TWENTY REASONS WHY YOU SHOULD USE BOXER (INSTEAD OF LOGO) *

Andrea A. diSessa
Graduate School of Education
University of California
Berkeley, CA 94720
USA
disessa@soe.berkeley.edu
www.PyxiSystems.com

Abstract

Boxer was designed as a successor to Logo, with the same educational goals in mind. Whereas Logo has incrementally added features over the years, Boxer changes the core computational structures of the language and environment. The aim is to make learning easier and more rewarding, especially over the long term.

* 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). Twenty reasons why you should use Boxer (instead of Logo). In M. Turcsányi-Szabó (Ed.), *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference*. Budapest Hungary, 7-27.

Introduction

This paper explains and advocates Boxer as a computational environment for educational purposes. I intend mainly to speak to the Logo community. However, I hope not to produce a paper for "insiders" only. Instead, Logo stands in for many open educational computing environments as, arguably, the best of the lot. When I say "Logo" in this paper, I mean to include successors like LogoWriter and MicroWorlds.

Boxer and Logo share a great deal in terms of philosophy and purposes. Indeed, the design of Boxer emerged over a dozen years ago, from within the Logo project at MIT. It was an attempt to design a successor to Logo, capitalizing on all we had learned using Logo with children and teachers. Both Logo and Boxer aim to provide the simplest but most powerful and unconstrained computational resources possible to "just plain folks" in the service of enhancing learning. Both projects believe that programming, in some form, is essential in truly liberating the computer's power as a learning tool. Differences at the level of philosophy exist and are interesting, but are not the main issue here.

The differences between Logo and Boxer that are relevant to this paper are technical. That is, they have to do with the designed structure of the Logo and Boxer environments and their associated programming languages. "Technical," however, does not mean either esoteric or unimportant. To go back to the beginning, the difference between text and a programming environment are "technical" in exactly the same sense. But, I think everyone would agree that the difference between what you can say and do with text versus a programming language is substantial.

Logo's principal claim to fame was that it made programming easier and more accessible. It basically adopted a subset of the capabilities of a "difficult" programming language, Lisp, and changed the way people saw and used that environment. For example, Logo had a more friendly syntax. In view of this central and well-advertised advance, it is stunning that the basic form of Logo's presentation to the user has remained essentially identical from the late sixties when it was originally designed. This lack of change is even more surprising in view of two other facts. First, Logo was plainly constrained by the teletype terminals that it originally used. There was no choice except using characters, words and lines as basic structuring devices. Bitmapped graphics and even mice were yet to become serious design possibilities. Second, I believe it was evident even in those days that the restricted structural possibility afforded by a "typewriter" interface caused difficulties. I wrote a memo to the Logo Group in the late 1970s about these limitations and made some suggestions that, eventually, became Boxer. By now, I believe these limitations are even clearer and empirically verified, especially in contrast to Boxer. I'll make reference to many particular limitations and some of the data that confirms these in the remainder of this paper.

I don't mean to imply that Logo hasn't changed in its nearly thirty year history. But, the changes are not, for the most part, in its basic structure. Logo has not taken advantage of dynamic, graphical display possibilities in its core. Programs are still words in sequence, broken into lines, and so on. Instead, changes have included having some internal application-like features (draw program features), some canned interface "widgets" (like buttons and sliders), and most notably, parallel processing. Even the latter significant change, however, involves no change in the basic presentation of the language—only a new command, **`launch`**.[1]

On this background, let me sketch twenty reasons why you should use Boxer (instead of Logo).

1. Lower Threshold

Teachers who have taught both Logo and Boxer tell me that the first stages of introduction to Boxer invariably go more quickly and smoothly. One of my most reliable sources (and, at times, a friendly critic), who is a high school teacher and long-time Logo user, tells me that he can now get through his basic introduction to programming in more like two days compared to two weeks with Logo. This includes learning commands, procedures, iteration, and at least a little about variables.

A second measure of lower threshold is how long it takes for students to get into really interesting, self-directed projects. A benchmark for me in this regard also comes from the above-mentioned teacher. In a mathematics course that lasted only 10 weeks, he brought a not-particularly-brilliant class from the state of being non-programmers to where they each produced both cogent and personally fulfilling projects. This is significantly beyond anything I have seen accomplished with

[1] Actually, there is at least one interesting exception. The invention of "pages" and "flip sides" are directly along the lines of some Boxer innovations that I will be discussing. However, pages and flip sides didn't penetrate the language, even if they are an excellent innovation in the environment. It may be surprising, but Boxer's version of these innovations already existed when Logo "invented" them.

Logo.[2]

```
to square
repeat 4 [fd 50 rt 90]
end
```

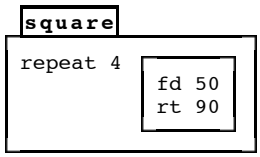


Figure 1. A square procedure definition in Logo (left) versus a visually transparent Boxer presentation (right).

This is not magic, but it results from some basic properties of Boxer. For example, procedures appear as visual entities, boxes. Consult Figure 1. Little simplifications, such as extremely clear visual boundaries, make a lot of difference to beginners. You don't have to learn the special meanings for a little piece of text, **`end`**, as the boundary of an object. In Boxer, you can't have a beginning (**`to`** ...) without an **`end`**, and you can't misspell syntactic boundary markers.

Logo has always, unfortunately, distinguished the mode of creating procedures from the mode of executing them in one way or another. Early in Logo's development, you entered a special mode for procedure creation during which you couldn't execute. Later, there was a separate editor, which became the "flip" side of the page, or the procedures page. All of these separations cause difficulties, especially for beginners. Most notably, you cannot easily—or at all!—see the effects of a procedure at the same time that you look at its form. This makes *learning by inspecting* difficult; you have to flip back and forth between different areas to see a procedure and its effects. In addition, it rules out a mode of *learning by interacting* with pieces of code, which is very powerful and characteristic of Boxer. For example, if you look at a line of code in Boxer and wonder what it will do, you can just double-click on that line, and it will be executed. This also turns out to be an extremely powerful debugging technique.[3] If something goes wrong, you can just step through the process by executing one line at a time. That is, the inherent inspectabiliy of Boxer is extended with "pokability." Without easy visual interpretability, inspectability, and pokability, is it any wonder that Logo beginners frequently just throw away old procedures without re-using them (rather than figuring out what they do)? Or they simply start again with a new procedure definition instead of debugging what exists.

---

[2] I am describing the "infinity" class, whose work and final projects are distributed with the demonstration programs available with Boxer. Go see for yourself if you believe you could accomplish the same amount starting with non-programming students!

[3] Of course, executing a piece of code that depends on a context created by the execution of other code makes debugging simply by pointing to an arbitrary line of code sometimes difficult. However, Boxer makes some of these cases easier to deal with in that if you execute an input line, you get local variables in which you can insert test values.

```
print :x
make "x 35
```

Figure 2. Contrasting Logo interactions with a variable via a conversational paradigm (left), versus direct visibility and access (right), characteristic of Boxer.

