3 Collaborating via Boxer

Andrea A. diSessa

Collaboration, as an educational strategy in mathematics or science teaching, is usually thought of as arranged by creating the proper social organization and spirit in the classroom. In this chapter I examine how collaboration may be supported by material means. I present several case studies of the use of the computer system, Boxer, in collaborative modes, and I identify the reasons Boxer seems especially good at supporting collaborative work.

Introduction

Collaborative learning is a watchword in contemporary educational reform, especially in the United States of America. Socially and collaboratively oriented images like "a community of learners" (Ann Brown and colleagues) or "cognitive apprenticeship" (J. S. Brown, 1989) have spread like wildfire. Vygotsky and other social theorists have a strong beachhead in thinking about classroom learning.

In this chapter I will not argue for or against collaborative learning. Instead, I accept collaborative strategies as part of a balanced repertoire that we need to understand and enhance in the pursuit of educational goals.

In seeking to enhance collaborative modes of instruction, the most obvious parameter at our disposal is the social organization of the classroom and of classroom activities. I focus, instead, on a less obvious but perhaps no less important issue, the nature of the material basis for mediating collaboration. Written language, of course, may be the best example of this, but technology is a more malleable and redesignable material substrate, hence worthy of particular consideration. Even given a focus on technology, there are more or less obvious things to which to attend. Network communications systems and explicitly collaborative software are among the more obvious foci. Again, my attention is in a less obvious, but arguably not less important direction.

Boxer is the name of a flexible, general purpose computational system designed to serve the needs of students, teachers and educational materials developers. Boxer has all-the-time accessible resources for text and hypertext editing, for dynamic and interactive graphics, for complex data handling and, centrally, for programming. Although I shall show and talk about some of Boxer's features, readers are referred to other articles for more detailed descriptions (e.g., diSessa and Abelson, 1986; diSessa, Abelson and Ploger, 1991).

At the top level, Boxer's goals are overtly social. We want to create a genuinely new and powerful medium—like written text only extended by essentially new capabilities computers can add. We would like to see this medium adopted by as broad a group as possible as the basis of a new literacy that can better foster, especially, mathematical and scientific thinking (diSessa, 1990).

At the next level, however, Boxer does not have on its surface any particularly collaborative features. Instead, Boxer was designed with an eye toward (1) learnability and comprehensibility, and (2) general expressiveness. Yet we have discovered, somewhat to our surprise, that some of our best successes have come from collaborations mediated by Boxer. This chapter presents and seeks to understand some of our early collaborative successes.

In retrospect, Boxer's success as a collaborative medium should not have surprised us. Our analysis here suggests that the same characteristics of Boxer that account for its comprehensibility and expressiveness also account for many of the ways in which it is a good collaborative medium. As with natural language, written or oral, collaborative structure need not be explicitly visible to be effective. You cannot see the fundamentally collaborative aspects of language in syntax, grammar or lexicon

A series of case studies follows. After the first case history, I present an analysis of Boxer's characteristics that help explain what happened. Further examples use and extend this analysis.

First course

The first organized Boxer course (a summer course for students age 13-16 on statistics) was, not unexpectedly, somewhat ill-prepared (Picciotto and Ploger, 1991). The software implementation at the time was incomplete and buggy. Machines crashed frequently and their 16 MHz 68020 processors (less than half the speed of entry-level Macintoshes now) fairly crawled. The teacher we recruited was superb and experienced with programming in BASIC and Logo, but he had not taught a statistics course before with any technological help, and he had received almost no Boxer training. There was no Boxer documentation, and we had had almost no experience teaching Boxer to anyone. To make matters worse the course was only six weeks long (only half the time each day devoted to Boxer), so we had little time to recover from false steps; and students in the course had an almost unmanageable range of expertise, from "hackers" to essentially computer-naive individuals.

Planning for the course initially was faulty as well. Buoyed by perhaps overoptimistic expectations with little counteracting experience, the Boxer post-doctoral researcher helping organize the course convinced the teacher to have the students implement every tool they used, from scratch.

In the third week, the course bordered on collapse. Even the students who were experienced in programming had failed to implement a workable version of the very first statistical tool assigned by the teacher. It was time to re-think.

Jumping ahead, by the end of the course each of four teams of two or three students had produced a fine programming project that illustrated and elaborated statistical principles from the course. Two of the four project groups contained computer-experienced students who, not unexpectedly, created fairly impressive products. For example, one group created a tool that generated "bar and whisker" charts showing confidence intervals on samples drawn from specified populations. The program did not rely on numerical charts or other "tricks," but used an internal simulation of sampling to generate reliable approximations to rather fancy statistical formulas.

Yet even the least sophisticated students also produced excellent working program-projects. One group of three, who, among them, had had only the barest prior experience programming BASIC, developed an excellent pedagogical simulation. They developed a generalization of the standard science museum display of the normal curve emerging from balls dropped into a triangular array of pegs, collecting in bins at the bottom. Their simulation represented sequential samples of binary values (yes, no) from a selectable distribution (say, 55% of the population say "yes") on some polled issue. A "yes" response corresponded to a move downward to the right, and a "no" corresponded to downward to the left. The simulation could be single stepped, allowing process explanations "in slow motion." It could also be run one batch (say, 16) samples at a time (say, producing one "ball" in the "7 yes" bin), or in multiple batches to show the growing histogram distribution. This program collected and sorted numerical data internally as well as showing it graphically.

