J. EDUCATIONAL COMPUTING RESEARCH, Vol. 9(3) 413-443, 1993

THE DESIGN OF SOFTWARE TOOLS FOR MEANINGFUL LEARNING BY EXPERIENCE: FLEXIBILITY AND FEEDBACK*

DAVID F. JACKSON
University of Georgia

BILLIE JEAN EDWARDS
Detroit Public Schools

CARL F. BERGER
University of Michigan

ABSTRACT

Experience in using commercially available software to teach students about principles of graphical data analysis suggests that several critical design modifications are advisable. In a quasi-experimental study, three different versions of an original graphing program were used by inner-city high school students solving scientific data analysis problems. A version incorporating "coaching" feedback into a highly flexible interface was found to be significantly superior to either an "open" version giving no extrinsic feedback or a "restrictive" one that disabled program options whose use was deemed inappropriate based on the data analysis context. As an illustration of one of the graph-based critical thinking skills developed by the students, results are presented as contrasting pairs of graphs in which one is designed to emphasize, and the other to downplay, the effects of interface design, gender, and their interactions.

Discussions of educational uses of computers commonly distinguish between three distinct roles for the computer [1]: first, as personal tutor; second, as a medium for experiential learning, especially through programming ("tutee" [2]); and third, as a multipurpose tool. The first two of these categories have frequently been the focus of research, and existing software spans a great range of sophistication in both

*An earlier version of this article was presented at the annual meeting of the National Association for Research in Science Teaching, Atlanta, Georgia, April 9, 1990.

cases: from simple drill-and-practice exercises to "intelligent" tutoring systems and from student-designed programs to expertly prepared simulations.

Recently, more attention has been paid to the "tool" software most commonly used in science education, microcomputer-based laboratory (MBL) materials. In particular, the results of research on teaching about line graphs representing physical phenomena have been more consistently positive than in any other area of computer use [3]. The software used in this study is more general in scope, providing for the use of four different major types of graphs (column/bar, line, pie chart, and scatterplot) with any data in the form of a standard cases-by-variables matrix. An important distinction is that both the data and the graphic representations are static, as opposed to the dynamic, real-time graphs typically generated in MBL activities.

While the use of more general applications programs (word processors, databases, spreadsheets, graphing utilities, etc.) in schools is increasing, the effects of the characteristics of these programs on subject matter learning have not often been critically examined. What are the consequences of using such software as a vehicle for experiential learning? This approach straddles the boundaries in the above classification scheme, and places teachers and curriculum developers in the position of adapting a technology to an unintended purpose [4, p. 7]:

> In many cases, applications software was originally developed for business rather than educational uses, and was aimed at adults, rather than children, as users... Because applications software is, almost by definition, not intentionally educational in nature, it is in many ways fundamentally different from other kinds of software used in the classroom.
>
> One critical difference between applications software and other educational software is that there are not specific learning objectives. That is, ... applications software is not *designed* to teach anything.
>
> This is not to say, however, that applications software cannot be used to teach a large set of problems, concepts, and skills.

Discussions of the design of the interface of applications software in relation to novice users are very common in the literature of computer science and industry (e.g., [5]). Such research, however, is most often concerned with the process of learning about the mechanics of the program itself, rather than the uses to which it can best be put in teaching and learning in content domains.

In educational applications, user interface design has received little attention, despite the fact that the interface is particularly important for educational software because "it must provide an entry to the content domain of the program rather than vice versa" [6, p. 93]. This issue is especially crucial when, as in the case of this study, no previous experience with the topic at hand (graphing and data analysis) is presumed. This concern goes much deeper than the nebulous concept most often represented by the buzz phrase, "user friendliness." The students in this study had already worked with the Macintosh™ operating system (renowned for its

favorable learning curve) for a minimum of five weeks when they first encountered graphing software. Nevertheless, many had several serious difficulties with the commercially-available software used in the early stages of the project.

In this study we investigate what special considerations should enter into the design of graphing software for *educational* purposes, rather than for use by scientists, engineers, or businesspeople (cf. [7] for the special case of graphs as representations of algebraic equations).