Variables are well-known to provide difficulties for Logo beginners. The problem is that Logo variables are very abstract. In Boxer, in contrast, variables are boxes just like procedures (although they are different kinds of boxes; procedures are *doit* boxes, and variables are *data* boxes). So, in Boxer you can literally see variables. Consult Figure 2. You can see their current values, and if a procedure changes a variable, you can see that change. Furthermore, if you want to change a variable, you can just edit its visual presentation as if it were plain text.

Logo never lets you see a variable—only its value—and you must interact with it "conversationally" rather than directly.[4] To see its value, you must execute a command like **`print :x`**, and to change it, you must similarly execute a command. Actually, Boxer is extremely unusual in this regard, as—as far as I know—no other "serious" language provides an interactive notation (you can edit it directly!) for the fact of a variable having a value, as opposed to a notation for the fact of changing a value or the fact of doing something with it. Spreadsheets are another context that shows the power of concreteness (visibility) and the ability to change "variables" (cells) directly.

2. See for Yourself

The reasons that Boxer is better for beginners, such as those explained above, are not accidental or particular to the way we chose to design a notation for procedures and one for variables. Instead, these are part of a larger plan, which accounts for very many of the advantages that Boxer has over Logo and other programming environments. In particular, there are two overarching principles that guided Boxer design. First, there is the principle of *naive realism* (or the principle of "concreteness"). Here, the idea is that the user of a computer system can pretend that what appears on the screen *is* the computer system. That is, you don't need to do a lot of mental work interpreting an abstract presentation that relates only indirectly to the fact of the matter (as, for example, imagining something called a variable that is changed or accessed by commands). Instead, naive realism means that everything in the system must have a visual presentation that allows easy interaction with it, including creating it, changing it, moving it, and deleting it.

One of the wonderful successes of Logo was that it developed the turtle, whose spatial state was always visible for learners to contemplate. So, the practice of semi-programming was born. Students can execute commands one at a time and inspect and think about the state of the world thus created. This is a tremendous boon to beginners as it frees them from the need to imagine a complex state created in the midst of a complex process. But, because Logo has no principle of naive realism for computational objects, as opposed to for the turtle (or other graphical "side effects" of computation), semi-programming can't work to support more abstract programming. In Logo, semi-programming can't work where only a set of variables are changing. With Boxer,

[4] Yes, sliders sometimes provide some of Boxer's inherent concreteness and visibility. But "sometimes" is a long way from "always." Consider how often one wants non-numerical data, and how often a slider is just more work than having a variable.

students can simply watch variables change just as they might watch a turtle. They learn "abstract" programming nearly as easily as they learn turtle programming. This is not just an advantage for beginners; it also helps experts watch their programs in action, and to debug them.

The second powerful and general principle of Boxer is the *spatial metaphor*. You can think of this, once again, as extending an excellent idea of Logo to apply more broadly. In this case, the Logo turtle allows students to use their very well-developed spatial understanding to become engaged in programming. Every child intuitively understands certain spatial relationships. Children can instantly see if the turtle is in the correct place, facing in the appropriate direction. And they can reason through what they want to have happen next. But, the turtle is not computational structure, per se. Boxer uses space and spatial relations systematically to represent aspects of "abstract" computation. In particular, Boxer has a wonderfully transparent hierarchical structure of boxes inside of boxes that represents huge ranges of computational meanings.

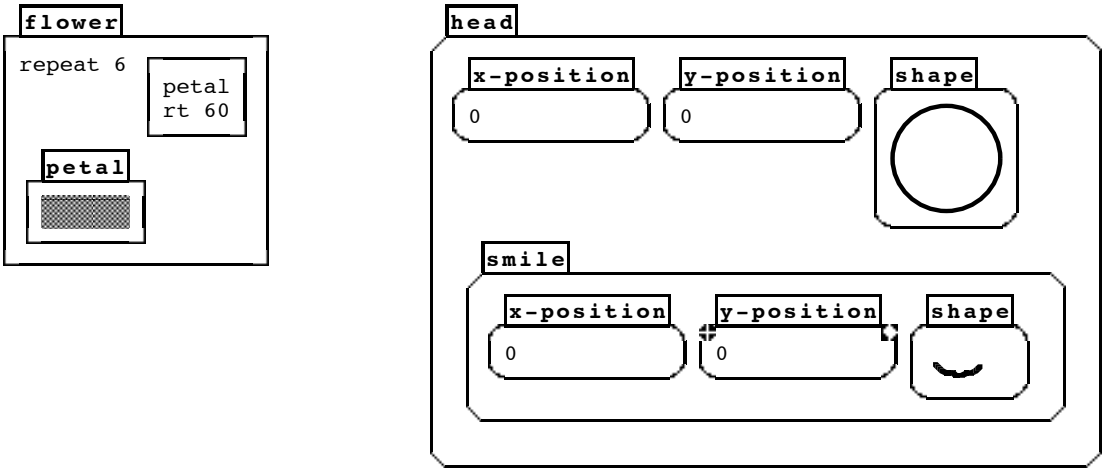


Figure 3. Boxer uses its natural hierarchical structure of boxes inside boxes to represent important computational semantics. On the left, a procedure, **flower**, contains its subprocedure, **petal**. On the right, a subsprite, **smile**, behaves as part of its supersprite, **head**, when the supersprite is moved, but it can also be independently addressed.

For example, procedures inside other procedures represent the "belonging" of a subprocedure to a procedure. See Figure 3. Similarly, the boxes that represent turtles (we call them sprites) may be nested one inside the other. A subsprite thus created moves with its supersprite when the supersprite moves, but moves independently within the frame of the supersprite when you direct commands to the subsprite. Subsprites make excellent components of objects, for example, arms or eyes of a person or animal that you want to have move with the complete animal. Yet they can also have independent actions, like raising an arm or winking an eye.

Boxer's spatial hierarchy represents literally dozens of computational relationships in a way that we have found is very natural. Look for continuing examples in the sections below. The spatial metaphor has proven much more successful than even we initially believed. For example, we thought people might get lost in a maze of boxes inside other boxes. One of the first utilities that we designed before we tried Boxer out with people was a map utility that showed the structure of your universe and provided a "you are here" indication of your current location. As a matter of fact, this proved completely unnecessary. After just a very little experience with Boxer, students

never get lost. Better said, if they get lost, they understand how Boxer is organized well enough to find themselves without difficulties.

3. Higher Ceiling

In addition to a lower threshold, Boxer provides a higher ceiling than Logo. This is, in part, a difference of orientation. Logo was designed for children. But Boxer is designed to grow with children into adulthood. One of the disconcerting things I found with Logo was that teachers learned it only for their students. It didn't really serve any of their adult purposes.

Simply put, students who stick with Boxer get a lot further than those who continue with Logo. I have several benchmarks. First, in all of our experience with Boxer, essentially every student in courses that last more than a few weeks has managed to produce a cogent project at the end of that time. This was not my experience with Logo. In addition, exceptional students have substantially surpassed what appears possible with Logo. For example, in a Boxer class we gave, a pair of sixth grade students who were certainly clever—but clearly not prodigious—managed to create a huge "graphing adventure game." The game presented players with dozens of graphs to interpret, kept score and had included help and a "reward" video game to play. This program contained hundreds of boxes and was larger than the average construction I program myself in Boxer. Exceptional high school students have created immensely complex programs. One example was a "molecular toolkit" that contained tools to analyze organic molecules, to display visual presentations of their structure (given only their chemical formula), to name them automatically, and so on (Ploger & Lay, 1992).

