

THE RIGHT SIZE BYTE

Reflections of an Educational Software Designer

Judah L. Schwartz

Massachusetts Institute of Technology
&
Harvard Graduate School of Education

An Introductory Remark

In this essay, I set down some ideas I have developed over the last decade about the problems of designing educational software. These ideas have evolved out of my own experience designing educational software in a variety of subject areas and for a range of audiences. I make no claim here that these views represent deep truths about educational software design, and the reader should be aware that they do not represent the thoughts of the majority of educational software designers.

Choosing A Topic

The first issue an educational software designer must address is the choice of content for the software that he or she is about to design. This choice will often, although not always, imply a relatively specific audience for the software. For example, a piece of software on special relativity is likely to be used in a high school or college physics class but not in elementary school. A piece of software on color names may be appropriate for kindergarten and early elementary grades but is not likely to be used in the later years of high school.

Whoever the audience, we can think about choosing content in the following way:

- old content using old approaches
- new content using old approaches
- old content using new approaches
- new content using new approaches

For the most part, traditional Computer-Assisted Instruction (CAI) uses old approaches to old content. By and large, the instructional content of the majority of CAI materials is already part of traditional curricula. The pedagogical technique employed in most CAI materials has an authority, in this instance, the computer, posing a question to the student, waiting for an answer, assessing the quality of the answer, and choosing the next pedagogical interaction.

Some educators at another extreme, bedazzled by the apparent potential of the microcomputer to affect education, have raised a clarion cry for new curricular content using new approaches. No longer will it be necessary for youngsters to learn to spell, there will be spelling checkers. No longer will it be necessary for youngsters to learn to divide, there will be calculators. These enthusiasts similarly anticipate that the presence of memory resident thesauruses, atlases, and dictionaries all hiding just behind the next *hyper-something-or-other* will make room for totally new content in the curriculum. The claim is made that these tools will make it possible for children and teachers to bypass the “curriculum of the culture” and to generate for themselves whole new subject areas to replace those which have traditionally been taught and learned.

Without commenting on the intellectual depth and solidity of these claims (and the examples offered in support of the claims which are for the most part very uneven), the sparkle and shine of new content does little to hide the quite traditional pedagogy that many of the “new content” people use. They tend to confuse the use of new media with new pedagogical approaches. Thus many, like the CAI traditionalists, use the computer to ask questions and assess answers. Others who are equally enthusiastic about the promise of the new technology, but seemingly unwilling to put forward concrete examples of their visions, abdicate responsibility and abandon students to find their own ways in the electronic intellectual treasury they now have access to.

In and of itself, there is clearly nothing either wrong or silly about the call for new content. Much of what we teach is indeed outdated or foolish. Moreover, it is important to rethink continually the content of what we teach and to revise that content to reflect the realities of the world around us. There is, however, a pragmatic difficulty. New content, in any medium, is frightening to many school boards and to some teachers. The problem is more severe in the case of new content and a new medium, in this instance the computer.

Does this mean that the promise of the microcomputer will not be realized in education until we overcome the intellectual inertia of those who want to keep teaching forever what they were taught as children? I suggest that judiciously chosen traditional content, implemented in thoughtful and engaging new ways that both promote and scaffold new learning and teaching approaches, is the key to bringing the new technology, and in the end new content as well, into the lives of our students and teachers in a serious way. In this chapter, I will discuss the design of software that takes this approach.

Choosing this approach has the virtue of making change more acceptable and more workable in schools. It also has a longer term and potentially far more important entailment: dealing with old material in new ways will press teachers to rethink and reformulate their own understanding of what they are teaching. Ultimately, I believe they will be led to introduce new topics and materials with enthusiasm and freshness. The value to students, and to society as a whole, of teachers modeling this behavior, should not be underestimated.

What Responsibility Should the Software Have?

Having chosen a subject for the software, the designer must next consider the question of how much instructional responsibility the software should have. The extreme stances on this question are:

- a. the software runs the conversation with the user, making inferences where necessary about the intentions of the user, and
- b. the software simply responds to the user, displaying the consequences of the user's actions.

My own position on this question is unequivocally on the side of the second option. I take this position for several reasons. First, I think understanding a user's response well enough to make inferences about his or her intentions is the central problem of artificial intelligence research. In general, this is a very difficult problem which has never been satisfactorily solved. Each of the partial solutions that do exist represents an enormous investment of human and financial resource, is pedagogically flawed, and requires hardware environments that are not likely to be affordable by schools for some time.

Even if these difficulties did not exist, I would still be uncomfortable with the first option. Normally in human discourse, when A asks B a question, B presumes that A cares about the answer. No student is ever going to believe that the computer that poses a question cares about his or her answer. We have more than sufficient evidence of the alienation of our youngsters from school settings in which they are asked questions with known answers or asked to do tasks that have no "clients" for their products.