THEORETICAL BACKGROUND: EXPERIENCE, EXPLORATION, AND COMPUTER-BASED ACTIVITIES

Like anyone claiming to be concerned with "learning by experience," we should first return to our roots in Dewey's technical definition of education: "the reconstruction of experience which adds to the meaning of experience, and which increases ability to direct the course of future experience" [8, p. 76]. Dewey is known for his emphasis on the inherent (vs. instrumental) value of "meaningful" experiences in the present, but the typical caricature of his work ignores the value that he also placed on past "truly educative" experiences by virtue of their effect on the future [9]. We have noted elsewhere [10] that the essence of instructional design in this spirit lies in an effort to "facilitate *and* regulate the exploration of alternatives on the part of the learner" [11, p. 43] (italics added). As Dewey later wrote, in response to the rampant misinterpretation of his ideas [12, p. 75]:

> It is a mistake to suppose that the principle of the leading on of experience to something different is adequately satisfied simply by giving pupils some new experiences any more than it is by seeing to it that they have greater skill and ease in dealing with things with which they are already familiar.

Many of the participants in the past decade's debate about the role of microcomputers in education could profit from reminding themselves of the oft-ignored complexity of Dewey's and Bruner's thought. At one extreme are "visionaries" [1] such as Papert [13], who place tremendous faith in the ability of the child-computer dyad to achieve valuable learning from repeated, *self-regulated* experience using flexible, powerful software. This idea has been summarized as "the opportunity does the teaching by itself" [14, p. 13]. To be fair, this seems to be one of many instances in which Papert's view has itself suffered the indignities of oversimplification at the hands of its critics.

At the other extreme lie those, also hailed as visionaries during an age in which the most modest computing resources were prohibitively expensive, who see the educational potential of computers primarily in terms of more consistent and efficient methods of didactic instruction (e.g., [15]). Translated into a world of

relatively inexpensive and powerful microcomputers in classrooms, this view rejects the "romanticism" of experiential learning in favor of sophisticated, artificially intelligent tutoring systems that have the potential to fully replace a human teacher in well-defined knowledge domains such as biology, chemistry, or physics [16].

Regardless of one's predilection for designing explicitly educational software as either tutee or tutor, as outlined above, the development of a theoretical basis for the design and use of tool software for education remains problematic. Is complete flexibility in exploration of alternatives always a reasonable model? Is it possible for a truly multi-purpose tool to provide any useful feedback or guidance?

If students, not just teachers, are ever to regard computing power as a useful means to a worthwhile end, rather than as a surrogate teacher or as an object of study in itself, perhaps tool software should be designed to call as little attention to itself as possible. The ideal would be to replace human-computer communication with "human-problem-domain communication" [17], allowing students to forget that they are using a computer and concentrate on the subject-matter issues at hand. At a more basic philosophical level, this recalls Polanyi's vivid metaphor concerning the "internalization" of familiar tools: a hammer, for instance, gradually comes to be viewed as a natural extension of one's hand, and ceases to be thought of as a separate technological artifact at all [18]. Winograd and Flores, attributing this metaphor to the work of Heidegger, have specifically included it in their high-level theorizing about the appropriate aims of software and computer systems design [19].

This argues against an attempt to burden a tool program with so many *ad hoc*, prescriptive functions that it becomes more like a clumsy and thinly veiled tutorial [20, p. 6, 9]:

> What are we trying to get the computer to get the student to do, in designing a teaching program? We need some model of the optimal learning situation to emulate. The one-to-one tutorial is not appropriate for a medium where the teacher is necessarily absent. The focus is not going to be student-teacher, but student-subject.
>
> . . . the way the student experiences the domain being learned [should be] through direct manipulation of . . . the objects in the domain, [with] feedback about the behaviour of the system in terms of the results of the manipulations.
>
> . . . The advantage of . . . goal-oriented manipulation is that it . . . allows the student to "act like a scientist."

