OPEN TOOLSETS: NEW ENDS AND NEW MEANS IN LEARNING MATHEMATICS AND SCIENCE WITH COMPUTERS*

Andrea A. diSessa
Graduate School of Education
University of California
Berkeley, CA 94720
USA
disessa@soe.berkeley.edu
http://soe.berkeley.edu/boxer.html

Abstract: *Open toolsets are a new genre of software that involves a greater number of smaller units than conventional educational "applications." The units are intended to be highly modifiable, extendable, and combinable with each other. Primarily with examples created in the computational medium, Boxer, I illustrate open toolsets. I suggest how they can be constructed and what learning properties they may have. In particular, I argue that open toolsets may allow very different development communities, involving teachers in essential ways. Finally, I outline areas of research that need to be done to support the best development and use of open toolsets.*

* This work was funded, in part, by the National Science Foundation in grant number RED-9553902. The opinions expressed are those of the author, and not necessarily those of the Foundation.

Reprinted from: diSessa, A. A. (1997). Open toolsets: New ends and new means in learning mathematics and science with computers. In E. Pehkonen (Ed.), *Proceedings of the 21st Conference of the International Group for the Psychology of Mathematics Education, Vol. 1*. Lahti, Finland, 47-62.

Introduction

In this paper, I wish to introduce, explain and advocate a new genre of educational software. My interests are mainly in mathematics and science education, but I am quite sure what I say here will travel well into other disciplines. After introducing the notion of "open toolsets" and the standards by which they should be judged, I will spend a significant amount of time on some technical issues. This is because technology sometimes really makes a difference in what we can accomplish educationally, and it happens that this new genre of software is only just now becoming technically feasible.

Open toolsets are, at first blush, just what the name implies. They are an open collection of tool-styled software units that are aimed toward learning in some particular subdomain, like constructions in geometry, system dynamics, particular pieces of ecology or evolutionary theory. "Open" is meant in several senses. First,

each tool in the set has the usual properties of a tool. That is, it helps accomplish tasks and is pedagogical to the extent that those tasks are educationally relevant. Tools, in general, trade off explicit representation of educational goals, which may be more evident in other, more didactic software, for a sort of authenticity in which software is instrumental to the goals of its users.

My motivations for developing tool-styled software are both aesthetic and instrumental. Aesthetically, I am committed to maintaining a sense of personal agency and responsibility in the users of educational software, teachers and students alike. This aesthetic goal also has instrumental overtones. If we want responsible, self-motivated learners and thinkers, we need to set up learning environments where those orientations are cultivated. But, I also persist in reminding that the educational experiences of our students are not just steps toward practical ends—for example, to get and keep good jobs—but they are also an expression of the values of our civilization. We cannot escape deciding how we want our offspring to experience life in school, beyond any practical measures of utility. I want them to feel directly empowered.

The more direct educationally instrumental value of tools, for me, follows from a basic orientation toward knowledge—that it is cultivated best in the service of goals understood by learners. I won't explain this presumption in detail here, except to note two things. Working within learners' goal systems serves "motivational" ends, keeping them engaged. But this also has epistemological ends. Goals help refine knowledge by providing an understood "measuring stick" to determine if current knowledge is powerful enough. Understood goals have high epistemological leverage.

So far what I have said needs only the "tools" part of the open toolsets moniker. The "sets" adds an additional dimension. These days, I believe software generally comes in too few and too big chunks. Flexibility is a key issue. The primary way to achieve flexibility in a big software system is by adding an endless stream of features. I am not the first person to note the relentless commercial pursuit of features and the downside for consumers. Mike Eisenberg (1995), for example, writes eloquently about this issue. Elegance, comprehensibility and mastery generally just fade away when software systems become too complex. Although a full-time user may appreciate a huge range of specialized features, when software is used educationally, we frequently pass from one piece to another as our specialized needs for learning evolve. So, in general, students and teachers just don't have time or need for exotic features hidden in a maze of options, subtools, preferences, and the like. The route advocated here is many simpler, easier to understand tools, compared with fewer, more complex ones.

There is a place for complex, broadly useful tools. I have been a long-term advocate of a very rich set of generic capabilities, like text and hypertext processing, graphical, network and programming facilities as the basis of students' experience with computers. This is the very concept of "computational medium" that has motivated

my own work with computers in education for a decade and a half (diSessa, 1995). But the concept of a computational medium suffers the same "problem" as generic literacy. Knowing how to read is only useful if there is a literature about the subjects you want to know something about. (And, incidentally, learning to read is best accomplished in the context of reading about something you care about.) So, open toolsets is a candidate for the content "literature" that will be the basis of learning particular things with computational media. These toolsets will interpolate between the best general-purpose environment we can imagine, and the scores or hundreds of learning micro-contexts we must create for students to become mathematically or scientifically literate.

We've gotten as far as "tools" and "toolsets." But why the apparent redundancy of "*open* toolsets," if the very meaning of tool connotes a degree of openness? The answer is that I want to emphasize toolsets that will never be finished and complete, but will always be open to changing old tools and adding new ones. Changing old tools is the easier of these desiderata to grasp. Tools like hammers and saws are flexible in one way—they don't "care" what they are used for. But they are not in themselves malleable and changeable. With electronic tools, if we are clever, we need never be limited in this way. Although, again, this is not a trivial cleverness. Almost all current software tools are much less changeable than I believe optimal for open toolsets.

The second part, adding new tools, is simple enough in principle. But there is still another level of openness. The key to this other level is that open toolsets work best with an important degree of synergy among components in the set. Each tool needs to be open to working with an unlimited range of new tools that might be added to the set. Merely having the toolset running on the same machine is far from sufficient. Having a toolset with components that can work flexibly together is critical, but still insufficient. Instead, the ultimate principle of synergy is that tools need to be interconnectible and combinable in ways that are not anticipated by the initial toolset builders. Indeed, they need to be combinable with other tools that may be added to the set in the future.

I would like to begin to articulate the criteria by which the success of open toolsets may be measured in more definite terms. To be sure, such software must support good learning by students in classrooms. As I said before, this learning will be critically dependent on the activities that we foster among students and teachers. There is quite a lot to think about with regard to this, and at least as much to design. But my emphasis here will be in a different direction. The idea of open toolsets is motivated in substantial degree by issues of the appropriation of technology deeply into existing cultures, and the co-development of technologies and new cultures.

I like to think about technology fitting and developing "social niches." These are the broad, repeatable patterns of production and consumption defined by the multiple social constraints of value, capability and interest that impinge on genres of technology and their use. The simple idea is that any software type, open toolsets in

particular, must suit many people's interests and ways of working. These pose strong constraints on what types of software exist and can come to exist.