There is one circumstance under which I am willing to take a different position on this question. When and if a student gives "informed consent," I am willing to use a piece of software that runs the conversation. Here is an instance drawn from my own experience. I do not know how to touch type, and I feel no compulsion to learn. But if I did, I would be more than happy to use a normative, didactic, conversation-running piece of software to learn how to touch type. I would not, under those circumstances, resent in the least the fact that the software was telling me what to do, how often, and when.

How to Think About The Intellectual Scope of A Program

Just as it is possible to prepare written materials that address a specific topic, and thus can be "used up" in the sense that students master the content and are ready to move on, it is also possible for sharply targeted software to be "used up." It is not a criticism of a piece of software to say that it addresses a narrow range of curricular concerns.

It is also possible to write software that has very wide, domain-independent utility. Perhaps the extreme case of software of this type is the programming language. A programming language is a tool for making tools. There is no reasonable way it can be

“used up” in the sense mentioned above. Obviously, it is not a criticism of software to say that it is a tool-making tool.

Though I believe both extremes of software types to be useful and desirable, I believe there is reason to consider where on this spectrum of specificity to generality lies the possibility of making the greatest educational impact. Practical problems of classroom and computer lab management accompany a heavy reliance on software that can be “used up.” There is a continuing need to find the right piece of software for students to be using at any given moment as well as the logistical task of keeping track of many disks and manuals.

The pragmatic difficulties that attend the use of the open-ended end of the spectrum of software type are different in kind but no less difficult to resolve. Open-ended programming languages are marvelous tools in the hands of teachers who have both mastered the language and know how to use it interestingly and imaginatively to challenge students to do projects in subject areas that schools will recognize as legitimate. Unfortunately, such people are rare.

I believe the greatest potential educational impact of microcomputer software lies neither in narrowly targeted software that can be “used up” nor in general purpose tools. I believe that the promise of educational software and the most exciting creative challenges lie in the design and programming of a kind of object intermediate to these two. In the kind of software environment I have in mind, the intellectual domain is bounded by the functions the software puts at the user's disposal, but within that domain users are unconstrained in what they may do. Software environments of this type do have a subject matter. They are not general purpose tools. They are certainly not tool-making tools. On the other hand, like tools, they do not question, judge, or assess their users' intentions. They simply display the consequences of users' actions and allow them to raise their own questions and assess their own actions.

Guided Inquiry - An Appropriate Pedagogy

I have explored the characterization of software along two dimensions. On the first dimension, that of constraint to open-endedness, I have given my reasons for preferring the open-ended end of the spectrum. On the specific to general dimension, I have tried to argue the importance of an intermediate position between narrowly targeted software on the one hand and programming languages on the other.

Having thus located myself in a kind of instructional space, I turn to the question of what kind of pedagogical style is most appropriate to the stance I have adopted.

I believe that the kind of microcomputer use I favor dictates a sharing of learning and teaching roles among teachers, fellow students, and the individual student. The computer, like the encyclopedia, the atlas, the dictionary, and the thesaurus, can serve them all, either collectively or individually, when needed.

The teacher's responsibility is to challenge, to lead, and, above all, to *educate*, to draw out of students their understandings and misunderstandings, their insights and their inventions. Our rhetoric about teaching, for the most part, honors this role. It gives substantially less prominence, however, to the teacher's role as continuing learner of the subject that he or she teaches. For a teacher to be a continuing learner of the subject, he or she must be a careful listener to the words of students. Students often have fresh and interesting ways of both understanding and misunderstanding a subject. Careful attention to what students say can help teachers continue to broaden their own grasp of the subject and the ways students come to learn it.

Listening to students carefully, while necessary, is not sufficient. Teachers must actively seek to extend their own knowledge and understanding of their subject. Often they do this in professional groups and in courses. I believe that software of the type I have been discussing offers yet another avenue for the professional intellectual growth of teachers. It is a particularly desirable avenue of growth because, in contrast to coursework and professional associations, it involves the very same medium their students use. The opportunity for students to see their own teachers acquiring and making new knowledge is certain to make a both a lasting and desirable impact on them.

Students also have roles and responsibilities. They must learn the “curriculum of the culture” in some depth. By this I mean both their own culture as well as enough of others' cultures so that they come to respect and cherish the diversities of both groups and individuals. They must also learn that what there is to learn is not static, but rather is a live and growing body of knowledge to which they can, and indeed are obliged to, contribute. Traditional school practices do not make the learning of this latter lesson easy. Such learning cannot happen without teachers who encourage and challenge. Given teachers who do so, I believe it is easier when students have access to curricular environments in which they can explore their nascent inventions and hone them to the point where they are willing to share them with others. These inventions will sometimes extend the previously known. At other times they will challenge previously held beliefs.