Although a sample of four projects is scarcely definitive data, our prior experience with summer programs of this sort using Logo were rather different. Many groups failed to produce the quality of projects demonstrated by this class. Even worse, projects combining hackers with less computer sophisticated students were typically dominated by the hackers to the point that other students completely lost ownership of the computer aspects of the project. Only hackers could run and explain their group's program.

Although we were not prepared with the kind of data collection one might have hoped for, we had both the expert observations of the teacher and videotaped exit interviews of students concerning their projects. The teacher also believed these achievements were clearly beyond what he had experienced before. The video tapes, including one narrated by a student who was nearly computer-naive beginning the course, showed excellent mastery of the project programs, including debugging on the fly and expert perusal and explanation of the program's internal structure.

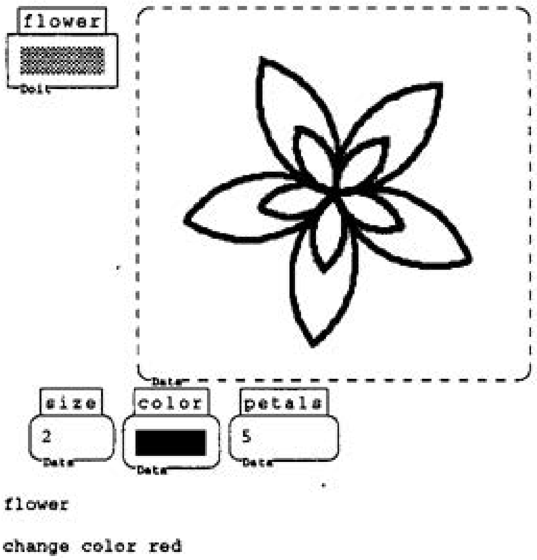
How had the course been turned around from near disaster to unusual success? In a word, code-sharing saved the day. After bailing out from "every programmer for him/herself," the teacher had begun bringing some central program-tools into the class to give to students. For example, he programmed a simple statistical calculator that had several essential statistical functions ready at hand. Graduate student observers dropped in other tools, e.g., a MEMBER? primitive (is X a MEMBER of set Y?) and a simple sorter of numerical data. Students too began contributing to this pool of tools, which other students borrowed.

The transition to a tool-building and sharing culture was not as simple as project members and teacher priming the pump. Some students initially thought borrowing code was cheating. Others kept code they had received private, as if it were theirs to hoard, until this inconsistency was publicly pointed out. Still other students persisted in writing deliberately inscrutable code with great pride to the end of the course.

So, clearly, sharing was a cultural construction and achievement, at least in some measure. But, did Boxer have a role in fostering it? We believe Boxer did have an important role.

In the first instance, after five good years of further experience, we have found code sharing consistently to be a powerful influence in Boxer classes. It is, for example, much more prominent than in any of our many years working with Logo.[1] Continuing examples below help make this point. More profoundly, when we look to see why sharing in Boxer works well, it seems some of Boxer's structural innovations are at the root. Here we list four, which we elaborate and extend in other cases, below.

(1) Visibility—Boxer is designed to allow students to see all aspects of the system as directly as possible. A good example is variables, which are visible and manipulable boxes containing a value. When a variable changes, the result automatically appears on the screen. In contrast, no wide-spread language has a notation for the *fact* of a variable having a certain value. Instead, if variables are shown at all, they are shown in statements that command their existence (declarations) or change a value ("set" or "make"). Not only can you see more structure in Boxer (see also below), but you can dynamically see how programs change their environment. In terms of learnability, the idea of variable, a well-documented problem in children's learning of programming, ceases being a barrier. In terms of sharing, seeing what another's code does and how it does it is much easier. Figure 1 illustrates some aspects of Boxer visibility.

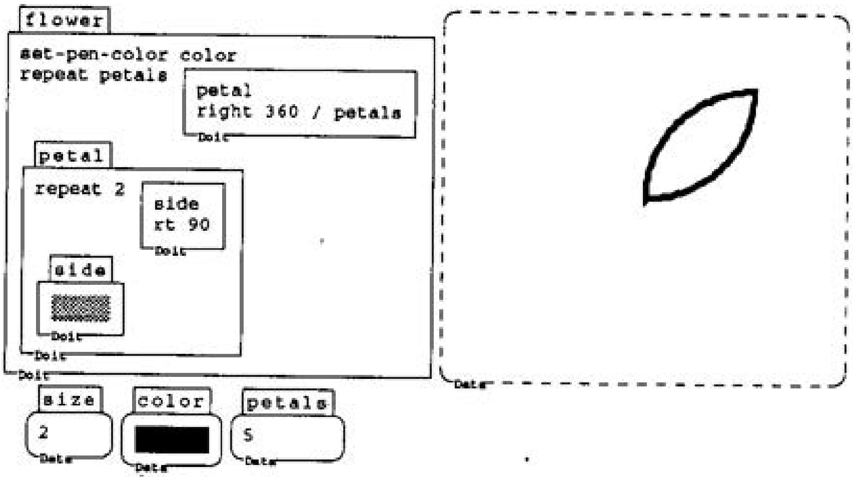


```
flower

change color red
```

Figure 1. All aspects of Boxer are rendered visible on the computer screen. In this case a program, `flower` *(shrunken, to hide its contents), and three variables that it uses are concretely present on the screen. A user may change a variable (e.g. ,* `size`*) by simply editing its contents (e.g.,* `size` *was changed from 1 to 2 and a new larger flower is produced by* `flower`*). If* `color` *is changed by a programming statement, the* `color` *variable will visibly reflect the change.*