These arguments were originally advanced in the context of open-ended simulations of natural processes, many of which have been used in teaching physics (e.g., [21]). We believe that they can also fruitfully be applied to tool software in a case (such as that of a graphing utility) in which the "system" involved is a set of

general-purpose mathematical representations, rather than a mathematical model of the operation of natural laws. Any such program gives *intrinsic* feedback, that is, the substantive results of the student's computer commands in terms of the system's behavior (including both modified graphs and messages such as "that is not possible"). *Extrinsic* feedback, in contrast, is a nonspecific evaluative comment on the match between the students' input and some predefined goal state ("correct," "incorrect," or "close, but . . . "), which is more characteristic of a tutorial program [20]. One version of our original graphing software used in this study is strictly limited to intrinsic feedback—learning about graphs from the samples it provides relies heavily on self-regulated critical thinking on the part of students.

The available empirical evidence calls into question the notion that many students will learn important lessons from relatively unsupervised exploration of problems, whether or not they are aided by powerful and appropriate computer software. Research in this area has not been limited to the specific, emotionally-charged issue of evaluation of the educational use of LOGO programming (e.g., [22, 23]). Powerful, potentially helpful program features are rarely used by students, no matter how much they might help, if they are clearly presented by teachers as optional [24]. The educational value of time- and labor-saving software can depend heavily on the context in which it is used, including the level of student motivation: " . . . it seems that the avoidance of cognitive load in the absence of any real press to behave more elaborately leads to a failure to exercise higher-level skills" [14, p. 13]. Extensive studies of problem solving in abstract mathematics have shown that students also fail to recognize or internalize useful ideas or skills that were practiced or explicitly modeled but not explicitly identified or explained [25].

Certainly, software designed to be used by students to gain experience in graphical data analysis must be flexible enough to allow them to explore unknown territory, and therefore to make some mistakes. The hope is that these will ultimately contribute to learning. The implications of actions which are logically possible but inadvisable, reflecting the application of erroneous concepts, can then be confronted and evaluated.

Our early experience, however, clearly indicated that unrestricted freedom of choice from among a large array of powerful software commands can be disorienting. Taking a cue from many successful commercial programs, another version of our graphing software selectively disables ("grays out," in terms of the Macintosh™ interface) many program options in certain contexts. We characterize this as "preemptive" feedback—a tacit yet tyrannical form.

Students cannot reasonably be expected to induce the major principles of the appropriate use of graphs without some degree of appropriate feedback. Applications software that is too restrictive or prescriptive, however, is probably not conducive to thoughtful experience and, therefore, to meaningful learning. If an important goal of computer-based educational activities is to cultivate an

appropriate balance between the exercise of raw computing power and of human judgment (cf. [26]), how should software best be designed?

A theoretical perspective on tool software that we find most intriguing is Salomon's concept of "artificial intelligence in reverse" [27]. Rather than a computer modeling human intelligence, might human students come to emulate certain aspects of the computer's performance? In this view, the power embodied in the responses of a computer program serves a role analogous to that of Vygotsky's "more capable peer," in the spirit not of a tutor but more an assistant or partner, with whom a student can accomplish things that she or he could not handle alone [28].

The primary benefit of a computer tool is its "enabling function," allowing the user to test new possibilities and examine their consequences. But perhaps tool software can (although programs aimed at professionals rarely do) perform somewhat intelligent "guidance" and "modeling" functions by *subtly* raising questions, signaling possible errors, or providing externalized metacognitive guides [27]. One version of our graphing software allows great freedom in experimenting with different graphic forms for displaying data, but periodically gives verbal reminders and warnings ("coaching" notes) about the assumptions implicit in the action that the user has taken. It provides, in other words, not artificially intelligent tutoring but "heuristic guidance" in order to facilitate "semantically constrained exploration" [29].

Salomon also raises the issue of evaluating the success of students in computer-assisted activities [27]. Particularly well-designed software as an intellectual partner can be expected to upgrade performance during the partnership, but is there a useful "cognitive residue" that can eventually be used without the scaffolding of the external, intelligent tool? Building on Vygotskian terminology, there may be a zone of proximal *performance*, but not necessarily a zone of proximal development. Improved performance is necessary but not sufficient to demonstrate learning. In other words, besides the enabling function of the software, which cannot (and need not) be internalized by the students, do the guidance and modeling functions seem to stick with them? This concern is our reason for evaluating the various software interfaces by the criterion of their effect on written post-test scores, rather than on immediate success in answering the "research questions" addressed in the computer-assisted problem sessions.