Let me illustrate social niches by talking about what I propose for a new mode of software development for open toolsets. Currently almost all software development is done by teams of experts. Teachers and students, if they are involved at all, are testers and commentators. The paradigm I have most hope for with open toolsets is that these will be developed in larger groups, which include teachers and students doing important work in the design. Over an extended period of time, tools are tested and modified, activities are designed and shared, as a community with solid ties to the educational practice converges on a toolset that is both powerful and flexible enough to serve a wide range of local styles and interests. There may well be software developers (programmers) in the community, but far more time will go into building a joint culture around the toolset than into "coding" per se. We need to reverse the emphasis on code and interface and put it more appropriately on activities and cultures.

Is this a viable image? It certainly has attractive properties. But there are important uncertainties. First, how do people make money out of this scheme? The first law of capitalism is that, unless someone expects to make a lot of money, the whole program is unlikely to succeed. There are some scenarios that lessen this threshold. For example, if open toolsets prove extremely valuable in some experimental context, government in some form may decide it is in the public interest to facilitate their development.

A second critical issue is the expertise of teachers. Are there enough creative and thoughtful teachers for that subculture to take a substantial role in "software development," even if it is a very different form of development and a different form of software at issue? I am more confident than most people that there is a tappable, or at least developable subculture of teachers who can lend an important realism and "bottom up" sensibility to software development by contributing to open toolset creation. Even if this works within development communities, will the miniature communities that build toolsets be representative enough that other teachers, not in the development community, can easily join in? These and a host of other complex and uncertain issues about the viability of this paradigm are basically asking, "can we foster an effective social niche for open toolsets?"

Without proposing to answer such questions—we just may not know the answers without trying—let me review the rationale for open toolsets. First, an open toolset is a flexible collection of smaller, less "expensive" to build parts. We want to preserve initiative and ownership for students and teachers by designing a flexible base for learning and instruction, a base that grows and can continue to grow organically without ever imposing straightjacket-like constraints. Students will do "work" they understand and with which they can personally identify using these tools. Teachers, also, will be served by the toolset. They can exercise their personal style, teaching sensibilities and sense for their own students and local context in

designing and modifying activities with the tools, and even modifying the tools themselves (and combinations of tools) involved in activities. These properties can help with the critical task of teachers' appropriating technology into their professional practice. Open tools "meet teachers half way" in giving them more control over the technology. This extends further the fact that open toolsets may be developed in substantially different ways than current software; the school community may have a much more substantial *and sustained* role than at present.

No new idea is completely new. Open toolsets are an extension of many current ideas about learning with technology. They are in the same spirit as "constructionist" and microworld approaches to learning, popularized, in particular, by Papert and the MIT Media Lab (Epistemology & Learning Group, 1991). I also already mentioned Eisenberg, at the University of Colorado, in connection with his attempt to combat complex but still inflexible applications via programmability. Another project that is working toward many of the open toolset goals is the SimCalc project (Roschelle & Kaput, 1996).

Technology

Technically, how do we create these open toolsets? One obvious possibility is that each tool is a separate application. This is certainly compatible with the general framework of current computational systems. But it poses a range of limitations and problems that have become the subject of a lot of discussion and attempted "fixes." First of all, applications tend to be large and complex. Because the operating systems currently available supply only a limited range of services to users, applications must build from scratch, and they frequently duplicate each other's basic services. A good example is text editing. Almost every application does or can make use of text processing, but anything beyond the most basic services are built anew for each application. This is not only technologically inefficient, but leads to difficulties in learning—each implementation of a service will have idiosyncracies that users must learn in moving from application to application.

Other limitations of using separate applications are weaknesses in sharing data and in interaction. Each application typically has its own data types. You are lucky to be able to cut and paste text (and usually you will lose formatting) and simple pictures, much less a complete interactive object. Why shouldn't a business document contain an active spreadsheet? Or why shouldn't a student's science report contain both the data she used and the analysis tools, so the teacher can check the analyses or run a different one to show, for example, how a different conclusion is also supported? In terms of interaction, it is currently very difficult to connect different tools-as-applications. Suppose you want to connect a nice simulation, say of an ecological system, with a statistical analysis tool and a graphing tool. This is sometimes possible these days, but as often it is awkward, and too often it is just plain impossible.

There are three movements in contemporary computing that are related to each other and to the problems described above. First, "scripting" is really just the ability

to control an application (or a component—see below) with a programming language. This means applications can at least use each other's resources. You could, for example, have a script that sent some data to an analysis tool, and returned a graph-as-picture to your "home" environment (say, a text editor). You can essentially create a new application by gluing together resources provided by multiple existing applications. Clearly, this is in the spirit of open toolsets, and it is an important step toward realizing their promise.

"Component computing" is an attempt to allow multiple, small "applications" to work together efficiently. Microsoft established one standard, OLE (object linking and embedding) with a follow-up called ActiveX. Apple and IBM pursued another standard (OpenDoc, now apparently defunct). A newer possibility might be most familiar to you. The programming language Java, which is most visible in network applications, has a standard that allows multiple little "applets" on the same page of a net browser. Each job you want to do might require a different collection of several small applets, rather than several big applications. "Document-centered computing" is a slightly different way to describe component computing. The idea is you should always work on the basis of the document you want to create. The resources you need for the different parts of the document (text processor, picture editor, analysis tool, spreadsheet...) should just come with the document. You should not need to run off to load a big, separate application just to work on a piece of your document.

In this paper, I will use a rather different technological basis for making open toolsets feasible. I will use Boxer, the system we have been designing, implementing and testing for quite some time at the University of California, Berkeley (diSessa, Abelson, & Ploger, 1991). Boxer has some advantages over the movements described above. In particular, it makes full programmability by a simple programming language (simpler than Logo for beginners, we believe) the heart of its capabilities. But it also has disadvantages. The most obvious disadvantage is it is not "industry standard," and therefore it is not ubiquitous. I won't continue the discussion of advantages and disadvantages of Boxer here, even though these are interesting and important. That is because the point of this paper is to paint a picture of what open toolsets might be like, and what they would achieve educationally.

Before we can begin serious assessment of the educational possibilities of open toolsets, I want to give a more detailed sense of what such tools might be like. I will do this by describing a number of examples, implemented in Boxer. For more technically oriented readers, I note that the capabilities that make these examples possible come in three levels. First, any component system allows a level of combinability. Second, some possibilities described below also require programmability. Finally, a few (such as graphical/computational objects; see Vectors, below) require a computational medium, like Boxer, where every object is computational.

A Visible Calculator

Let me begin with a trivial example—a simple calculator. I do this mainly because

everyone knows what calculators are and how to work them. So, what is new in this example is exactly what makes this an open tool, unlike the calculator you have in your desk drawer, or more appropriately, the one on your computer. You should be able to see clearly just exactly what "opening" this sort of tool can mean.