(2) Pokability—Variables can be set by hand simply by editing their contents, as well as by program control. More importantly, programs themselves can be poked in bits and pieces to see what they do. Concrete stepping (executing bits of a program one at a time) is trivial and frequently used in Boxer to understand a piece of code. All a user has to do is point to each line in sequence and execute it with a keystroke or mouse click. Figure 2 shows the result of opening flower and stepping one of its parts, petal.



```
flower
set-pen-color color
repeat petals   petal
                right 360 / petals
petal
repeat 2   side
           rt 90
side
```

Figure 2. Opening `flower` *reveals its code. Pointing to* `petal` *and double-clicking the mouse shows* `petal`*'s effect. Note we have also opened the subprocedure* `petal` *to inspect its code.*

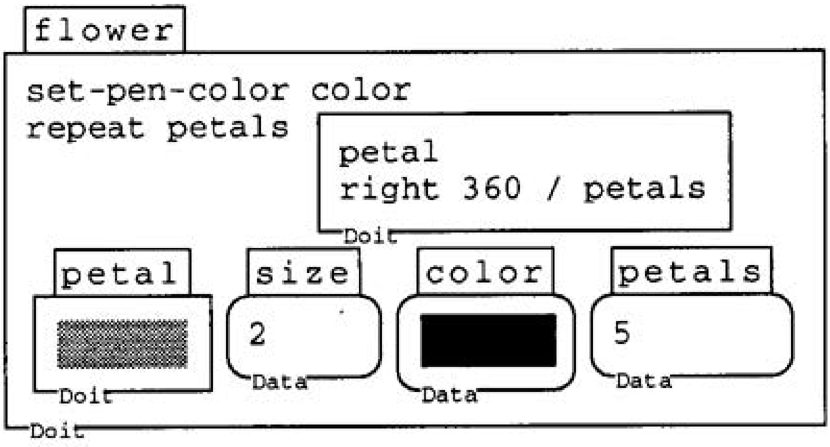
Poking and interpreting the results of a poke is made even easier in Boxer because of another of its characteristics. It is easy to arrange that both code and the output of that code are simultaneously visible on the screen. This makes for very easy association between, say, a program step and the effect it produces. One clicks the former and simultaneously watches the latter. In contrast, for example, it is unusual for code (script) and its effect to be co-present in Hypercard, and this is impossible, in fact, in most Logos, where code only appears in a separate editor or on the "flip side" of the graphical locus of effects of that code. Figure 2 illustrates also how easy the co-presence of procedures, variables and graphics can make interpreting how a procedure works.

(3) Structure—Boxer provides a natural and powerful visual structure to organize code. One uses boxes inside boxes (inside boxes) to organize hierarchical code. Subprocedures may be defined locally in any box, just where they belong, at any level of the

hierarchy. This provides a much more expressive presentation for program structure than the typical list of procedures. Programmers don't *have* to program so as to communicate with better organized structure, but Boxer provides resources that allow this. We have some reasonably well-studied cases of student programmers achieving breakthroughs in programming complexity by mastering the expressive possibilities inherent in Boxer organizational resources (Ploger and Lay, 1992).

Boxer also provides a natural and intuitive way of *perusing* complex code—shifting focus, suppressing or displaying detail as needed. Boxes can be expanded or shrunk, or they can be "entered" by expanding to full screen. This makes for an easy inspection of program structure, allowing students to keep the screen as simple as possible, but exposing as much context as desired (within limits). Figure 2 shows both hierarchical structuring of `flower` and easy perusal via successively revealing substructure. The transition from Figure 1 to 2 involved opening shrunken boxes `flower` and `petal`; `side` remains shrunken, to be opened at need.

(4) "Chunking" into visible, manipulable units—Because of Boxer's structuring, it is fairly easy to arrange meaningful chunks that appear as complete, manipulable units on the screen. So a complex program with many subprocedures, may appear simply as a small black box that can be cut out as a unit, transported and pasted in an appropriate place in another program. Well-designed units "travel" more easily in Boxer than in less concretely organized kindred languages. Figure 3 illustrates that `flower`'s variables may also be encapsulated inside `flower`, and the whole unit simply cut and pasted into a new context.



```
set-pen-color color
repeat petals [petal
               right 360 / petals]
```

Figure 3. `Flower` *and all its subprocedures and variables constitute a visible and functional chunk·that may be cut from an old context and pasted into a new one.*