Some of these accomplishments, and the higher ceiling for Boxer generally, come about for completely obvious reasons. Boxer provides many more advanced facilities compared to Logo, including different styles of programming (see below), a much more flexible and reconfigurable environment, advanced structures (e.g., compound graphical objects—the subsprites described briefly above) and so on. I will describe some of these in more detail in subsequent sections. However, other reasons that Boxer has a higher ceiling are more subtle, although equally important. These reasons are what I wish to discuss in Boxer advantage number three.

I already said, but it bears emphasizing: The characteristics of Boxer, naive realism and spatial metaphor, which, in part, make it comprehensible to beginners, also help relative experts. Inspectability and pokability help one understand and manage complex programs, as well as understand how simple programs are created. It's a lot easier to inspect the state of your program by watching its variables than to try to imagine what is going on. And executing a little piece of a large program is such a powerful part of debugging that making that very easy—as it is in Boxer—pays huge dividends.

A problem that occurs with Logo is a set of plateaus that appear regularly with respect to structured programming. First, students hardly ever begin programming in a structured style on their own. Instead, they produce "spaghetti code" programs, if ever they create large ones. This is not a cognitive limitation or even bad instruction (at least, not entirely), but it is a case of the expressive environment not facilitating effective organization. Contrast the visually clear capability of Boxer to put local procedures and local variables directly inside a superprocedure. (Again, consult Figure 3.) In contrast, Logo subprocedures at best follow their superprocedure, and there is no automatic and evident visual connection. Complex procedures with many subprocedures tend to become a disorganized jumble, unless one takes great care and invents ways of associating who belongs to whom. Obviously, what goes for local procedures also goes for local variables—except, arguably,

the situation is even worse.[5] Again, you must use a "conversational" technique of declaring a local variable, rather than just putting one where you want it. And if a subprocedure calls a local variable that is not in that subprocedure, you have to do a complex process of guessing and finding superprocedures that call your subprocedure to see which one contains the local variable. In Boxer, you can often just scan visually outward to find the superior box that contains the relevant variable.

We find that Boxer beginners often begin "accidentally" to structure code reasonably without instruction. For example, they find themselves wanting a subprocedure in the middle of writing the code for a superprocedure, so they just interrupt writing the main procedure and write the subprocedure right there, within the superprocedure. Accidentally doing the right thing is a great, facilitating effect of well-designed environments. Similarly, students automatically assume that different box-environments are independent. (Boxes can contain entire environments, and it is not uncommon to have several such box-environments in your Boxer world at the same time.) That is, they expect the procedures and variables in one box-environment to work independently of others. This happens to work just fine, even before we teach students how variables "scope."

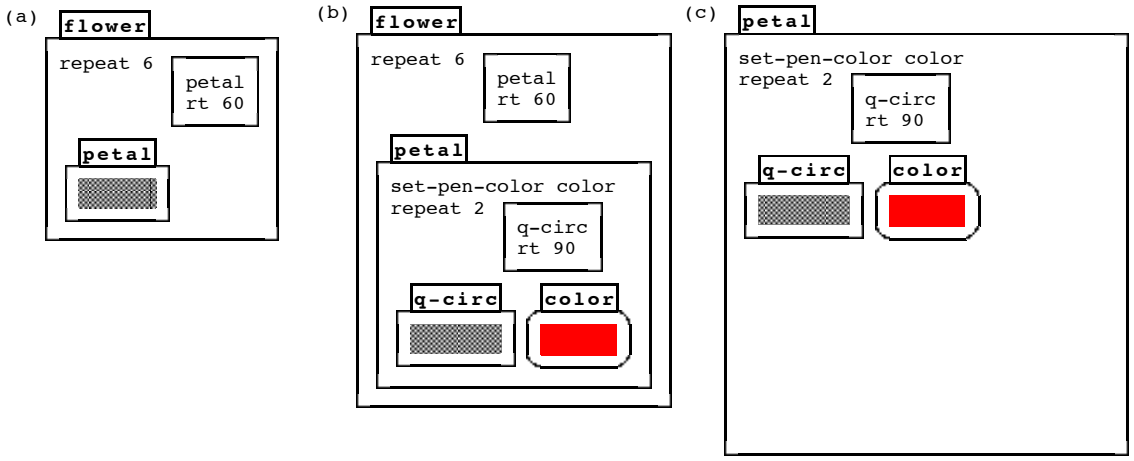


Figure 4. (a): A procedure contains a subprocedure, which is shrunken to hide its details. (b): Clicking on the subprocedure opens it to reveal its contents. (c): If the contents are complex, the user can click to expand the subprocedure to full-screen size. A click to shrink the box brings one back to (b).

Of course, accidental facilitations only go so far. To manage really large and complex programs, you have to do some work. Boxer provides several resources for effectively managing complexity. In the first instance, boxes within boxes are a great organizing feature that is automatically provided and easy to understand. In order to further control complexity and facilitate viewing a complex environment, the visual presentation of boxes is easy to manipulate. Any box can be shrunk to a small, gray box, hiding its contents and making more of the surround visible and easy to see without distraction. Consult Figure 4. Or, if a box is complicated and the student wants to focus his or her attention right there, the box may be expanded to full-screen size, effectively hiding the context. It takes a little practice to use these capabilities effectively, but, in our experience, they are easy to understand, and not much direct instruction is necessary.

In addition to controlling complexity by adjusting the visual presentation, Boxer allows a wide

---

[5] Here is a good little research project. How many Logo users even know about local variables? In Boxer, people begin using them even without instruction because the idea of putting a variable where it belongs is so obvious.

variety of ways to distribute code into semi-independent, meaningful units. For example, since visible graphical objects (Logo turtles or Boxer sprites) have a directly inspectable and modifiable box form, you can put code for behaviors that belong to that sprite right inside him. If you want a turtle to dance, you can (and probably should) put the **dance** procedure right inside him. Then, if he dances funny, you know where to look. I will discuss other methods of meaningfully distributing code to make programming and debugging easier in some other sections.

The final method that I will mention by which Boxer facilitates the construction of large programs also comes back to the fact that it has such a useful, spatial presence. In particular, in order to join two programs, or even two complete environments, the first step is usually trivial. Simply cut and paste the pieces so that they appear together, in the same place. Then, you can gradually integrate the procedures of the two parts so that they work together fluidly. I first noticed this process in the construction of the graphing adventure game by the two sixth grade students, mentioned above. One of them had a complete and working "video game" in a box, and wanted to make it part of the graphing adventure game. The first step was trivial. Just cut the video game box and paste it into the middle of the graphing adventure box. Gradually, the two young programmers integrated the code so that, for example, you could not enter the video game box before you managed successfully to complete some number of graphing challenges. They also added code so that your score on the video game affected what happened on following graphing challenges.[6]

4. Structure, Structure, Everywhere

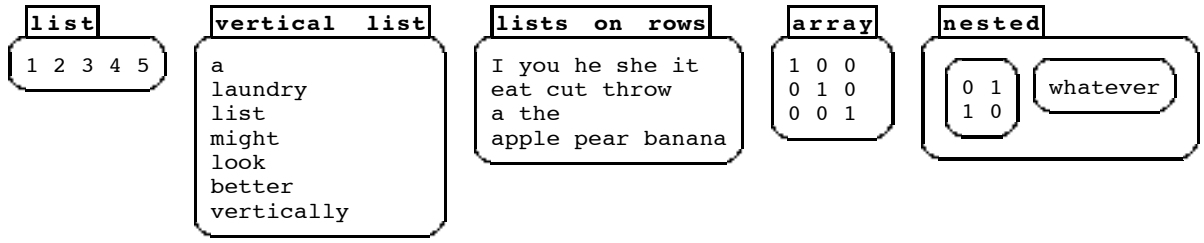
In the prior sections, I emphasized how the spatial metaphor and Boxer's principle of naive realism provide advantages for beginners and also for more advanced Boxer users. In this section, I want to show just a little bit of how these same features provide ease of construction of, and wide flexibility in, the data structures that one can develop in Boxer.