DESIGN AND PROCEDURES

This study was part of a curriculum development project, undertaken jointly by University of Michigan researchers and teachers in the Detroit Public Schools (DPS) whose classes were involved. In total, eighteen teachers and over 800 students from six high schools participated.

As a group, students from these schools typically fall far below state and national means both in overall achievement and in exposure to science and

mathematics courses. Detailed background data on individual students were not available from the district for our research purposes. Most participating students were in the ninth and tenth grades, and fewer than 15 percent reported having taken bona fide algebra or laboratory science courses in the past.

The unit on graphical data analysis was part of the DPS Computer Applications Program (CAP) curriculum, and comprised seven fifty-minute class sessions. The computer lab classrooms were equipped with Apple Macintosh™ (512K, unenhanced) hardware. The application program used most recently, *CAPGraph*, was developed to incorporate several software design principles derived from our experiences with students in the first stage of this study, in which the commercial program *Cricket Graph*™ was used [30].

In the first three class sessions we used didactic, whole-class instruction aimed at familiarizing the students with reading and using graphs and with the capabilities of the software. Exposition of general principles was intertwined with numerous examples of the use and misuse of graphs in various data analysis contexts. Students were repeatedly given "hands-on" exercises in using the computer commands and interpreting their results.

Three sessions were then devoted to loosely guided practice in graphical data analysis in the context of problem sets. For each problem, students were given a data set (in the form of a spreadsheet file) and a "research question" which could be answered by creating and modifying an appropriate graph on the computer. Students most often worked in pairs at each computer station throughout the unit, an arrangement originating in necessity but which has several advantages both for teaching and learning and from the perspective of cognitive process research [30].

The final class session comprised two tests, a hands-on evaluation of facility with the mechanics of the software and a written test on graphing principles. The latter was a paper-and-pencil, free-response test, designed to correspond as closely as possible to the task environment familiar to the students from the computer problem-solving sessions. Questions asked on the test probe not only the students' basic competence in reading and drawing inferences from graphs, but their ability to make evaluative judgments about the design and modification of graphs appropriate to various kinds of problems.

This study was conducted in three distinct phases, corresponding to an evolving set of research questions and methodological approaches. First, in early trials with a small number of classes at two schools, students used a commercial program, *Cricket Graph*™, which is extremely popular in business and academia and known for its especially "friendly" interface. It eventually became apparent that this extremely powerful software had several serious flaws from the point of view of experiential learning by high school students.

Second, we programmed an original application, *CAPGraph*, designed specifically to alleviate the most troublesome problems, from the students' point of view, of the commercial software. This software was pilot-tested by students at three schools, whose critical comments and suggestions were actively encouraged and

usually heeded. During this phase, significant aspects of the program's interface were often changed, literally, overnight. This mode of operation empowered everyone involved, researchers, teachers, and students, rather than immediately adopting a more circumspect mode of research in which existing computer artifacts are implicitly taken as the universe of possibilities. This is a reflection of [31, p. 95]:

> . . . a "two-directional image": not only do computers affect people, but people affect computers . . . we affect computers when we study their use, reflect on what we see happening, and then act to change it in ways we prefer or see as necessary to get the effects we want. Such software engineering is fundamentally a dialectical process between humans and machines. We define the educational goals (either tacitly or explicitly) and the create the appropriate software. . . . Through experimentation, new goals and new ideas for learning activities emerge.

This process of progressive refinement characterized not only our instructional and software design process, but also our theoretical perspective. For a reflective participant observer, iterations of such "feedback loops" cannot help but also modify the set of values and assumptions that color the very process of observation and measurement ("feedforth" [32]). In this intermediate phase, hypotheses were generated for the quasi-experimental study that followed.

Third, we explicitly took stock of our educational goals, "froze" certain basic aspects of the design of the software, and created three systematic variations of the interface, which were then used by students in four schools. Entire classes were assigned on a random basis within each school to work with a particular interface. The differences among the three versions of the otherwise identical software were very well defined, avoiding the thorny issue of satisfactorily "equivalencing" qualitatively different experimental treatments [33], a common cause of confounding of variables in educational computing research [34].