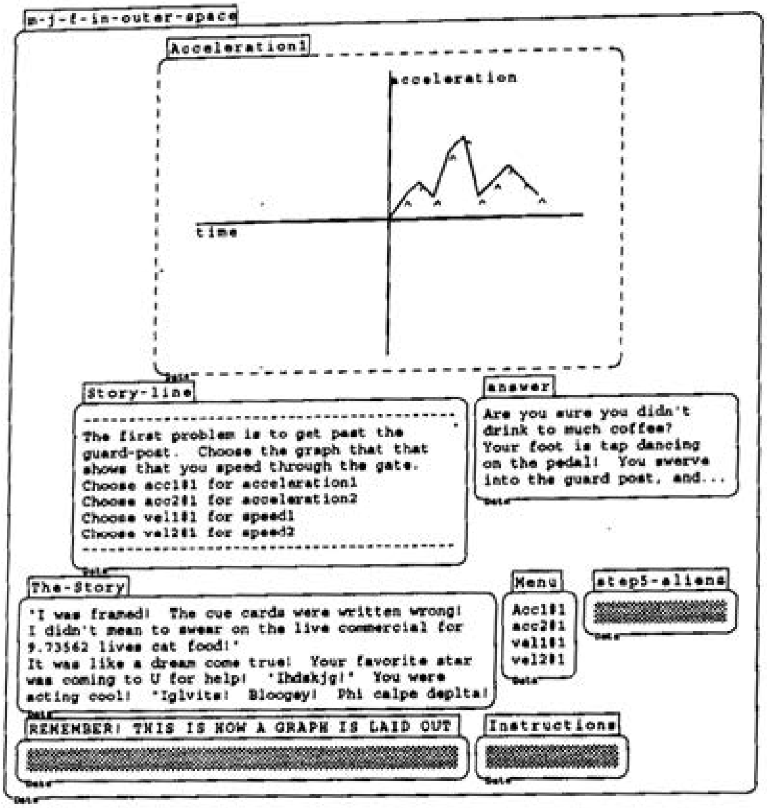
"Travel" is a very apt metaphor for moving things around in Boxer. Boxer's very spatial presence of boxes and text inside of other boxes is not limited to code, but, in fact, organizes every "place" in which users interact in Boxer. So, a very concrete first step to borrowing code is literally to cut it from its old place of use and paste it into its new place, never opening the old code that is borrowed nor boxes containing code in the new context. That latter step is only necessary if processes from both contexts need to be interleaved. Our next case study provides an excellent example of this stepwise process.

We were not in a position to document in detail how sharing worked in this first Boxer class. But the class's success first alerted us to the power of sharing in Boxer and began the analysis that pinpointed areas such as those listed directly above. We could see more directly how things worked in examples below.

Steve's Graphing Adventure Game

Four years ago we gave a full year-long class on physics for sixth graders, age 11 and 12, based on Boxer (diSessa, in press a). Two of the students, Steve and Bill (pseudonyms) produced an impressive program as a final project. It was an elaborate graphing adventure game in which players were invited along on a fantasy journey in outer space. The hero of the fantasy was confronted by a series of challenges involving motion, and the player had to select from among a set of 5 position, velocity and acceleration graphs the one that indicated a motion that would extricate the hero from his current predicament. If you selected the correct graph, you advanced to the next level of the adventure. If not, you would be scolded for selecting the wrong graph (with an insulting but enlightening description of the motion represented by your selection), and you would be sent back to level one to start again. Level five of the challenges was, itself, an independent game of "space invaders," shooting invading aliens with three levels of difficulty.

The program itself was stunning in size and complexity for sixth grade students. It consisted of approximately 500 boxes, with file size over 100 kilobytes. The program stored and displayed dozens of graphs, contained a textual introduction and help with reading graphs, produced interactive text and dynamically changing menus according to the current game context, and it had a cumulative scoring subsystem. The story-line and interactive text was marvelously inventive, filled with typical sixth grade humor. (See Figure 4.)



```
The first problem is to get past the
guard-post.  Choose the graph that that
shows that you speed through the gate.
Choose acc1#1 for acceleration1
Choose acc2#1 for acceleration2
Choose vel1#1 for speed1
Choose vel2#1 for speed2
```

```
Are you sure you didn't
drink to much coffee?
Your foot is tap dancing
on the pedal!  You swerve
into the guard post, and...
```

```
'I was framed!  The cue cards were written wrong!
I didn't mean to swear on the live commercial for
9.73562 lives cat food!'
It was like a dream come true!  Your favorite star
was coming to U for help!  'Ihdskjg!'  You were
acting cool!  'Iglvita!  Bloogey!  Phi calpe deplta!
```

```
Acc1#1
acc2#1
vel1#1
vel2#1
```