In summary, teachers must guide, but they must also inquire. Groups of students must inquire and help to guide one another. Clearly, in order for this to happen, teachers and students must come to formulate their roles and responsibilities in these ways. They will need curricular environments that permit these activities to happen easily and well.

Before closing this section, I want to add a comment about motivation. I believe it is important for the software designer to design software in such a way as to make the intellectual content of the software appealing and engaging. If it is necessary to have a “cover story” that drives users through the software by a series of external motivating “gimmicks”, then, in my view, the design is not yet as good as it can be.

The Problem of Simplicity and Complexity

Having selected a topic, a position on the constraint-open-endedness dimension, a position on the specific-general dimension, and a pedagogic style, I now turn to the

problem of designing a piece of software. One way to formulate this problem is to ask the following two questions;

What actions do you want users be able to carry out in this software environment?, and

What elementary tools do you wish to put at the disposal of the users to make it possible to carry out these actions?

The first of these questions is really about the operations of greatest complexity that the designer is willing to allow the user to be able to carry out in the software environment. The second question deals with the other end of the problem, i.e., how simple should the simplest possible operations in the environment be? Perhaps the best way for me to answer these questions is to draw on the example of THE GEOMETRIC superSUPPOSER, a software environment designed by Michal Yerushalmy and me.

This is not the place to describe this software environment in detail. For our present purposes, it is sufficient to say that the superSUPPOSER allows the user to make a series of geometric constructions on a shape such as a triangle, rhombus, a set of intersecting circles, etc. The software tracks the user's construction as well as any measurements that are made and records them. The construction and measurements can then be repeated effortlessly as many times as desired on new shapes of the same sort. The pedagogic position of the superSUPPOSER is consistent with the stance I outlined earlier. I believe that in the long run, we must help our youngsters learn to pose, as well as solve, problems. In my view the best way to do this is to allow them to practice posing problems in an environment in which it is inviting to do so, and having done so, to explore potential solutions. Thus, the superSUPPOSER does not pose problems for the user.

Let us turn now to a more detailed look at some of the design issues the superSUPPOSER raised for Michal and me. Two facilities characterize the superSUPPOSER. The first of these is a construction facility, and the second is a facility to operate on and manage the constructions one has made.

The construction facility of the superSUPPOSER contains such primitive operations as the ability to draw angle bisectors, perpendicular and parallel lines and circles that circumscribe and inscribe triangles, as well as the ability to label intersections, subdivide segments into N equal length segments, reflect both points and lines in lines, and place random points inside, outside, or on geometric shapes.

The facility to operate on and manage constructions includes the ability to measure angles, lengths, distances, and areas, as well as the ability to rescale the construction, to clear the construction, and most important of all, to repeat the construction on a new primitive shape, i.e., a new triangle, quadrilateral, and so on.

The design of the construction facility was not intuitively obvious to us. Indeed, it took a fair amount of time for us to understand two important issues about it. These were the problem of making the primitive operations of the software environment too simple

and the problem of making them too complex. In addition we had to come to realize that we were building a pedagogical tool for learning, teaching and making plane geometry, not a Computer-Aided Design (CAD) system.

Let me expand on these problems a bit. The mathematically trained person is likely to fall prey to a temptation to conceive of the primitive construction operations of a computer environment in geometry as the ability to make straight lines and arcs of circles. After all, is it not possible to derive all of Euclidean geometry from these simple and elegant beginnings? It is, of course, but at a price. The price is the tediousness of the task. If a user of the software must make an elaborate series of straight line and circular arc constructions in order to draw a median or an angle bisector, that user is unlikely to explore playfully the properties of geometric constructions that are rife with medians and angle bisectors.

The reader will no doubt object to this point, arguing that in a computer environment it should be possible to capture a series of commands and define them as a 'macro', that is, to create and name a composite command which consists of the sequence of commands that have been captured. In fact, is this not what the superSUPPOSER itself does? If it were possible to construct 'macros', then the user would not have to undergo a painful repetition of intermediate constructions to arrive at a perpendicular bisector or a median.

This point is logically correct, but pedagogically it is somewhat less compelling. Building and naming a procedure as a way of extending a language is a subtle and delicate idea. Although it is a very important idea and one that is certainly worth learning, it seems not to be an easy idea for many people to grasp. One needs only look at the many widely used spreadsheet and word processing programs in which powerful macro facilities are present and unused. We felt that it would be unwise to confound the problem of learning geometry with another set of important ideas about formulating and naming procedures. To be sure, the superSUPPOSER contains the ability to name and store procedures. But, the primitive objects in the superSUPPOSER are not straight lines and arcs of circles. Rather, the superSUPPOSER contains as primitive objects a rich collection of geometrically interesting structures such as orthogonal circles, obtuse isosceles triangles, quadrilaterals inscribable in circles, regular hexagons, etc.