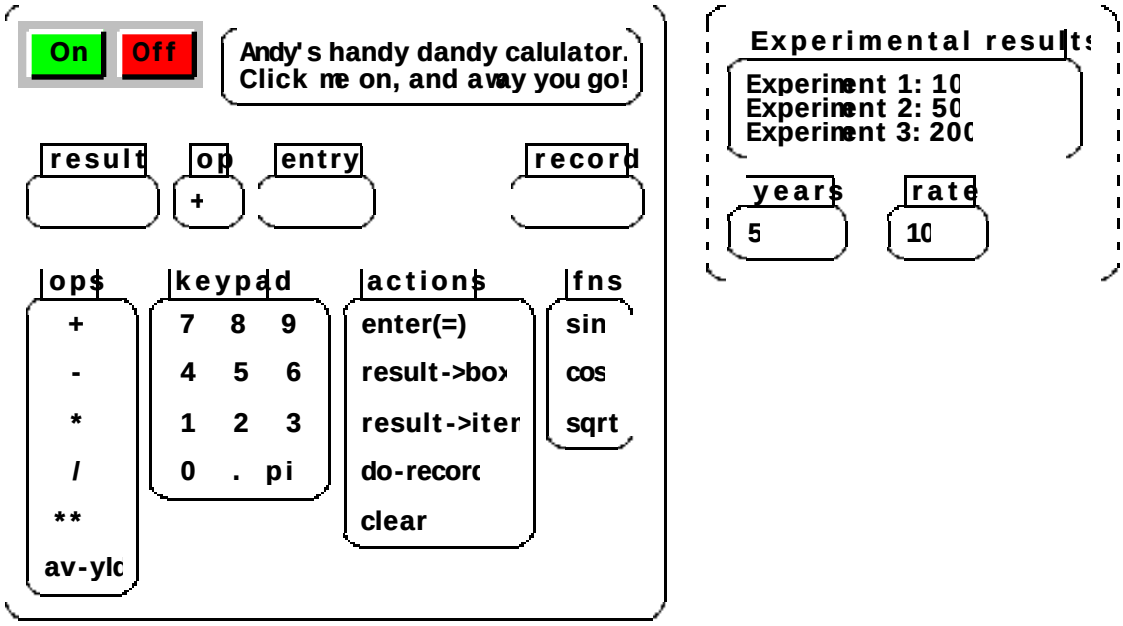

Figure 1. A "visible calculator" (left) and some of the surrounding environment (right).

Figure 1 shows a Boxer "open calculator" (the box on the left) and a few pieces of the surrounding environment (on the right). This is pretty much generic Boxer. Boxer is an environment constructed out of boxes that contain text and pictures. Text and pictures, themselves, can contain more boxes, and hence more text and pictures, in meaningful box-chunks. Here, the calculator is a box. It contains, for example, a keypad, which also is a box, and on and off buttons, which are boxes that happen to have a graphical "boxtop" presentation (icon). You can "flip" the graphics to see the *real* box underneath. Each of these, the calculator itself and its parts, might be considered components in a component computing model. In particular, in Boxer you can cut, copy and paste any box (and text and pictures, of course, also). This means you can move your calculator into any workspace or document you like within Boxer. You can also paste a copy of the calculator in a document to leave as a permanent part if it, for example, to allow users of the document to do their own calculations with the data provided. Why shouldn't a tax form come with a calculator, or a written assignment for a student come with the tools s/he needs to do the assignment? Why shouldn't you be trying out the calculator in Figure 1, rather than just looking at it?

• *Open tools can be cut and pasted into any document or work environment, and left there for any future work.*

Boxer adds a new and important sense to "open" in the tools it allows. The boxes that serve as display devices, **result**, **op** (operations) and **entry**, show you the actual working state of the calculator. They are, in fact, simply variables in Boxer, which are nothing more or less than named boxes. You are literally looking at a part of the working mechanism of the calculator, not just a display. In this case, I (as designer) decided it might be more useful to see the two numbers you are going to operate on and the operation that is going to be performed, in contrast to just the standard single number display. This form of openness, showing the works of the tool, is pretty simple in its implications here, but not completely trivial. For example, you can see what is going to happen when you push the **enter(=)** key; you don't have to guess. Did you accidentally push **/** rather than **+**? What happens if you press **123 + 54**, and then decide you meant **123 - 54**? Do you have to start over, or can you just press **-**, then **enter(=)**? (It happens that you can just do the latter, which is nice.) What is the difference between pressing **enter(=)** the key once and twice? Since you can see the "internal" state of the calculator, you know what happened and what is going to happen. When tools get more complicated than a calculator, being able to easily show some of the internal state and workings is even more of an advantage, as later examples will better illustrate.

• *Open tools can be open in the sense of "transparent." They can show users what is going on.*

The calculator has an important property, trivial modifiability. The keypad is nothing more than some text typed in a box. So, if you happen to like a different arrangement of keys, just do it! In fact, I added the **pi** to the calculator since I frequently do scientific calculations that need it. To add it, I positioned my typing cursor and typed "p" i". Not a difficult task. And if you don't want that "key" in the keypad, just select and cut.

• *Some aspects of open tools may be trivially modifiable. Accomplishing a change in the tool may be almost as easy as thinking of it.*

Of course, modifying **ops** (operations) and **fns** (functions) may be just as simple. Sin (sine), cos (cosine) happen to be built into Boxer, so inserting them was as easy as typing three characters. Similarly, a log function or tan function come and go from the tool with a few keystrokes. Modifiability and adaptability are hallmarks of open tools.

Now, trivial modifiability just may not do the job. Not every function or special number is built into Boxer. Suppose you need a statistical function, like standard deviation. Suppose you need a financial function like average-yield, in order to compute the average yield of an investment that appreciated 50% in 5 years. (Note in Figure 1 that **av-yield** actually appears in this calculator!). The calculator was written so that any function that you can write in the Boxer programming language can be added to the calculator by cutting and pasting the function into the calculator works, and, again, adding its name to the appropriate keypad. In this case, **av-yield** has been pasted into a hidden part of the calculator, its closet. Every box has a

closet, so this is something most Boxer users, especially tool users, know how to find and use.

• *Boxer open tools are always open to "extensive modification" as well as trivial modification. You can program extensions or changes in how the tool works.*

If a tool is well-written and you understand Boxer programming, "extensive modifiability" may be nearly as simple as trivial modifiability. This means a lot for Boxer literate folks. If means they can't be straighjacketed by any supplied tool. But it means just as much in a community that includes Boxer novices. If a novice makes a suggestion, frequently a more expert colleague can just make the change on the spot.