Logo's data structures are patterned on Lisp's, which in turn emerged from what is easy for computers to do. In particular, Lisp capitalized on the idea of a linked list, where each data item has a unique successor. The innovation of Lisp over other programming environments was that each element in the linked list could be an arbitrary object, for example, another list. And, one did not have to reserve a fixed amount of space associated with each object, for example, specifying in advance that you needed only a sequence of three parts in your list. Logo, essentially, made no changes in this structure.

Boxer, on the other hand, was driven by what is easy to see and manipulate with modern display technology. The basic data object is a box, containing arbitrary elements in a two-dimensional display, like a sheet of paper. The two-dimensionality of data boxes gives one tremendous flexibility in considering how you want to configure and think of your data. First, you can, of course, think of your data as a list, a sequence of items reading left to right, top to bottom. But you can display your list horizontally or vertically, or mixing the two by grouping a number of elements on each row. Or, you can consider your box of data to be an array, and use array indices to fetch or to change parts of it. Or, you can ask for elements of your box, or change them, one row or column at a time. Figure 5 shows some examples of different organizations of box data.

---

[6] More on this graphing game and how the students managed to accomplish such a complex programming feat appears in diSessa (1995a).




```
item 3 list  ||  item 3 vertical_list  ||  row 3 lists_on_rows  ||   row 1 array
                                                                     column 2 array
                    (element at Row 2, Column 3 of the array  -->)   RC 2 3 array
```

Figure 5. Some different arrangements of Boxer data and, below each, Boxer expressions selecting a part of the data.

Boxer allows much greater flexibility than Logo in terms of what kinds of things one can place inside data. In fact, Boxer places no restrictions whatever on what you can place in a data object. So, for example, Figure 6 is a record in a database that may easily be constructed just by making boxes, naming them, or cutting and pasting anything you can find in Boxer. The named subboxes can be addressed by name. For example, if this entry has been assigned to the variable **entry**, then **entry.last_name** provides access to the **last_name**. Notice also that pictures (graphics boxes) can be contained in this compound data object, and also colors (appearing as colors, not as a name or numeric code). Finally, procedures are also first-class objects, and my favorite fractal procedure can, like anything else, be placed in a compound data object.

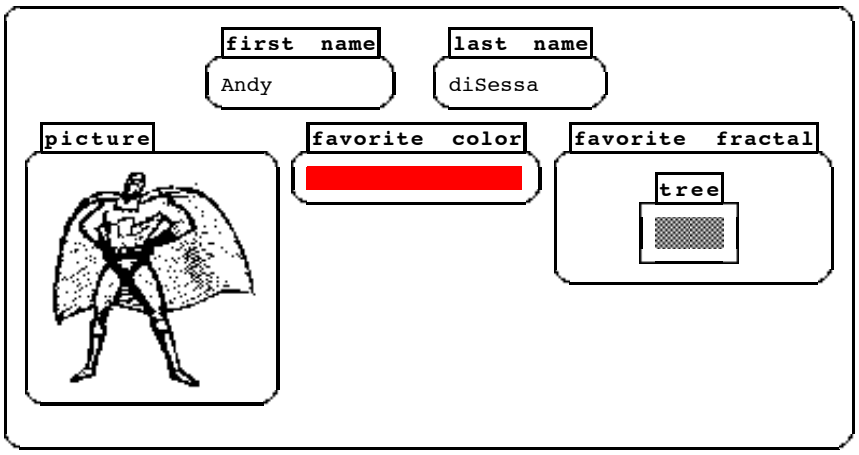


Figure 6. An entry in a database can contain any kind of Boxer data, including pictures and procedures.

5. A REAL Work Environment

A critical test for a life-long learning environment is not whether it is a nice place to visit, but whether you would like to live there. Boxer is designed to provide a flexible and practical work environment in which you can collect and integrate a set of tools and information to suit particular needs. Because Logo does not have the flexible structure of text anywhere, organized by boxes inside boxes, it lends itself more to a presentation environment for a single program. For example, in Boxer you can collect a number of box-tools (see reasons 8, 10 and 15, below) in the same place and surround them with working data. The tools can use and operate on surrounding data.

You can write notes to yourself about what you want to do, which is, what I call emblematically, "scribbling on the desktop." Indeed, because everything in Boxer is computational, you can write a little program to re-organize your "desktop" as easily as you could write a program to manipulate any other kind of data.

Let me give a couple of specific examples. The first relies on the fact that boxes saved as files can remain present in any Boxer environment. So, when you open a Boxer file, you might see a number of subfiles scattered through it. These subfiles initially appear as black boxes, and they read themselves in whenever you click on them to open them as you would any closed (shrunken) box. You could type whatever annotation you want around these files, and thus create a well-documented "directory" structure. The point of this example is that by combining generic Boxer structure of boxes and text, you can create very many kinds of organizations to suit particular purposes—in this case, a personalized file organization.[7]

A final example shows how the flexibility you get when every aspect of your environment is computational can pay off in surprising ways. During an early attempt to create on-line (Boxer) documentation for Boxer, the project coordination went somewhat awry. Multiple people created multiple versions documenting the same command or structure, and different people used different formats for the units of documentation. In order to straighten this out, I just collected all the pieces of documentation and dropped them into a single box. Then I used a little search program that I had on hand to collect related documentation elements so that I could select the best, or cut and paste best features. As I used a documentation element, I just deleted it from the box-database. In a similar manner, when I finished the complete, hierarchically organized documentation of Boxer, I wrote a simple program to prune out all the details, leaving a nice hierarchical index. Try to do either of these things in any ordinary programming environment, including in Logo.[8]

6. Build It Yourself

List processing is another area with which many people never achieve competence using Logo. This is a place where Boxer made a small innovation, which, nonetheless, has proved very valuable in eliminating a plateau in learning that was evident with Logo. Instead of a fairly complicated collection of ways of assembling and disassembling compound data objects, **`list`**, **`sentence`**, **`fput`**, **`lput`**, etc., Boxer has essentially just one command. **`Build`** takes a spatially-organized template as input, and creates an output of exactly the same form, except that every part of the template preceded by an **@** is replaced by the contents of the box that follows **@**, and, similarly, **!** means to insert the full box referred to at that point. Consult Figure 7. Removing two levels of impossibility (lack of graphical data, lack of two-dimensional structuring), the equivalent Logo expression is *still* very difficult to produce. Try it!

