Vol. 23, No. 1, March 1991, pp. 43-47.