We haven't even gotten to the more interesting aspects of this calculator.

• *Open tools interact well with their environment. They are an integral part of a flexible workspace.*

It happens that you can click on any number outside of the keypad to enter something to compute with. This may seem trivial. It may just save a few keystrokes. But fluidity is not to be underestimated. If you take half the drudgery out of a job, a lot more learning can occur. And, this same principle, excellent integration of a tool with its environment, has deeper and more important forms than saving keystrokes. Examples to come will illustrate this.

Getting data into the calculator suggests that one might want easy ways of putting data out. The actions **`result->item`** and **`result->box`** allow you, respectively, to replace any number outside your calculator with the result, or to append the result to a box. So, the calculator has as many external memory registers as you like, and those can be within explanatory text (**`Experiment 1`**, ... in Figure 1), if you so choose.

• *Boxer Open tools are computational in several senses. Sense (a): Since Boxer has (is) a programming language, it is exceedingly easy to capture, reflect on, and change a process.*

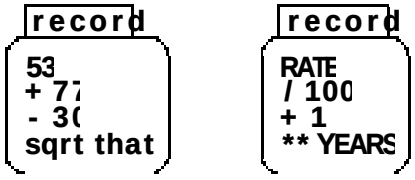

Figure 2. A record of a simple calculation (left), and one using variables (right).

The record box in the calculator automatically saves a record of the calculation you perform. This turns out to be terribly easy to program. Every operation on the calculator, a mouse-click here or there, winds up executing a Boxer command. The record is just a slightly modified listing of these commands. Many Boxer tools and microworlds benefit from such recording capability, which might be called "macro

construction" in some systems. Not only can you review your work, checking for mistakes, but you can edit the record and re-run it (action **do-record**). Figure 2 shows two such records. The first is quite ordinary: adding up a series of numbers and taking the square root. (**That** is a convenient notation for the result of the previous calculation. It is handy to use with prefix operations.) The second record uses a special computational feature: Instead of clicking on a number, we clicked on the name of the variables **rate** and **years**, which therefore appear in the record. This still computes the "formula" entered, which is, in this case, the result of compounding a certain rate of interest for a certain number of years. But, since **record** contains symbolic values, you can easily edit **rate** and **years**, and then **do-record** will compute a new result. Again, this really entails almost no extra learning: Any user of Boxer knows how to name a box to create a variable. And it requires almost no effort for the developer: Inside the calculator mechanism, within the Boxer code that runs it, a variable is always as good as a explicit number.

• *Computational sense (b): Boxer open tools can make use of a wide range of computational structures to make them both more flexible and easier to construct. For example, the written representation of a process can always be abstracted by using symbolic names, variables, in place of explicit values.*

Since computational sense (b) may be unfamiliar, let me further exemplified. This calculator politely explains obscure errors by replacing the introductory text "Andy's handy dandy..." with a message. Many Boxer tools are similarly informative.

• *Boxer open tools are often self-documenting; they can "talk to you."*

Of course, this idea is not special to Boxer tools, but could be implemented within any tool or component. However, Boxer makes this sort of thing particularly easy. The introductory text happens to be just another (unnamed) variable that is set by the calculator mechanism according to its internal state. (Boxer allows unnamed variables by using a special structure called a port. You may use a port to a box in any place you might have used the name of the box.) The generalization of this fact is critical for the open toolset program. For moderately expert Boxer programmers, creating a tool like this calculator is very easy, given the wide range of simple but useful computational capabilities built in to Boxer.

Here is yet another example of computational sense (b). The part of the calculator that re-runs the **record** is essentially a one-line program that sequentially executes each row of the **record** box as if it were a programming statement. The whole calculator was about a day's work for me, and most of that was design rather than coding. "A larger number of simpler tools" is well within reach if each tool takes a day or so to create. And, once again, we can turn our efforts to activity design and improving the tools through extensive use rather than "getting it right the first time."

Let me begin showing some tools whose pedagogical use is more novel than the calculator's.

A Simple Graphing Tool

Figure 3 shows another Boxer open tool. It is a simple graphing utility designed mainly to show time-parameterized data. Again, this was essentially a one-day design and programming project. The basic operations of the grapher are controlled by a pulldown menu that appears when you press your mouse button anywhere on the graphing tool.

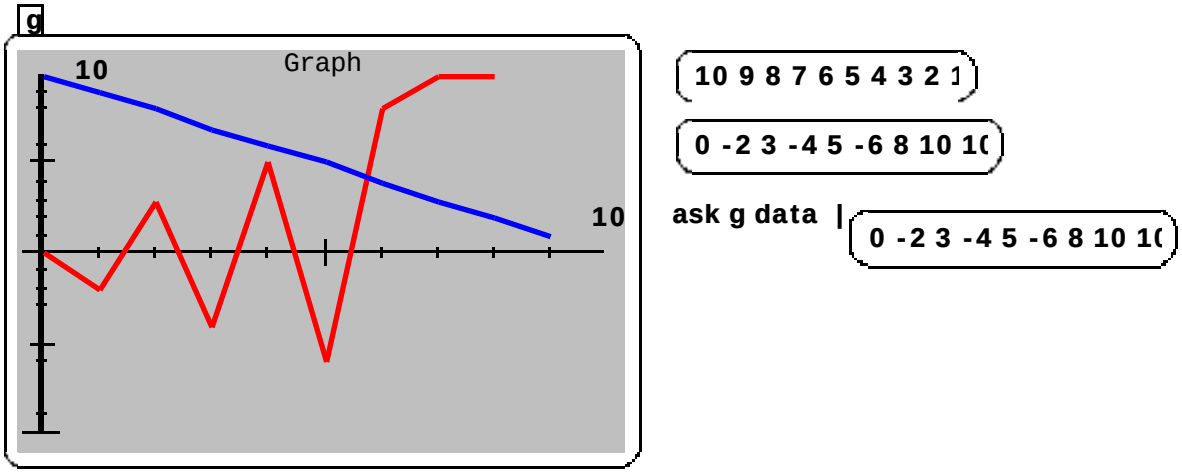

Figure 3. A graphing tool (left) and some of the surrounding environment (right).

Integration with the work environment in this case includes several capabilities quite similar to the calculator. In particular (after using the pull down menu to select **graph-data**), you can just click on a box full of numbers to graph them. In Figure 3, the top two boxes to the right of the graphing tool were graphed, changing the color of the graph in between. A similar capability is that one can direct the graphing tool to create a graph in real time, as a simulator runs or as some data arrives from an external probe. In this case, the ability to script the graphing tool is important. You can just "**ask g plot *a-point***", where ***a-point*** is whatever datum you want to plot. After you collect a set of data in a graph, point by point, you can also "**ask g data**," which results in handing you the set of values last plotted (see the third item in the column to the right of the calculator in Figure 3). Of course, you can cut, copy and paste the graphing tool wherever you wish. In case you don't want a copy of the full tool, but just the picture it shows, you can use another generic Boxer capability. **Snap g** (**snap** as in "snapshot") returns a copy of just the graphics of any tool like this one.