---

[7] A slight modification of this "directory" idea makes for a very useful organization for teachers. One can make a file box with an "auto read" property, so that it is read in automatically when another a box containing it is read in. So, a teacher can put a file box containing a set of tools inside each student's Boxer world. Whenever students read their worlds, they get the teacher's most recent set of tools. (When a box containing a file box is saved, the subfile box is not itself saved—only the "pointer" to the file.)

[8] The result of this work is, in fact, the on-line documentation of Boxer we supply with the current release. You can look to see how useful the hierarchical index is, and recall that it was generated from the full documentation by a simple program (which appears elsewhere in the set of Boxer demonstrations).

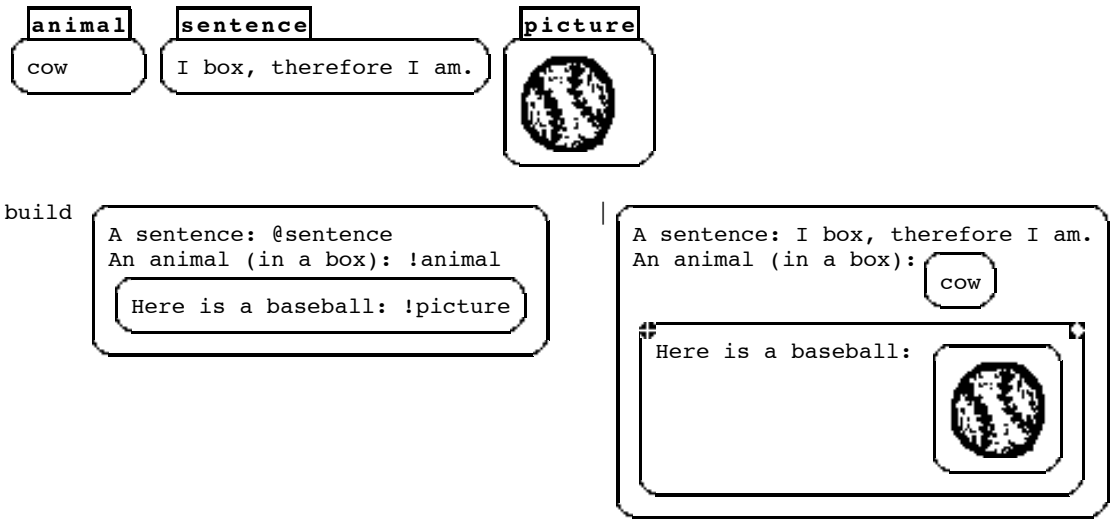


Figure 7. Top: A set of variables.
Bottom: **Build** takes a spatial template and fills in parts marked with **!** And **@**.

Build is one of the areas that has had independent and convincing empirical study. A study by two European researchers (Schweiker & Muthig, 1986) showed that learners achieved competence with **build** about three times as quickly has the equivalent constructs in Logo. And, after achieving competence, subjects were about three times as fast to create fairly complex objects, and to debug faulty expressions. This demonstrates, once again the power of visual, concrete structures, as opposed to invisible processes that you have to imagine.

7. Port Yourself to Infinity

Let's return to some parts of Boxer that were designed for more mature users, rather than for beginners. A *port* is quite similar to a regular box, except that its insides are identical to some other box, called the port's *target*. If you change either the port or its target in any way (editing or via a program), both are instantly changed. So a port provides access to a box that might exist remotely from the port, say, deep inside a complex program.

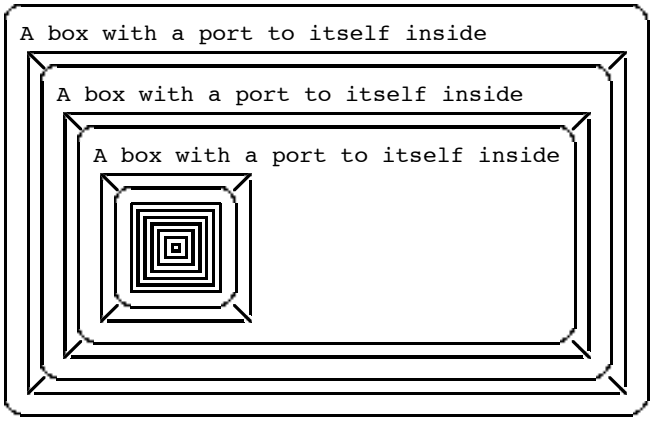


Figure 8. A box containing a port to itself is an infinite structure.

Let me start with a little "parlor trick" one can perform in Boxer, which, nonetheless, suggests the power of ports. Suppose you have a box containing a port to the box, itself, that contains the port. What you see inside the port will be the box, which contains the port, which contains the .... This is an infinite structure (Figure 8) that can be created in Boxer with a few keystrokes and mouse clicks. You can even "climb down" far into the infinite structure, in case you believe it is just a pretty picture.

Ports make great user interface devices, in addition to their numerous programming uses. For example, as suggested above, you can provide easy access (both for viewing and for changing) to variables or programs whose natural place might be otherwise buried deeply within the box hierarchy. Ports also make excellent hypertext links. You can directly see and use (or change) things that don't exist locally. Boxer *is* a hypertext environment, the first hyperprogramming environment, a colleague once said. One of the built-in Boxer uses of ports is that error messages, when possible, include a port to the offending procedure. That means you can frequently correct errors like misspelling or missing inputs immediately, without flipping the page or wandering around to find the procedure's definition.

Ports implement data sharing in a natural way. For example, if you want to have a database where two people share the same phone number (and changing one should change both), then ports are the right thing.

Unsophisticated programmers can skip to the next section at this point. This is for relative experts: Here is a somewhat esoteric but extremely powerful use of ports. If you name a port to a procedure instead of the procedure itself, then you get the effects of "lexical scoping." That is, when you use the procedure by name, any variable names used inside that procedure refer to variables accessible from the location of procedure definition, rather than from the place the procedure is executed. Lexical scoping in this way is a natural consequence of the meaning of ports (which I won't explain in detail). If you don't know about the great debates about the merits of lexical versus dynamic scoping (dynamic scoping is the usual way Boxer scopes, and the only way Logo scopes), suffice it to say that both have their advantages in different situations. In particular, lexical scoping is superior in that it always does the same thing, in contrast to dynamic scoping, which accesses one variable or another depending on where the procedure that contains the variable happens to be called.

Finally, ports implement "object access" in Boxer. Logo really doesn't have objects at all in that you cannot tell the difference between a data object and a copy of that data object. If you don't know much about these issues but wish to understand their importance, consult Abelson & Sussman (1985).

8. Beg, Borrow or Steal

Boxer was not designed, particularly, to be a collaborative environment. But I have been surprised to see how much better collaboration has gone in Boxer, compared to my experience with Logo. During the several years Hal Abelson and I ran an NSF-sponsored summer program for bright high school students, one of the constant difficulties we had was that, typically, one student always took over the programming for a collaborative project. Other students became very dependent on the "programmer" of the group, since they couldn't even use the program very well without him.