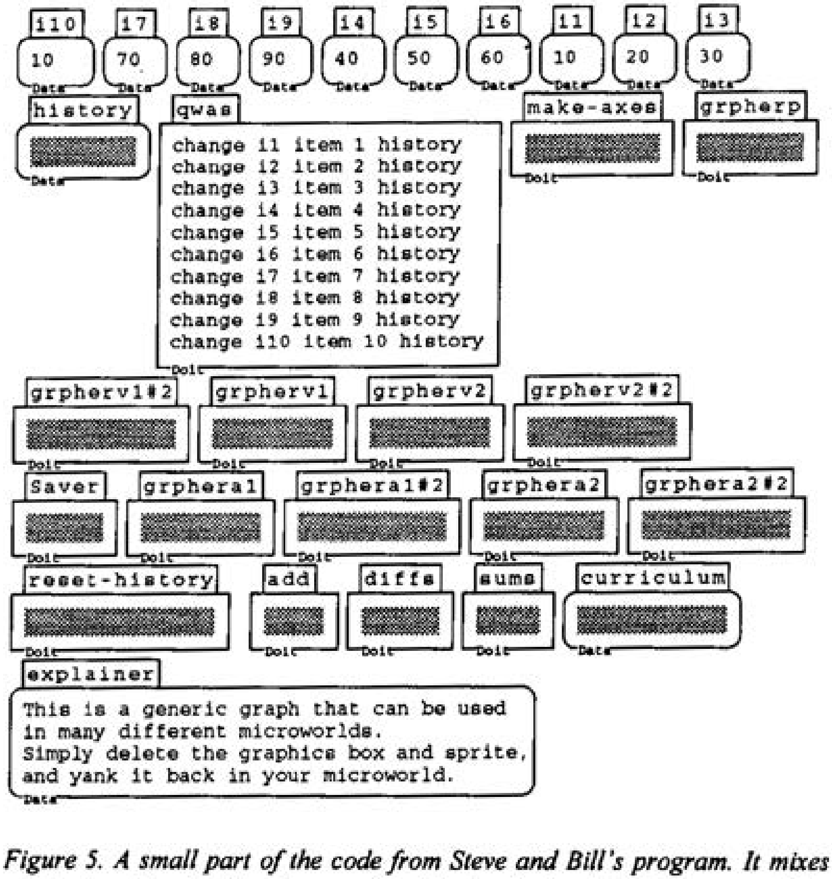
Figure 4. Part of Steve and Bill's graphing adventure game (only one of five challenge graphs are shown). `Story-line` *poses a challenge.* `Answer` *provides feedback (in this case, from selecting an irregular graph for a smooth motion).* `The-Story` *sets the scene and* `step5-aliens` *is an "arcade" game one can enter and play after solving four graphing challenges.*

How did Steve and Bill manage such an elaborate production? First, Steve was an intent and good "hacker." He was the most dedicated programmer in the physics course, but not by a very wide margin. (In the first half of the course, several other students, including one girl seemed as competent and interested.) Steve did, in fact, do almost all the programming for the project, but it seems clear various modes of collaboration and code sharing helped tremendously.

Bill produced essentially all the interactive text for the game and, we believe, he provided the basic form as well. Excellent synergistic collaborations between students with complimentary talents cannot be claimed as Boxer successes, except to this extent: Boxer's visibility and inspectability seem to help ordinarily less involved students keep a general understanding of, and feelings of ownership toward, Boxer programs. Indeed, there are many aspects of producing a typical Boxer program, including box layout (user interface design) and hypertext editing, that involve little more than the skills one learns in the first few days of Boxer experience. These can be manifestly important contributions while still not requiring exotic programming skill. We mentioned the surprisingly good collaboration between expert and less expert programmers in the above case study, and we add others below.

There were other collaborative factors. On inspecting the game's code, I came across some telltale signs. (See Figure 5.) I found, for example, some documentation and instructions for a graphing tool that was obviously unfinished (it contained notes to the author about features to be added and other unfinished and undesigned aspects of the tool). This had been brought into the classroom by a graduate student. Steve saw its value and "borrowed" it.

What Steve took from the tool was quite interesting. First, he reused the basic spatial organization of having a separately named graphing "turtle" in each graph box so as easily to manage multiple graphs. Steve obviously had added new graphs using the same paradigm as the tool he borrowed. Second, Steve used the axis drawing part of the tool. Third, he took the idea of storing graphical data as lists of numbers that was the basis of the graphing tool. On the other hand, Steve left a lot of the tool behind. Much of the working structure was simply deleted from his world, including the basic routine to plot points. Instead, he wrote his own, rather inelegant graphing routine. Either he felt he did not need to borrow that, or he felt he needed something slightly different, or both.



```
change i1 item 1 history
change i2 item 2 history
change i3 item 3 history
change i4 item 4 history
change i5 item 5 history
change i6 item 6 history
change i7 item 7 history
change i8 item 8 history
change i9 item 9 history
change i10 item 10 history
```

```
This is a generic graph that can be used
in many different microworlds.
Simply delete the graphics box and sprite,
and yank it back in your microworld.
```

Figure 5. A small part of the code from Steve and Bill's program. It mixes original code with that from a "professionally" written Boxer graphing program at fine-grain scale. The variables at top and qwas *are part of Steve's (inelegant) rewrite of the actual graph drawing function. On the other hand, Steve borrowed the idea of storing graphs as lists of numbers*

(`history`), and several other ideas and functions (e.g., `make-axes`). The bottom two rows contain pieces from the pirated graphing program that Steve did not use, but neglected to delete.

All in all, Steve's borrowed and new code and their articulation show that he rummaged selectively for ideas as well as code components and had little difficulty joining his own ideas with those of others. Visibility, pokability, structuring resources and clear unit boundaries, we believe, played essential roles in allowing the construction of this complex program.

There is one additional collaborative aspect of this construction that is easy to overlook. Almost all the features of Boxer that allow excellent collaboration and code sharing in general apply to individuals, as well. Good modularity and structure allow easy combining of past work with present. All the inspectability of Boxer helps programmers regenerate and extend an understanding of their "old" code. Thus, many of the good collaborative properties should show up in individual programmers' combining and extending their own work into very complex creations.

Steve and Bill's graphing adventure game had an excellent example of this. Level five of the game, as we mentioned, was an independent subgame that Steve had, in fact, finished as an earlier, independent project. To first approximation, Steve had simply and literally cut this game out of its original context and pasted it in the middle of the graphing adventure game. (It is `step5-aliens` in Figure 4.) All the working structure of the game—menus, graphical display and hierarchical code—came, intact, in a working visible unit. Steve and Bill only needed to find an appropriate place in the adventure game to add the old game. Contrast this with systems that require linking code by splicing it into the thread of activation, within the loops and eddies of the "main program."