"Recording a process," like the calculator's **record** feature, takes the form of a pulldown menu option to draw a graph with the mouse. **Ask g data** then fetches the numerical data that you entered graphically.

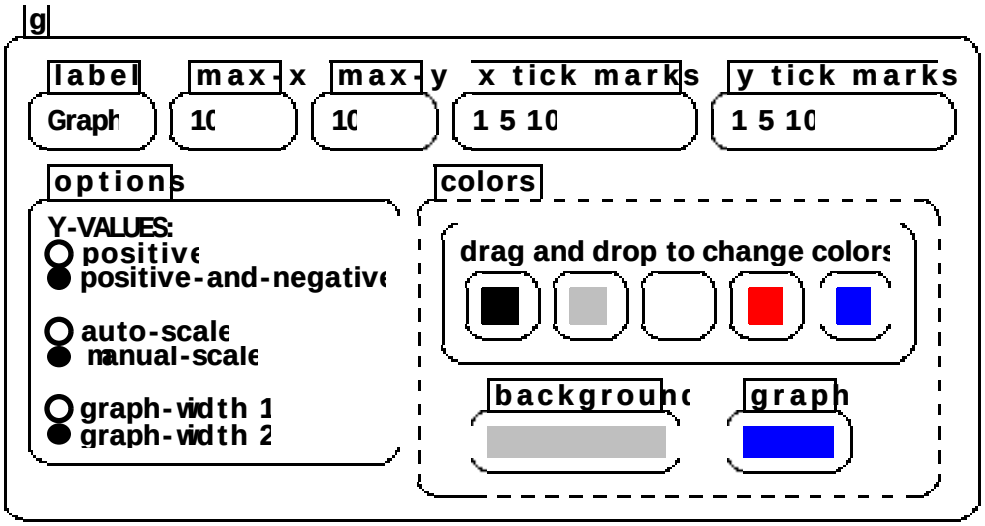

Figure 4. “Flipping” the graphing tool box shows its settings.

Figure 4 shows the built-in options that you can adjust for producing graphs. As I mentioned before, you can “flip” any graphics box to see its “real” box contents, and these options are on the flip side of the graphing tool box. You can adjust the label of the graph, the maximum x and y values, the places where tick marks are shown on the axes, whether the graph plots negative as well as positive values, whether the scale (maximum y) is automatically computed for a given set of data, and the width and color of the graph lines, as well as the background color of the graph. In showing these options, my real motivation is to show another aspect of an open toolset. Tool building is recursive! These options were created by cutting and pasting other existing tools into the graphing tool. In particular, the little subtool that allows you to drag and drop new colors onto the color variables was taken directly from a general tool library we are accumulating. Similarly, the “radio buttons” that allow selecting some options were taken from the same library. Finally, the pulldown menu that controls the graphing tool was also taken directly from that library (as were the on and off buttons in the calculator).

• *Existing open toolsets make building new open tools much, much easier.*

As a token of the many uses that a graphing tool like this may have, and also to illustrate the teachers’ role in an open toolset development community, let me recount an experience. At a workshop to introduce some high school teachers to Boxer, I found myself demonstrating this little graphing tool to a mathematics and science teacher. His eyes lit up as he suggested that this would be a fine way to have his students get an intuitive idea of how a derivative relates to a function. He imagined that the student would draw a function, and the graphing tool would overlay its derivative. Figure 5 shows the essence of his wish, which we realized in about 5 to 10 minutes. It is a little program that scripts the graphing tool, named **g**, to (1) change the graph color to red, (2) draw a graph of the pair-wise differences in the data of the existing graph (which, presumably, the student had drawn), and (3)

change the graph color back to blue for the next student-drawn graph. (**`Differences`** is a one-line program that a fluent Boxer or Logo programmer can create almost instantly.)

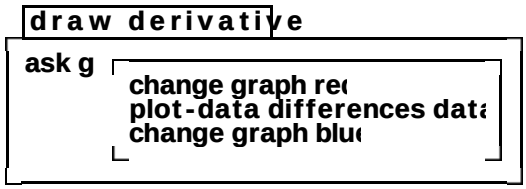

Figure 5. A program that overlays the derivative on a drawn graph.

Here is the generalization: Good tools suggest interesting uses to teachers. Even if the teacher is not up to programming the modification or extension, another, more technically expert member of the community can often quickly extend an open tool in the way the teacher would like. Then it's off to the classroom to see how this works.

Vectors

This example is of a tool type that is both particularly powerful and also particularly characteristic of Boxer. Several years ago our group developed a course on physics for sixth grade students (diSessa, 1995). As part of the course development, a graduate student, Bruce Sherin, programmed an extension toolset that allowed us to use vector quantities in Boxer in pretty much the same way as one usually uses numbers. First, a special keystroke created a vector, which appeared as an arrow within a box. One can adjust any vector by dragging its tip with the mouse. Of course, vectors are scriptable, so any program can also command a vector to change. In addition, similar to the grapher, the flip side of a vector shows its coordinates, which are directly editable. Like any object in Boxer, a vector can be named. In addition, the toolkit provided commands to add vectors (**`add vector1 vector1`**), to multiply a vector by a number (**`mult vector number`**), and to have the vector have effects on other graphical objects (**`move vector`** causes a Boxer turtle to move the length of the vector in the direction of the vector).