Our experience with Boxer has been the opposite. Again and again we have seen students, even students of very different programming capabilities, working extremely well together. For example, we have "exit video tapes" of students from a summer program course in Boxer where

each student is interviewed about the project. Even quiet students produced excellent explanations of how the projects were programmed, and, frequently, they would stop in mid-stream to make a small edit or two to improve performance.

In retrospect, some of this improved collaboration comes from evident differences between Boxer and Logo—in fact, from differences I have already discussed. When a program is easily inspectable, pokable and changeable, it is much easier to share. Anything you miss in the construction can be made up by playing with the code. Reviewing video tapes of students collaborating reveals another very important mode of keeping everyone in sync. Boxer's very rich visual presence on the display screen means that it is easy for students to configure the screen in order to point and explain what a program does. "Look, this says to increase the variable **x** each time. Watch **x** while I execute that line. See, this procedure calls that one; here, let me open (expand) it."

9. Reconstructible Interface

This set of features is for more advanced users, or for teachers and developers to prepare easy-to-use environments in Boxer for less sophisticated students. Logo has innovated, just a little, with respect to allowing the user to customize and reconfigure its own user interface. For example, you can create buttons and sliders to begin activation of a program or control a variable. Boxer takes that a step further. Rather than providing a small set of canned elements, Boxer provides resources to *create* these sort of things. For example, you can re-define what it means to click the mouse button anywhere in Boxer. You can define what a mouse click means on a particular sprite, on a particular graphics box, in a particular box, or on all sprites and graphics boxes, etc. You can similarly define what keystrokes do—either everywhere, or in a particular place.

The mechanism for doing this is really very simple. Every user interface action executes a procedure of a certain name. Clicking the mouse in a box executes **mouse-click**. Clicking on a graphics box or on a sprite executes **mouse-click-on-graphics**, or **mouse-click-on-sprite**. If you define a new procedure by that name, then it will be executed instead of the default Boxer action within the box in which you define that new procedure (and also in all subordinate boxes that don't have their own versions of that command).

To make a box that serves as a button, you just define a **mouse-click** procedure in that box. Another simple feature of Boxer allows you to make your button "pretty." If you define a graphics box called **boxtop**, then when you shrink the box containing **boxtop**, it appears as the graphic contained in **boxtop**. So, a pretty button is nothing more than a shrunken box with a **mouse-click** procedure and a **boxtop** image inside it.

These resources make it easy to create many kinds of interface objects, in addition to buttons and sliders. One of the advantages is these are all easily inspectable (to learn how they work) and, of course, changeable to suit your particular purposes. Boxer comes with some sample objects, including buttons, sliders, pulldown menus and "clickers." Clickers, in fact, are one of my favorite interface objects. They look like ● (on) or like ○ (off). If you click on them, they reverse their state. You can distribute clickers in the midst of code in order to allow easy turning on or off particular segments. I use clickers all the time as part of the user interface of microworlds and tools to turn various options on or off. Mixing code, data, and interface objects like clickers is a hallmark of Boxer and impossible in Logo.

10. First Class, Interactive Objects

The generalization of clickers turns out to be one of the most powerful kinds of objects in Boxer. You can make interactive objects that respond to mouse clicks and other interface actions, and

which have the following properties:

(a) They have all their "works" inside, so anyone can cut, copy and paste them anywhere work needs to be done.

(b) For the same reason, they can be opened to inspect them to see how they work, or to modify and extend them.

(c) They can be used as part of a program; just put one in the midst of code in an appropriate place.

(d) Similarly they can be used as inputs to procedures, or they can be created as outputs from procedures. The latter makes it easy to create procedures (which I call "factories") that create specialized interactive objects and return them directly to users to be cut and pasted where they are needed. For example, you can make a button factory that returns a fully functioning button to your specifications.[9]

Objects that can be placed in a data structure and can be used as inputs and outputs of procedures are called "first class" objects in the parlance of computer science. Boxer is almost alone among computer languages in allowing first class objects that are both graphical and interactive, in addition to fully functional in the language. Logo has no means to redefine interaction, and graphical objects can't exist in data or as inputs and outputs of procedures.

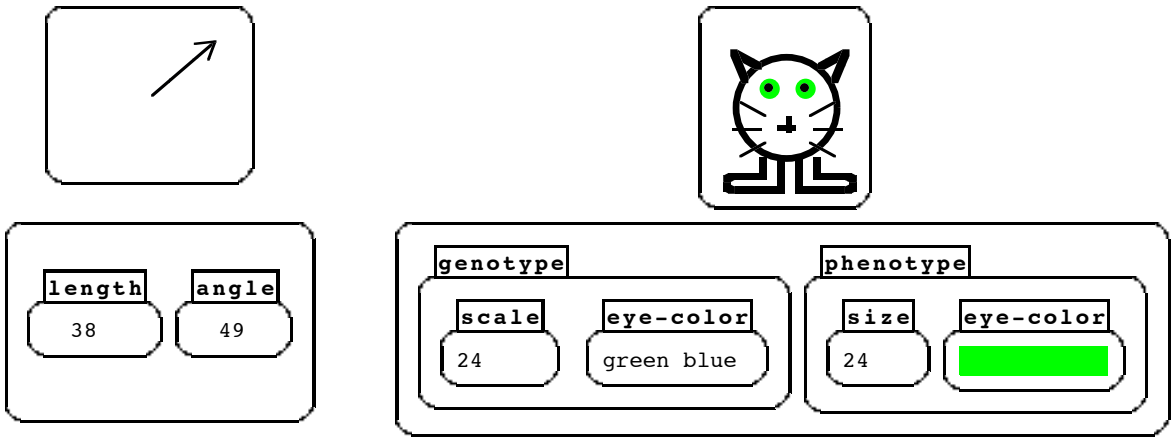


Figure 9. A vector and its flip side (left); a "scat" and its flip side (right).

Figure 9 shows two interesting interactive Boxer objects. The first is a vector that you can control by dragging the arrow tip of its graphical presentation around. On the inside of the vector are its coordinates, which you can set by hand or by program. Vectors can be added, as in **`add vector1 vector2`**, which returns the sum vector. (If you don't understand vectors, for now just realize they are powerful quantities representing things like force, velocity and acceleration in physics.) In addition, you can command sprites to move with the speed represented by a vector, and simultaneously you can change the vector to see how that affects the motion of the sprite. One of the nice things about Boxer vectors is that they are so useful and so much fun in creating and controlling motion that children spend a lot of time programming with them. This leads to a lot of learning of things that are usually considered "advanced." Students in one sixth grade experimental class we ran learned about vectors and motion in this way (diSessa, 1995b).

---

[9] See the button factory in the Boxer demonstrations included with the current Boxer release.

The second graphical, interactive object in Figure 9 is a creature called a scat. The insides of the scat include a representation of its genotype along with a computational version of its phenotype. Students can play with changing the genetic characteristics of the scat, and teachers or students can develop simple programs to experiment with breeding scats.

11. Can Your Turtle Do This?

Boxer turtles (sprites) have a number of advantages over Logo turtles, some of which are already apparent, above.

(a) In addition to all the usual attributes—shape, x and y coordinates, heading, pen-width and pen-color—Boxer turtles have a couple of other handy attributes. These include an overall size parameter and a home position where the sprite goes when you clear the graphics box in which the sprite resides.