Having argued against making the primitive operations of the software environment too simple, let me now discuss the problem of making them too complex. It seems to me there are several good reasons to avoid too great a degree of complexity in the primitive operations available to the user in a software environment. The first of these reasons is that complexity of primitive operations conflicts directly with ease of use. The more "bells and whistles" that are present in a piece of software, the less obvious its operation is to the naive user and the longer it takes to learn to use the program to accomplish useful work. It is a source of some pride to Michal and me that the superSUPPOSER seems to require no computer background or experience of the user. They do, of course, require some geometrical knowledge of the user, but that, we believe, is as it should be.

A second reason for not making the primitive operations of the environment too complex is pedagogical: the software should provide an environment in which the user can formulate an extended line of argument. If the primitive operations provided the user by the software allow for complex ideas to be instantiated instantly with no hint of their internal structure and composition, it is likely that intellectual opportunities will be lost.

Yet another reason for not making the primitive operations of a software environment too complex is to avoid a trap that Michal and I fell into early on. We managed to extricate ourselves from the trap before we were embarrassed into doing so by a community of users, but there is little guarantee that the software designer will always be so fortunate. In an early version of one of the GEOMETRIC SUPPOSERSs, the predecessor environments to the GEOMETRIC superSUPPOSER, we included a primitive that allowed the user to subdivide an angle into N angles of equal measure. The subdivision of an angle into an arbitrary number of equal angles is not a construction that can be made using the primitive Euclidean construction tools, i.e. straightedge and compass. We had lost sight of the primacy of the Euclidean geometry and were seduced by the capabilities of the software environment. If we had left such a primitive in place, we would have crippled the ability of the software to serve as an environment for the making and exploring of conjectures about Euclidean geometry.

Before leaving this discussion of the merits and demerits of simplicity and complexity, it is worth adding a comment about the distinction between tools for doing things and tools for learning how to do things. It is not difficult, for example, to imagine a Computer-Aided Design (CAD) system that would incorporate the features of the GEOMETRIC superSUPPOSER among many other capabilities. Would such a system be a more appropriate tool for the geometry classroom? I believe not. CAD systems, no matter how elegant, enable users to manipulate spatial constructs in order to design specific objects. They address fundamentally synthetic ends. The implicit agenda of CAD systems is to provide a tool to do things with, and they are not fashioned with an eye to inducing users to reflect on their actions. The implicit agenda of the superSUPPOSER - and one that it seems to address well is to help users to learn how to do things. It tries to seduce users into addressing analytic and reflective ends: wondering, conjecturing, and finally proving things about the general categories to which the spatial objects manipulated in the program belong.

Jumping Into An Interesting Middle

Our attempt to avoid the overly simple as well as the overly complex is an illustration of a broad pedagogic approach that I think of as “jumping into an interesting middle”. What I mean by this is finding a way to engage learners by letting them interact with and manipulate aspects of a subject that are complex enough to be interesting and simultaneously simple enough to be understood. Thus, in the case of the GEOMETRIC superSUPPOSER environment for Euclidean plane geometry, we chose, for example, to make the median of a triangle a primitive tool for the user. The median of a triangle is a moderately complex object in its own right. In terms of the logical development of the subject of plane geometry, it needs to be constructed from straight lines and arcs of circles, the constructions that form the logical foundation of the subject. If however, students using the software were obliged to construct each median they were interested in using in a painstaking fashion from arcs and straight lines, it is highly unlikely that they would have arrived at the rich collection of geometric theorems, both already known and totally new, that they have.

The pedagogic point here is larger than the geometric context from which it emerges. It seems to me that in order to engage a student in a subject, one must show the student quite early on something of the true nature of the subject and why it is that human beings are willing to expend effort in it. If one is introduced to a subject at a level of complexity that allows real problems to be wrestled with, then the point of studying the subject is clear. Furthermore, if it becomes clear to the student why the subject is worth studying, then it is likely that the student will be willing to expend the effort to understand its intellectual foundations. The level of complexity must be such that students can work their way back to the foundations without too much difficulty.

Satisfying the constraint of simultaneously not being too simple and not being too complex is a central challenge in the design of all curricular materials. The need to do this in the design of educational software should not come as a surprise.

Conclusions

Other much more detailed design questions are also worth discussing, but I have limited myself in this chapter to those that directly engage pedagogical questions. In so doing I have tried to focus on the problem of selecting a subject matter topic, recognizing one's pedagogic ideology, implementing it in the software medium, and finally steering a course between the problems that inhere in both simplicity and complexity of design.

It is likely that others who design educational software will take issue with the views expressed here. I can think of little else as salutary for our emerging field as for them to express their views publicly so the debate among us can proceed in a way that benefits from the views of students and teachers.