Our three different versions of *CAPGraph* represent different combinations of flexibility and feedback (Table 1): first, a maximally flexible interface, permitting all logically possible command choices and showing their resulting effects on the graph without comment; second, the flexible interface with "coaching" feedback presented in conjunction with the graphical results when a questionable command is issued; and third, a relatively restrictive interface that disables several program commands in contexts in which they are logically possible but not advisable, according to widely-accepted principles of exemplary graphing practice (e.g., [35]).

The programmed functions generating the "coaching" messages are based only on the data structure of the current graph and a general model of appropriate graphing practices for each combination of data types. The modestly artificially intelligent performance of the software did not depend on any information specific

Table 1. Versions of *CAPGraph* Software, Characterized by Degree of Flexibility and Feedback

Interface Design	Flexibility	Feedback
"Open"	flexible	implicit
"Coaching"	flexible	explicit
"Restrictive"	less flexible	"preemptive"

to the "research questions" in the problem sets with which the students were working.

We then compared and contrasted the experiences of students using each style of interface, using anecdotal observations to help interpret the results from the written test of practical knowledge of graphing.

RESULTS

Phases 1 and 2: Experiences with Commercial Software; Development and Refinement of the Original Prototype Software

Before this study began, the only alternative available to Detroit teachers wishing to include computer-assisted graphing in their curriculum was to use the graphing facilities of *Microsoft® Works™*. While this integrated software tool includes a word processor and spreadsheet that are more than adequate for student purposes, the user interface to its graphing functions leaves most students (and many adults) overwhelmed and befuddled.

When the basic "new chart" menu command is selected, this program immediately confronts the user with a large number of simultaneous choices (Figure 1). This ability to specify all aspects of the appearance of a graph all at once, in a single "batch" process, is a boon to efficiency *if* the user is an expert professional who has already planned her or his graphic display in advance *and* had the foresight to note down the appropriate column and row numbers on the (totally obscured) spreadsheet that contains the data to be graphed. A student, in contrast, is very likely to spend nearly as much time figuring out which numbers and letters to type into which boxes as he or she would have spent plotting a graph by hand. In either case, little class time is likely to remain for the students to consider the important substantive issues involved in designing the graph display itself. This program can save some time and labor by using the computer for drafting graphic presentations, but is not conducive to thinking or learning about graphing and data analysis while doing so.

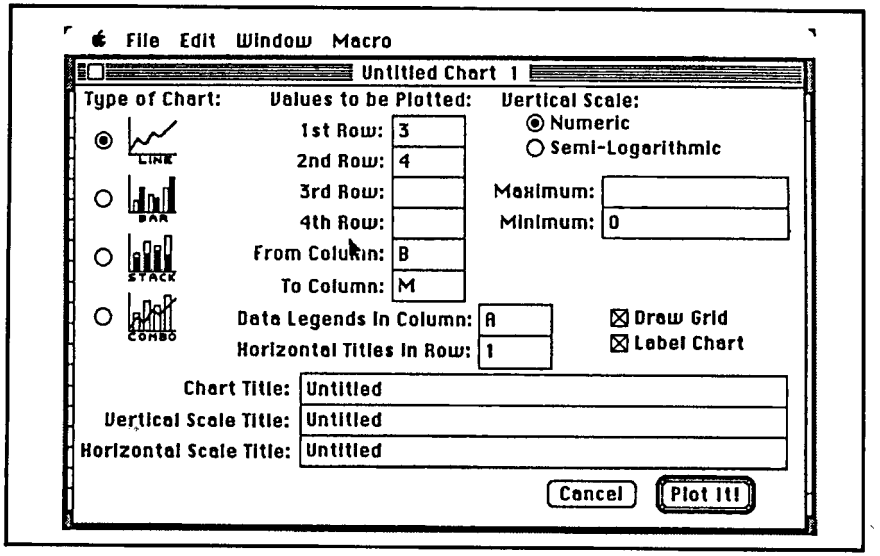

Figure 1. A relatively complicated, non-intuitive interface (*Microsoft Works*™): all important design choices are concentrated in one dialog and must be specified in terms of hidden spreadsheet coordinates.