(b) All those attributes are directly inspectable if you "flip" the graphics box to see its "logical" rather than graphical side. This is just the principle of naive realism. If there is a computational structure, you should be able to see and modify it. Of course, you can name sprites in the same way you name all boxes.

(c) You can put as many sprites as you want in a graphics box. You can even have a program add new sprites.

(d) You can easily add new procedures or attributes to any sprite or collection of sprites. In Logo, there is no logical (as opposed to graphical) representation of turtles, so this is impossible or, at minimum, awkward.

(e) Sprites are sensitive to mouse clicks, as explained above, so you can define their behavior when clicked.

12. Skeletons in the Closet?

Because Boxer makes things so visible and present, we have had to be somewhat inventive about allowing people to put things out of sight when they don't want to see them. One of the chief ways of doing this is with closets, which are part of every box in Boxer. Closets may be opened or closed at will. For example, when you look at a sprite (in its logical presentation), you see usually only the most-used attributes. But in the closet of the sprite, you can find all the other attributes. Similarly, if you look back at vectors and scats, you see only the most necessary parts. The rest is accessible in their closets.

A closet is an excellent place to hide the works of a microworld that users will ordinarily not need to see or change. Closets are a good place also to hide things like boxtops and key- and mouse-redefinitions for particular boxes.

13. Object-oriented Programming

Boxer allows other paradigms of programming, in addition to the usual procedural paradigm supported by Logo. This and the next section very briefly treat two other paradigms.

In Logo, you can **`ask`** a turtle to do something. But, it's awkward, at best, to teach a particular turtle new tricks, and very difficult to add new attributes to it. Most distressing, turtles are about the only thing you can **`ask`**. In Boxer, you can ask any data box to do things, and, of course, you can fill that box with whatever local data and procedures you find convenient. Thus, Boxer gets most of the important features of object-oriented languages like Smalltalk and Object Logo, but in a

very concrete, spatial-visible way.[10] Object-oriented programming, using objects and messages, has a number of advantages over plain procedural programming. First, it is a better, more modular organization of data and procedures to have meaningful chunks of them grouped together. It leads to systems that are easier to understand, and systems that are easy to extend, even if you don't understand everything about them. As important, objects are the natural way to think about and model many physical situations. Creatures running around a graphical display (sprites!) are one notable example, but there are many others.

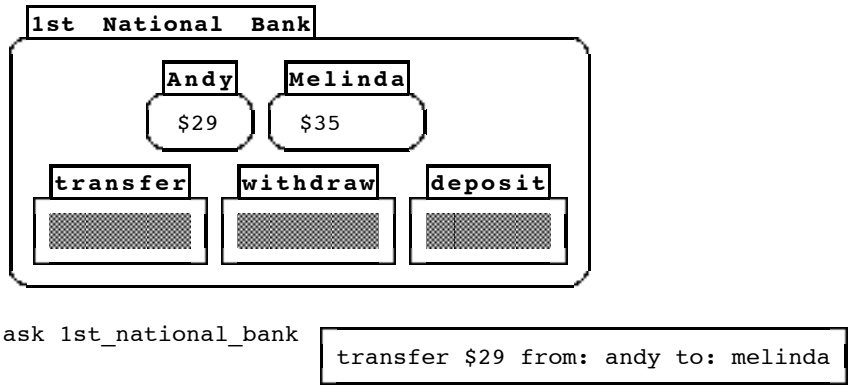


Figure 10. A bank has local data, knows how to do things like **transfer**, and can be asked to perform these functions.

Figure 10 shows part of a bank object. Banks, of course, know how to do things like deposit, withdraw, and transfer among accounts.

14. Activation-oriented Programming

Having a program automatically executed on certain conditions is a very useful way to program. You don't have to explicitly write the action and its conditions into other code that may have nothing much to do with it. For example, you may want to update some display any time a particular variable is changed. The display may just be a graphical presentation of the variable, say, a thermometer that shows a **temperature** variable. You don't want to have to put the graphic-changing procedure in every possible place where the temperature variable may be changed. Another example of activation orientation is a spreadsheet. If you change a cell, the recompute function is automatically executed by the very fact of that change.

Boxer has several activation triggers. You can set a trigger to execute when any box is changed by the user directly, or by a program. Similarly, you can set triggers to execute whenever you enter or leave a particular box. Activation-oriented programming is sometimes tricky because you can easily generated surprising and unintended chains of triggering. That's the negative side of the simplicity of saying "anytime this happens, also do something else." But, sometimes it is exactly the right thing, for example: with a spreadsheet-like uses of boxes; to keep logical and graphical representations in sync; or to initiate actions, say, starting a microworld box program executing

[10] When I first programmed in Smalltalk, I was terribly distressed to find out that objects, including graphical objects, don't "live" in a particular place the way they do in Boxer. But, Smalltalk didn't have either the principle of naive realism or the spatial metaphor to make its objects easily comprehensible.

when you enter it.[11]

15. Tool Building and Sharing

Building on reasons 8 and 10, Boxer makes an excellent environment in which a community can build a flexible set of tools to share with one another. As developers, my group of graduate students and I have experienced the kind of tool sharing that I never experienced with Logo. One of the seemingly little—but, after the fact, important—features of Boxer is that tools may exist just as a single box, with all the "works" inside to inspect, modify and extend. So, to start, you can just copy the box and use it directly; then, as you become familiar with it, you can open it up to modify and extend. One of the best kept secrets about tool-sharing in electronic environments is that nobody ever wants to use exactly the same tool as anybody else. If it's not inspectable and modifiable, it isn't much good.

Vectors turned out to be a marvelous general resource for our group when we designed several editions of a physics course. First, it is completely trivial to write simple tutorials that show how vectors work using working vectors! Second, vectors are great tools to build other tools and exercise microworlds. One of many tools we built in very short order with vectors was a analysis tool where students analyzed scanned stroboscopic images of balls flying through the air to find out things like whether their horizontal speed decreased (as most students expect) and how the vertical speed behaves. I already mentioned how vectors served as a tool for student projects.

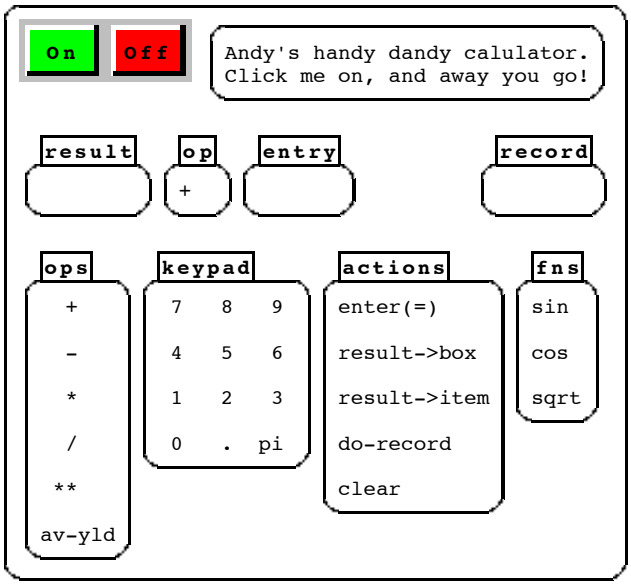


Figure 11. A simple Boxer tool—a calculator. See text for description.