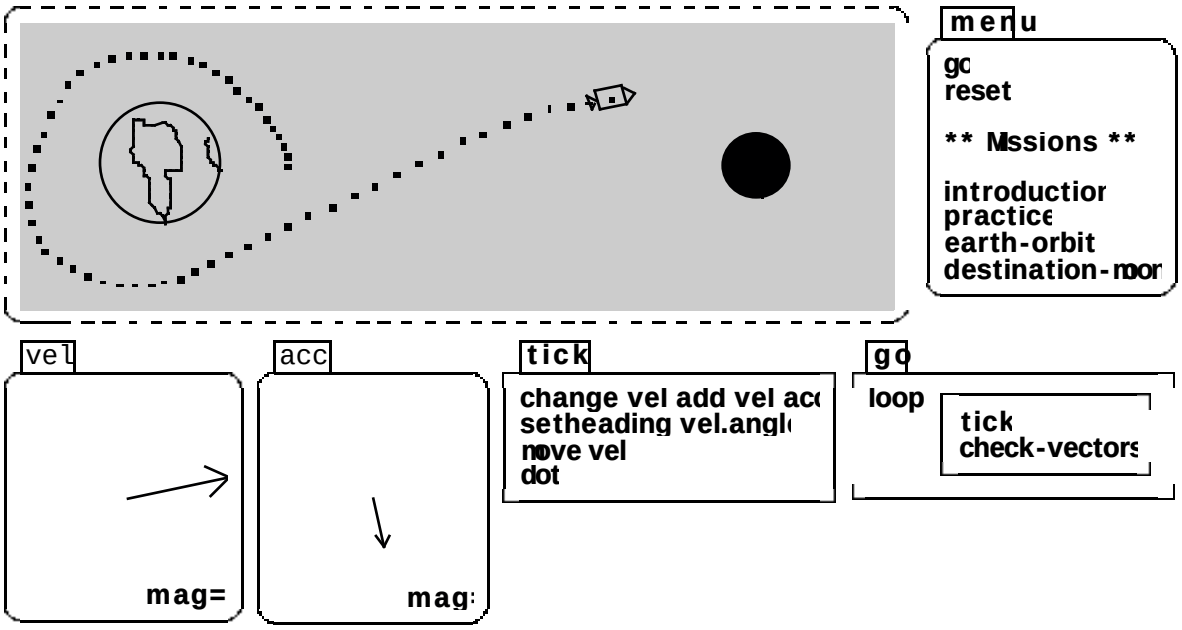

Figure 6. A vector microworld in which students drag velocity or acceleration vectors to control a space ship.

Adding vectors to Boxer in this way is nearly the same as adding vector literacy to the range of competencies that can be fostered within this medium. It is about as powerful as having numbers on the keyboard and numerical calculations in a programming language.

Figure 6 shows a simple exercise microworld built using vectors. In it, students are requested to drive a space ship around the earth and moon by adjusting the acceleration or velocity of the ship in real time. The task is quite entertaining and challenging. It was a significant part of our instruction on how to understand such things as velocity and acceleration as vectors, but also on how to think about complex motions in terms of vectors.

One of the nice things about this microworld is that nearly the complete code for it is right there for students to inspect or copy. The procedure **`tick`** shows what happens "each tick of the clock." First, the velocity is incremented by the acceleration (i.e., the velocity is changed to its old value plus the acceleration). Then the space ship is directed to **`set`** its **`heading`** in the direction of the **`vel`**ocity's **`angle`** of pointing. The ship then **`move`**s according to its **`vel`**ocity, and, finally, the ship makes a **`dot`**. The other procedure, **`go`**, simply repeats **`tick`** over and over, along with **`check-vectors`**, which allows vectors to be changed while the space ship is also moving. **`Check-vectors`** is part of the vector toolset.

It should be evident how simple the vector toolset makes it for teachers or curriculum developers to make a very wide range of exercise microworlds for students. We used vectors dozens of times in the original physics course and in subsequent versions. For example, we made a simple tool that allowed students to

analyze (in terms of velocity and acceleration) stroboscopic images they had created of tossed balls. In addition, most of what was given to students was exceedingly transparent. Students were expected to look at the code of the microworlds, like the space ship simulation in Figure 6, and learn from it.

Most impressive, vectors became thoroughly incorporated into the student culture. Many students made video games using vectors and the fragments of vector code they learned in exercise microworlds. Thus, a simple toolset that nonetheless introduces a powerful idea (vectors) flexibly into a computational environment like Boxer showed to us all the promise that we feel open toolsets may have in many other instances; evident utility to curriculum designers, teachers, and also students.

• *New graphical/computational objects, like vectors, may be among the most flexible and powerful of open tools for curriculum developers, teachers and students.*

In case you believe vectors are a special case—and certainly they are in some respects—Figure 7 shows a structurally similar tool. In this case, we are entering the area of genetics and evolution. The basic command, **`new-creature`**, creates an animal (they are called “scats”) with a certain genetic makeup (Figure 7a). The animal’s **`scale`** is a random number between 20 and 30 and its **`eye-color`** consists of one green allele and one blue one. If you flip the scat’s box, shown in Figure 7b, you see that its insides consist of computational versions of its phenotype and genotype. In this case, the genotype **`scale`** translates simply into the phenotypical **`size`** of the animal. The scat’s eye-color is a bit more complex. The two alleles, green and blue, interact, with the green allele being dominant. Thus a green-blue genotype results in a green phenotype. You can change the genotype directly “inside” the scat and see how the phenotype (automatically) changes.

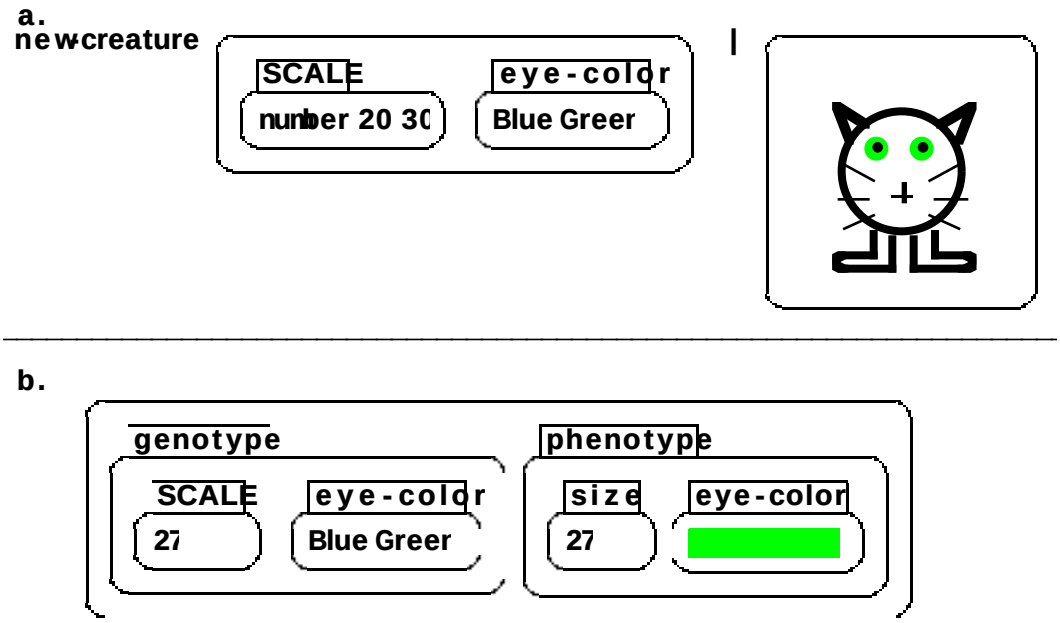

Figure 7a. **New-creature** produces a graphical/computational object.
7b. The "flip side" of the object shows its computational structure.