Cricket Graph™, in contrast, is an outstanding example of highly sophisticated scientific/business software which, because of its relatively convenient and intuitively satisfying interface, is easily learnable by inexperienced students. The user can create a basic graph of any type by choosing from a menu that includes both names and graphic depictions of choices, then merely clicking on the names of the variables that should be plotted (Figure 2). The most basic and important options are presented first, and one at a time, in a way that allows a student user to concentrate on thinking about the substance of the decisions to be made about the graph, rather than on how to communicate their ideas to the computer (cf. [6, 17]). More subtle changes in the resulting graph can then be made individually through separate pull-down menu commands or, in some cases, by simply pointing and clicking the mouse to the part of the graph that needs work.

Almost all of the students involved in this study took immediately to the ease with which graphs could be created, modified, and embellished using this basically well-designed interface. We became increasingly excited about the potential of such computer power and intuitively appealing software to facilitate creative, productive "messing about" on the part of students learning about graphs and scientific data analysis [36]. Eventually, however, a list of serious problems with using *Cricket Graph*™ for teaching grew long enough that we were inspired to design and program a new application, *CAPGraph*, specifically for this project.

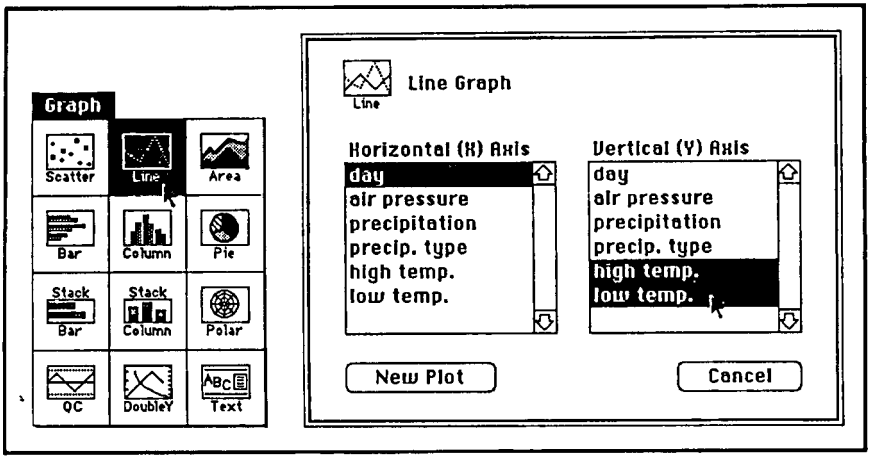

Figure 2. A simpler, more intuitively satisfying interface (*Cricket Graph*™): menu choices include graphics (left) and result in a dialog offering limited choices in terms of meaningful names (right).

Unfortunately, from the teacher's point of view, the combination of a "user friendly" operating system with a full-featured commercial program can rapidly become an attractive nuisance. A sizable proportion of the students were seriously distracted by the constantly available opportunity to play with the Macintosh™ fonts, styles, and sizes of text, or any of several superficially visually impressive, "professional-looking" cosmetic additions to their graphs, such as the illusion of depth ("chartjunk" [37]). These options were eliminated in our new prototype.

For a vast majority of the students, the more advanced mathematical features of the commercial program created a similar problem. Very few were ready to handle the interpretation of such options as logarithmic scales, nth-order polynomial curve fitting, inverse-square data transformations, or "smoothing" of data (Figure 3). Such facilities were not included in *CAPGraph*.

As the functions just mentioned would suggest, *Cricket Graph*™ is designed to be a general data-manipulation utility as well as a graphing program. Presumably because of this, the software is programmed to respond to some of the most powerful and commonly used commands in a way that students found highly counterintuitive. The "sort" command, for instance, has no effect on the current graph, which fills most of the screen, but rather on the data set itself, as reflected in a spreadsheet window that is typically barely visible beneath the graph window. If any reorganization of the data is to be reflected in a graph, an entirely new graph must be plotted, meaning that all of the design choices and modifications made up to that point in the work session are lost and must be reiterated. Students found this extremely frustrating. Many told us that they learned to avoid the sorting function