Figure 11 shows a more familiar tool. It is just a little calculator written in Boxer. However, unlike a regular calculator (or a computer calculator!) it has the following Boxer-like characteristics.

(a) It is visible. You can see the "internal" registers for both numbers (**result**, **entry**) and for the operation (**op**).

---

[11] Sprite attributes work in this precise way! In the closet of each sprite attribute there is a modified-trigger that changes the visual presentation of the sprite when you change that attribute by editing it, or under program control.

(b) It is easily modifiable. You can just edit the **keypad**; it's just text. And you can easily add any functions (to **ops** and **fns**) that happen to be useful to you.

(c) It has "permeable boundaries." You can click on numbers outside the calculator, as well using the **keypad** to enter them. And, you can "export" numbers to the surrounding Boxer environment (the **result->box**, etc., commands). There is no need to worry about limits in the number of registers!

(d) It keeps a runnable and editable program (**record**) as a written history of the actions you perform.

(e) It is "scriptable." That is, you can **ask** it to perform any of its available operations. (Well, maybe a scriptable calculator is computational overkill. But, to have a graphing tool that you can call upon to draw a graph and return it to you is a better example of the power of using Boxer tools as objects to **ask**.)

More details about Boxer tools and toolsets appear in diSessa (1997).

16. Completely Integrated Mail and Network Browsing

You can send and receive mail from within Boxer. You can transparently send and receive not only text, but any Boxer object, such as a microworld. Consider how nice it would be to sort mail in various boxes (with ports in one place to all your important mail), or to be able to write a simple program to sort mail for you. Non-Boxer file attachments or Web URLs appear as icons inside Boxer that you can double-click on to open.

Boxer also has net boxes. These are almost exactly like the file boxes described in reason 5. That is, they appear as black boxes, or as their iconic **boxtop**, if one is defined. Then, when you click on them or use them in any other way, their "insides" are read in over the network. Net boxes, like any Boxer objects, have full integration with the rest of the Boxer environment. This means:

(a) Boxer's familiar and easy-to-use browsing capabilities also browse net boxes. The hierarchical structure means it is easier to keep track of where you are.

(b) You can put net boxes wherever you wish in the Boxer environment. Your whole world is your browser, and you can even write programs to manage your net boxes, if you like.

(c) Anything you make can be a net box or appear in a net box—programs, tools, complete environments. Put a trigger in a box if you want a moving graphic. (Java "applets" are a great idea—unless *you* want to program them.) There is no conversion necessary to put something up on the Boxer Web.

17. New Social Modes of Materials Development

This section is unlike others in that it is not a demonstrated superiority of Boxer over Logo. Instead, it represents a hope. It represents a hope that the cumulative effect of multiple advancements may change the very presumptions about who makes what for whom in educational computing.

Building particularly on 15 and 16 (which, in turn, build on other Boxer properties), I have gotten enthusiastic recently about opening the development of educational software more seriously to teachers. In particular, I would like to engage a community of likeminded people, including

teachers and students, in the construction of physics materials for learning. Unlike almost all prior software, this will be a flexible toolset that supports a wide range of ways to use it. Of course, the toolset will come with pre-made materials and activities, but the fundamental idea is that it is always open to innovation and change. So many times in our experience, the real brilliance of a tool was in a little change a teacher made to suit her or her students, or a new idea about how to use an old, familiar tool. Logo might have accomplished this, but it made it too hard for teachers to learn to use and modify tools. It makes tools that are brittle and isolated, hard to combine. Boxer's inspectability, pokability and management of complexity may just cross a threshold that allows, not only open materials, but an open process of creating and experimenting with tools. It takes a long time and a lot of experience to create a good tool or toolset. As a developer, I need serious help.

I imagine starting a smallish collaborative of folks using, commenting on, and modifying some of the tools we have built ourselves to teach physics. A Boxer activity database should evolve. Imagine that each tool has built-in net-box links to the activity database, and to the current discussions of core collaborators. I happen to think a book, or several, will also be necessary to support practical use.

Can this happen? Can it help professionalize, empower and re-energize teachers? Will we really be able to muster the effort to build curriculum that is simultaneously effective by any measure, usable by real teachers, and also true to the open learning principles that inspired Logo? Ask me in a couple of years.

18. But What About ...?

This section has some brief comments on some special things Logo (or MicroWorlds) has that you may think you can't live without.

... a draw program: Boxer provides all the necessary "hooks" to write your own draw program. There are two interesting, if very simple examples distributed with current Boxer. As the Boxer community expands—it has barely begun to mature—you can expect very many general tools like this to be produced and to become available.

... buttons and sliders: Again, check the Boxer demonstration files. You may find you can't live without Boxer clickers, pulldown menus and the many other kinds of user interface objects that are constructible in Boxer.

... QuickTime movies: Audio-video boxes should be coming soon, if they are not already available. Upgrade your old Boxer.

... parallel processing: In many instances, it is better to retain control of parallel processes in order to make sure things run in synchrony. For example, we frequently write programs with multiple sprites (or other objects) that are all driven by a **`tick`** messages sent from a single controller. Boxer has its own parallel processing system, but we have not found it useful enough to debug it thoroughly. If you need thousands of turtles and just can't do with only scores, stay with StarLogo for now.

Are there any things I envy about current Logos? Sure. They are pretty, slick and require less memory than Boxer. These are things that only commercial efforts can manage.

19. Upgrade Your Skills

It's fairly easy to start doing Boxer if you know Logo. We have been careful not to change things gratuitously just to be different. But be warned: It may be easy to fall into the trap of thinking Boxer is mostly just like Logo, and to use it to program just like Logo. As many of the items and examples above should have demonstrated, that would be a serious under-use of Boxer's power.

Constructs like ports, object- an activation-oriented programming, etc., can make things that are difficult in Logo much easier.

20. It's Free!

See:

http://www.soe.berkeley.edu/boxer

or send mail to boxer-inquiry@soe.berkeley.edu.

Conclusion

Logo began with a grand image of the computer transforming learning from an often painful, alienating and awkward process into a more natural-feeling and empowering one. But, I believe Logo tripped by not realizing its transitional nature—born of teletypes and printers. Instead, it pursued glitz and contemporary-looking features rather than changing its infrastructure. This has left users able to do nice-looking things quickly, but without the kind of deep penetration into learning cultures that we need. With Boxer, we started from scratch designing an environment that uses display technology to make things easier to do and easier to understand. This paper invites you to explore new possibilities. I have tried to explain what I believe to be advances of Boxer, and why these could make a big difference.

References

Abelson. H. & Sussman, J. (1985). *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press.

diSessa, A. A. (1995a). Collaborating via Boxer. In L. Burton and B. Jaworski (Eds.), *Technology—A Bridge between Teaching and Learning Mathematics*. Bromley, Kent , UK: Chartwell-Bratt, 69-94.

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

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.

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

Schweiker, H., & Muthig, K. (1986). Solving interpolation problems in Logo and Boxer. In P. Gorny & M. J. Tauber (Eds.), *Visual Aids in Programming*. Heidelberg: Springer-Verlag.