With such a toolset, it is easy to set up a simple situation where scats breed with one another and produce offspring. Then you can "play Darwin" by selecting the scats you want to breed for the following generation. You can select for size, or eye color, or whatever you like. It happens that scats are easily extendible to add other genetic characteristics, and you can also change the little program that "expresses" (computes) the phenotype in terms of the genotype. Or write a different program for selecting scats to breed, and so on.

Modeling Kits

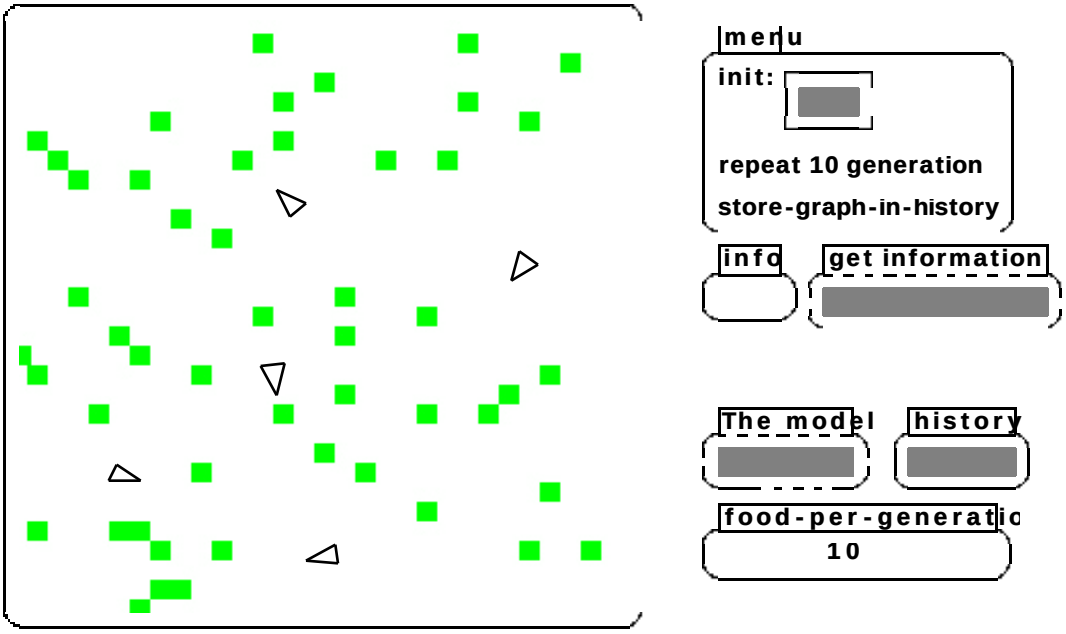

Figure 8. A configuration of a modeling kit that allows exploration of creatures (triangles) living in balance with their food (gray squares).

Figure 8 shows a toolkit to allow modeling of both ecological and evolutionary phenomena. On the left is a field in which creatures (the triangles—let's call them turtles) wander around, foraging for food (the gray squares—they are actually green). Each turtle has built-in properties, like its age and energy level (corresponding to stored calorie reserves). In addition, the kit has built-in functions to "**birth**" new turtles of any specification, to cause a turtle to **die**, to generate a certain number of food squares, and so on.

The configuration of the toolset shown is arranged to facilitate a certain kind of experiment. The **init** command sets up a certain amount of food and a certain number of turtles. In Figure 8, the gray Boxer boxes on the right are actually shrunken, but they may be clicked on to open and make available their insides.

The **generation** command runs the simulation for about one life-span of a turtle. **Info** provides helpful information to users while the models are running, similar to the greeting box in the calculator. **Get-information** contains resources to find out many things about the current state of the model: like the number of live turtles, their ages and energies, the amount of food available, and so on. **The model** contains the specification of how turtles work, and it is meant to be adjusted by students. The **history** box is simply an empty place in which students can keep notes about their various experiments. The last line in the menu just puts a graph of the population of turtles and amount of food into the history box. You shouldn't be surprised that the graphing tool described above is used to generate those graphs.

So, again, available tools make building additional tools easier.

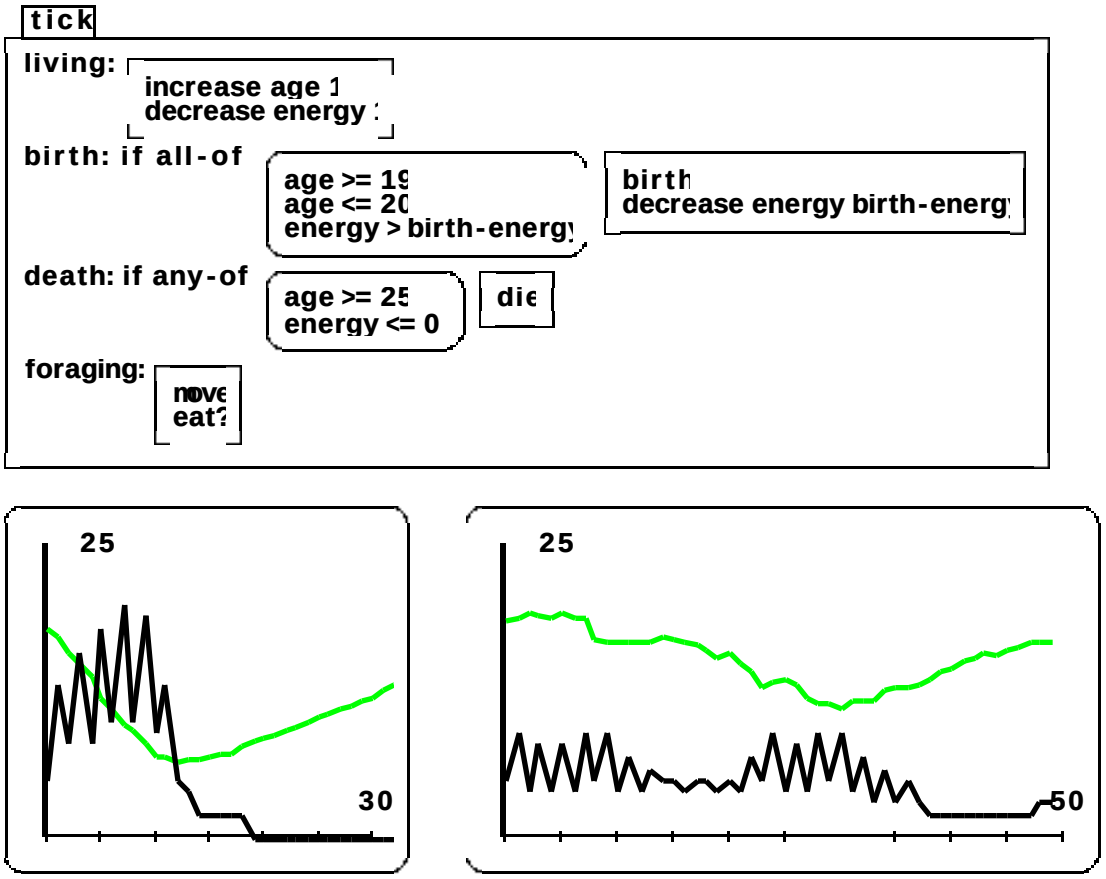

Figure 9. The progam that defines behavior of the creatures, and two graphs of their population (black, jagged) and food supply (gray, smoother). The first graph shows "boom and bust," and the second a relatively stable, limited population.