Trivial to perform and easy to conceptualize joining of old and new code is a vital first step. But then customizing and refining are equally important. Given that the old game was played squarely in the midst of the program structure of the new game, interlinking is made easy. For example, Steve and Bill "locked" the invaders game box by adding a command that instantly reshrank the box if someone tried to enter it before successfully completing the first four levels. (Boxer contains easy hooks to activate code on a user's entering, leaving, or changing a box.) This relied on a hidden global variable that kept track of the player's progress. Indeed, on exiting the game, Steve and Bill had a small segment of code to neaten up the space invaders game for a new player and to pass on the score the current player had achieved, upon which some of his/her future fortunes would depend.

Steve put most of his linking code exactly where it belonged. For example, the "triggers" that activated an entry and exit from his `aliens` box were placed in that box. Strongly associating code with place makes an excellent modularity principle in Boxer's overall structure. It is especially powerful on modifying and extending for new and more elaborate contexts.

The People Mover

Among the nice pieces of video data we have from our year in sixth grade with Boxer is one that shows, from start to finish, a two hour programming project undertaken collaboratively by three students. The project was to program a "people mover"—a moving walkway such as found at airports—so that one could experiment with different speeds and directions of walking and walkway motion. Our intent was to create an engaging and programming-mediated encounter with some of the basic ideas of relative motion. Students did learn a fair amount about relative motion from this exercise, in interesting ways. The point I make here is only to note some of the things we learned about Boxer-mediated collaboration from this video.

One of the students in the group was an exceptional Boxer programmer who had been using Boxer for the better part of a year. He, naturally, contributed substantially to the success of the group. But so did the others. In fact, about a third of the time the other two collaborators worked without him, and it is notable that the work continued nearly seamlessly through his departure and return. This was excellent indication of joint ownership.

Of the two non-experts, one had had Boxer experience. The other was a complete computer neophyte. One substantial point about collaboration is that even the neophyte contributed significantly. Some of his contributions, in fact, were critical to the project; they concerned how to computerize (in arithmetic) some of the group's intuitive ideas about relative motion. The important point about Boxer is mainly that the student understood enough of what was being programmed, and how, that he could make productive suggestions without prior programming experience.

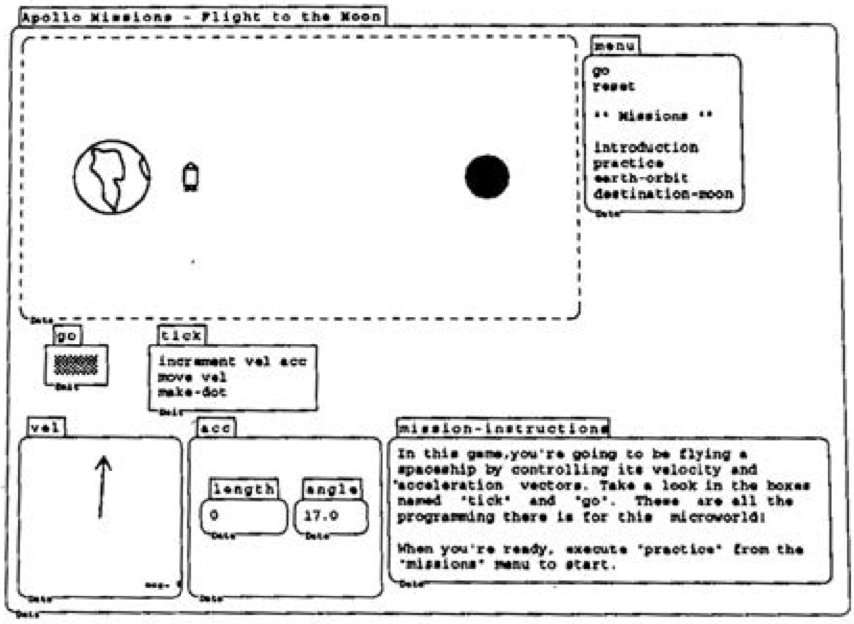
The more general point, which we could see in all of the students' interactions, is that Boxer provides an extraordinarily rich and direct visual presence. Students can not only see a lot on the screen in terms of program structure (e.g., where the pieces are that accomplish various tasks) but also they can see how things operate. As noted previously, Boxer is nearly unique in allowing one to see at the same time the effect of a program and the programming structure that causes those effects.

Visible variables and pokable code allow easy access. This is familiar from "long distance collaborations" (participants not working at the same computer) described in the case studies above. The novel thing we could see in this context was how much the screen served as a place to point and explain, like a super (interactive) blackboard for students to coordinate their ideas and actions, make visible and articulate their contributions. We have noticed in other videos of joint programming how much a contribution the Boxer screen makes to a wide and effective communicative channel (diSessa, in press b). Talking, alone, is less effective. Boxer expressiveness thus manifests itself also in real-time interaction effectiveness as well as in "asynchronous" collaboration.

Vectors

For our sixth grade class and subsequent high school courses on physics, we developed an extension to basic Boxer. In effect, we added vectors as a basic data type. With the addition, one makes a vector by pressing a key, like ordinary boxes. Vectors appear as interactive graphics boxes containing an arrow that one can resize and reorient using the mouse. Vector operations were also added. One can add two vectors and see the resultant, one can tell a sprite (Boxer's mobile graphical objects—like a Logo turtle) to move along a particular vector, and one can name and change a vector like other Boxer variables. In addition, all graphics boxes in Boxer can be "flipped" to reveal a non-graphical presentation. In the case of vectors we chose to show the vector's numerical components on the flip side. These could be manipulated directly or by programming commands as an alternative to direct mouse manipulation of the vector arrow in the graphical presentation.

The sample microworld in Figure 6 is a simple construction using vectors. `Vel` (velocity) is shown in graphics presentation, and `acc` (acceleration) is flipped.



```
increment vel acc
move vel
make-dot
```

```
In this game,you're going to be flying a
spaceship by controlling its velocity and
'acceleration  vectors. Take a look in the boxes
named  'tick'  and  'go'.  These  are all the
programming there is for this  microworld!

When you're ready, execute 'practice' from the
'missions' menu to start.
```

Figure 6. An exercise microworld using Boxer vectors. `Vel` *shows a vector in graphical presentation so that the vector arrow can be directly manipulated in real time while the simulation is operating.* `Acc` *is a vector that has been "flipped" to show coordinates.* Tick *is the complete program that moves the spaceship "each tick of the clock" according to velocity* `vel` *and acceleration* `acc`*, leaving dots along the way.*

Vectors provided extraordinary resources for our whole teaching and learning community. We developed many tutorials and exercise microworlds out of vectors. The flexibility they afforded us is illustrated by one occasion in our high school course. We had had students take stroboscopic pictures of balls tossed and dropped in the air, scanned them in, and were prepared for students to study these motions analytically by driving sprites around over the images. Unfortunately, the lessons on two dimensional kinematics were not going as well as we had expected. We decided to scaffold the students' analysis with a tool that suggested how one should think about the motion and what results one might get. Using vectors we spent about an hour as a group designing and implementing the tool, which was used successfully the next day in class. To emphasize the flexibility and ease of modification Boxer-plus-vectors provided, we note that one student suggested yet another, different type of analysis; with the teacher's help, he succeeded in modifying the supplied tool.

Vectors infused the course more generally. We used simple vector programs to define and illustrate basic kinematics terms, a role that would ordinarily have been taken by algebra (diSessa, in press a). In addition to exercise and tutorial microworlds, vectors spread to students as a central part of personal projects. It is rare that intellectual tools, like abstract vectors, can be concretized to the point that they provide such obviously helpful support for student-initiated activities.

Along with the general properties of Boxer programming that made vectors successful contributions to this community, this example illustrates one key property that, as far as we know, is unique to Boxer. Namely, Boxer vectors were complete, self- contained interactive objects, with their own mouse interface, and yet they were as well first-class programming data objects that could be used in student or teacher extensions in the same way other built-in Boxer objects could be used. Vectors could be named, changed by programming statements, supplied as inputs and returned as outputs of programs. This dual citizenship—as simple screen-interactive objects and also as full-fledged, extensible computational entities—meant easy learnability and near-complete adaptability for students' and teachers' particular needs. As such, they are powerful tools that can grow effectively in particular collaborative communities to serve local purposes.

A Class Box

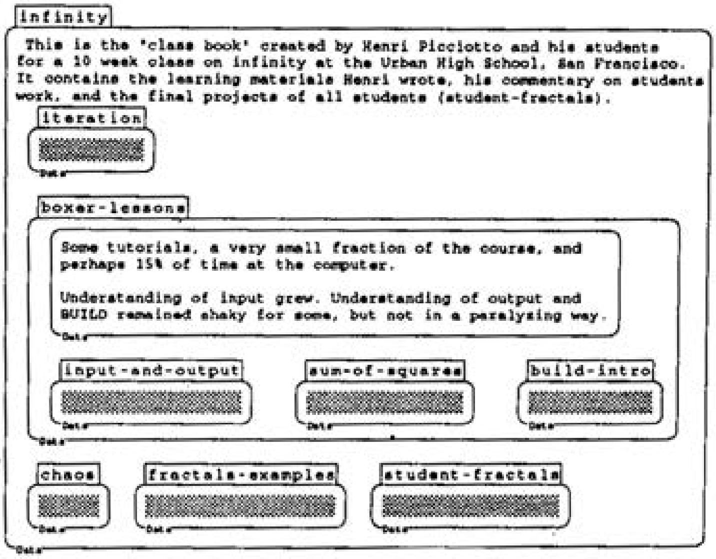
The last case is a very simple but potentially powerful collaborative product. It relies on Boxer's simple, perusable spatial structure, easy composition of complete working units, and the ease with which text and hypertext annotation may be added. The teacher of a high school class using Boxer to teach about infinity and fractals decided to produce a class box. It contained:

a) all the tutorial materials the teachers had developed, in Boxer, to teach both Boxer and the mathematics of the course, including the exercises he assigned;

b) teacher comments on how students did with various materials in the course, including successes and difficult points;

c) all of the students' final projects, each in a mutually agreed standard form, which had working programs, textual explanation of mathematical analyses, and demonstrations. The students worked in groups of two and three, and the final projects of this particular class were visually impressive, artistically presented and generally mathematically cogent.

Figure 7 shows the top level of the class box, and Figure 8 shows an example of student work.


infinity

This is the 'class book' created by Henri Picciotto and his students for a 10 week class on infinity at the Urban High School, San Francisco. It contains the learning materials Henri wrote, his commentary on students work, and the final projects of all students (student-fractals).

iteration

boxer-lessons

Some tutorials, a very small fraction of the course, and perhaps 15% of time at the computer.