The top part of Figure 9 shows the initial turtle model—what the turtle does each "**tick**" of the clock. This has four parts: what happens during the normal course of living; the conditions for and affect of giving birth; the conditions for dying; and the foraging behavior (in this case, **move** is just some random motion).

The first graph in the bottom part of Figure 9 shows a typical behavior of this ecological system. It shows the turtle population (darker line) quickly but irregularly increasing while the food supply diminishes. This "boom" is followed by the expected "bust" part of the cycle; with food greatly diminished by many, hungry turtles, the large turtle population quickly dies off, leaving the food supply to replenish gradually in the absence of turtles. Keeping a stable population, in fact, is quite a difficult task, as I discovered myself when I first started to play with this modeling kit. My first solution was simply to put a cap on the population of turtles. I inserted a condition for birth that there be no more than 8 turtles. While this is artificial, it is effective, as shown in the second graph in Figure 9.

The first trial of this modeling kit with students actually was with student teachers in a secondary mathematics and science teacher credentialing program. The initial pedagogical difficulty was surprising. The student teachers of biology—who were quite steeped in the details of complex, real biological systems—rejected the possibility that a simplified mathematical model like this could tell anything about the real world. Even the "expected" boom and bust cycle was not convincing. I think this is not an accident of the population of teachers. Instead, knowing how mathematical modeling makes sense, in view of the simplifications it must make, is an important instructional goal.

The way this group of students completed their study illustrates some important points about open toolsets. In particular, they used the modeling kit in ways I had not anticipated. While I had provided means to get information about the live population, they wanted to do a post mortem on dead turtles to see why they died. Because the kit is open to inspection, they could easily delve into the internals of the kit to find and examine dead turtles.

The solution the students eventually found to solve the boom and bust problem is also illuminating. They noted that all the turtles age together, give birth together, and die of old age together. (From their post mortem, they discovered that almost all the turtles were dying of old age, not food insufficiency. The turtle population was dying, it turned out, not because lack of food was starving them to death, but because insufficient food kept them from having sufficient energy to give birth to a new generation!) So, the teachers changed the birth conditions to allow turtles to give birth over a longer age range. The resulting more diverse population, in fact, turned out to be significantly more stable! Before, the simulation produced a sequence of critical periods when the whole population is fertile together. At those times, the turtles must have sufficient energy to propagate, or the population perishes. Now, given a greater range of ages, "out-of-synch" turtles may weather tough times to propagate when food has regrown. Again, I did not anticipate this sort of change in the model. However, because the model was simply a program that the students could change, it could easily accept their innovative and excellent ideas.

Reflections on the Research Program

I find the possibilities of open toolsets intriguing. (1) They appear to offer new directions for learning with excellent properties. (2) They may solve some difficulties in current computer-based instruction, like how to make software adaptable in the classroom. But open toolsets are also intriguing for pointing out how little we know about those possibilities. They even suggest that the generality of our current research paradigms is limited with respect to answering important questions. Let me explain.

I would divide a research program on open toolsets into three overlapping levels. The first is the one we know most about. It is the learning level. Most educational research on mathematics and science instruction documents and seeks to explain particular conceptual difficulties that students have in learning particular subject

matter. This level finds a new context in the use of open toolsets. But, presumably, we can continue our research on student learning into these contexts. This includes both cognitive studies of individual difficulties and more socially oriented studies of learning-in-context.

The second level begins to outstrip current interests and paradigms of study. This is the level of "tools and activity structures." I mentioned earlier how important interest and personal (and communal) involvement is for the success of open toolsets. One example was the fact that vectors were taken up into the student culture, with consequent greatly extended learning time. Students simply cared to use vectors to accomplish goals they understood, so the properties of vectors in accomplishing those goals became important to them. A great deal of lip service is paid to motivation, especially in designing instruction. However, our understanding of the details of interest patterns, particularly how they can evolve, and their connection to competence and material support (like open toolsets) is extremely meager. I am struck at how little study is conducted that looks at any long term, felt-to-be-coherent engagement of students.

The third level focuses on "social niches and patterns of appropriation." Change in cultures and communities is a critical barrier that we face in improving education. There has been a significant amount of study of issues relating to this, largely under the rubric of reform. But almost all of this is in ancient correlational forms. There is hardly any structural study of the evolution of cultures and communities, and even less concern for the role of artifacts (like open toolsets) in facilitating change. Social theory and science, technology, and society studies focus on either stability, or on case studies of change. Neither are particularly apt to prepare us to design better for change. In particular, what current research can tell us whether open toolsets have better social properties than other forms of software? This is not necessarily an issue of turning our attention to societal, structural matters that bear on education. The process of appropriation, and *change* of community practices and values, happens in every classroom that changes to adapt to a different material support system.

References

diSessa, A. A., Abelson, H., & Ploger, D. (1991). An Overview of Boxer. *Journal of Mathematical Behavior*, **10**(1), 3-15.

diSessa, A. A. (1995). The many faces of a computational medium. In A. diSessa, C. Hoyles, R. Noss, with L. Edwards (Eds.), *Computers and Exploratory Learning*. Berlin: Springer Verlag, 337-359.

Eisenberg, M. (1995). Creating Software Applications for Children: Some Thoughts About Design. In A. diSessa, C. Hoyles, R. Noss, with L. Edwards (Eds.), *Computers and Exploratory Learning*. Berlin: Springer Verlag, 175-196.

Epistemology and Learning Research Group of the MIT Media Laboratory (1991). *Constructionism: Research Reports and Essays, 1985-1990.* Norwood, N.J. : Ablex Publishing Corporation.

Roschelle, J. & Kaput, J. (1996). Educational software architecture and systemic impact: The promise of component software. *Journal of Educational Computing Research*, **14**(3), 217-228.