Understanding of input grew. Understanding of output and BUILD remained shaky for some, but not in a paralyzing way.

input-and-output

sum-of-squares

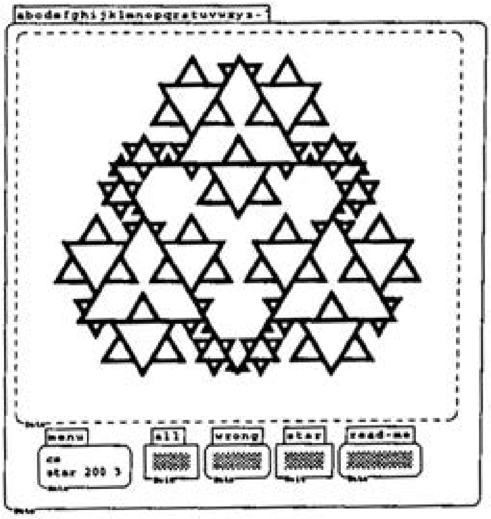
build-intro

chaos

fractals-examples

student-fractals

Figure 7. A "class box" containing all of a teacher's prepared materials and student projects from a class on infinity. One box is opened to show some teacher commentary on how his tutorials worked.


abcdefghijklmnopqrstuvwxyz-

menu

```
cs
star 200 3
```

all

wrong

star

read-me

Figure 8. A sample student project. Typically, students included a complex graphic and some mathematical analysis, say, computing its fractal

dimension (not necessarily an integer), total area or path length. In the `wrong` *box, the student here explains a prior failed attempt at this design.*

In addition to providing a motivating and informative class product, such a hypertext document provides excellent dissemination of instructional ideas to other teachers. Seeing student work is an excellent source of ideas and calibration of expectations.

The Future—Community Boxer

Boxer was not designed with collaboration explicitly in mind. However, we have seen after the fact that many of Boxer's learnability and expressiveness properties contribute to excellent code sharing, and to asynchronous and synchronous collaborations. I have traced Boxer's success in supporting collaboration to properties such as:

- A high degree of visibility of all Boxer structure, which allows collaborators to see what each other has done and is doing.
- The easy capability to activate any part of a Boxer program and observe what it does.
- An expressive spatial and hierarchical structure, which allows one to "put things where they belong" and peruse complex products top-down, successively revealing details at need.
- The possibility to pack complete functional systems into a visible unit that can be easily transported to other contexts.

At present we are considering extending Boxer to better allow multiple machine collaborations. Our goals are straightforward. We want to provide generic multi-machine collaborative support in the same way Boxer now provides general facilities for programming, personal or small group work, microworld and tool building. We want users to design and modify their own collaborative structures in the way Boxer users now develop, share and modify traditional Boxer materials. The means toward this end are also relatively clear. We intend to expand Boxer's current spatial metaphor and its sharing and hypertext structures across network connections. Users may "inhabit" the same Boxer space or share substructures arranged to support many forms of collaboration.

Notes

[1] It is worth noting that the structure and presentation of Logo programs, per se, have not changed essentially at all through the years, even while the programming environment has been enriched with text processing (LogoWriter), better file and other organization, and (especially with Microworlds) ready tools such as a paint tool and prefabricated buttons. It is the program structure and presentation, however, that we argue is the root of Boxer's collaborative spirit.

Acknowledgment

This paper is based on an earlier version that appeared in P. Georgiadis, G. Gyftodimos, Y. Kotsanis, and C. Kynigos (eds.), *Logo-like learning environments: Reflection and prospects*, Proceedings of the fourth European Logo conference, Athens, Greece: Doukas School, 351-357. The work described was supported, in part, by the National Science Foundation, grant numbers NSF-MDR 88-50363 and NSF-RED-92-52725. The opinions expressed are those of the author and not necessarily those of the NSF.

References

Brown, J.S., Collins, A. and Duguid, P. (1989), 'Situated Cognition and the Culture of Learning', *Educational Researcher*, January-February, 32-42

diSessa, A. A. (1990), 'Social niches for future software', in M. Gardner, J. Greeno, F. Reif, A. Schoenfeld, A. diSessa and E. Stage (eds.) *Toward a Scientific Practice of Science Education*, Hillsdale, NJ: Lawrence Erlbaum, 301-322.

diSessa, A. A., (in press a), 'The many faces of a computational medium', in A. diSessa, C. Hoyles, R. Noss, with L. Edwards (eds.), *Computers for Exploratory Learning*, Berlin: Springer-Verlag.

diSessa, A. A. (in press b), 'Designing Newton's laws: Patterns of social and representational feedback in a learning task', in R.-J. Beun, M. Baker, and M. Reiner (eds.), *Dialogue and Interaction*, Berlin: Springer-Verlag.

diSessa, A. A. and Abelson, H. (1986), 'Boxer: A reconstructible computational medium', *Communications of the ACM*, **29** (9), 859 - 868.

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

Picciotto, H. and Ploger, D. (1991), 'Learning about sampling in Boxer', *Journal of Mathematical Behavior*, **10** (1), 91-101.

Ploger, D. and Lay, Ed. (1992), 'The structure of programs and molecules', *Journal of Educational Computing Research*, **8** (3), 347-364.