Boxer Profile:
Component Computing within a Computational Medium

Andrea A. diSessa

The Boxer Project
University of California, Berkeley

Overview and History

Boxer is not a "component computing" project in a narrow sense. Instead, the Project has nurtured a long-term, broad and well-articulated position on the use of computers in education (diSessa, 2000), which is independent of components per se. However, in the last 5 years or so, a fair proportion of Boxer work has moved toward a component basis; the group has developed experience and a point of view, within its larger educational frame, concerning component computing in education. While Boxer is not currently and hasn't been funded to do component work, its technological basis is probably the most distinctive of the projects we profile; the distinctive technological basis provides useful contrasts, and it highlights difficult, potentially important judgments and choices in how to pursue the advantages of component computing. In addition, Boxer's philosophical position also highlights some choices of orientation within the spectrum of possible component-computing systems.

The Boxer Project developed out of the pioneering work of the MIT Logo Group in the 70s and 80s. Logo presented a stark contrast to the Computer Aided Instruction most popular at the time. With Logo, students learned mathematics[1] by programming, rather than traditional exercise-and-test methods. The Logo Group originated the "microworld" methodology for learning with computers, in which students explore interesting, rich environments, which are milked by teachers and instructional designers for their mathematical content.

The Boxer Project branched off from Logo in two basic ways. First, in philosophy of technology use, it moved closer to a view of computational resources as a general "medium of expression," rather than concentrating on a narrower focus, on programming. Viewing the computer as a medium means that it should be the basis for the broadest possible range of uses within education, including menial usefulness in everyday tasks, as well as scientific and other forms of expression. The rhetoric of the Boxer Project highlights the notion of *computational literacy*, in which everyone—students, teachers, curriculum developers—can compose, as well as use products developed by others, in a common medium.

[1] Although work in other disciplines took place, mathematics was a primary focus in the Logo community.

The second distinction from Logo was in technology per se, to align the design of the system to better support the "literacy with a medium" image. In particular, Boxer was designed from scratch to support both easier learning and a greater breadth of use, including much more utility in performing everyday tasks. Boxer's technology, a single, completely integrated system to support all forms of educational use (including, but not limited to, programming) provides a contrast with the other systems we profile here. In addition, Boxer introduced component-facilitating features that Logo, and almost all other programming environments, don't have. These feature make construction and use of components—screen entities that one moves around, copies, combines and modifies to produce new software—very easy.

To preview Boxer's characteristics of a component system, we note that it provides the most flexible container environment of any component system represented in our set of profiles. A container environment defines the "screen space" and computational interconnection possibilities of a component system.[2] The most popular container environment currently in use is the web browser so that, for example, visually placing components may be done using HTML. Boxer's container environment is a principle part of the system itself, in contrast to, for example, ESCOT, where the screen container (browser) is quite distinct from the system of constructing components (Java plus interconnect protocols). Boxer's container is very flexible, constituting a full hypertext editor, and the environment is tuned toward construction and reconstruction, rather than just use. Few users of web browsers edit HTML; essentially all users of Boxer build from scratch, or at least modify the presented organization of Boxer learning materials. For example, moving a component in Boxer can be accomplished with nothing more than a cut and paste.

The internal structure of components in Boxer involves precisely the same structures and protocols as the container environment. In contrast, both E-Slate and ESCOT distinguish the internal structure of components (Java) from their screen-space container environment. Boxer is constructed mainly of boxes containing text and more boxes; individual components and systems composed of components are identical in this respect.[3] That means that users can easily move into the mode of authors, or at least "tinkerers and tailors" by opening components (which are simply boxes that are ordinarily closed), or by moving components around. Easy shifting from "user" to "developer" is promoted not only by an interface that doesn't distinguish between users and developers, but by the fact that ordinary use of the system provides experience with the same structures that one needs in order to explore, change and construct any object in Boxer.[4]

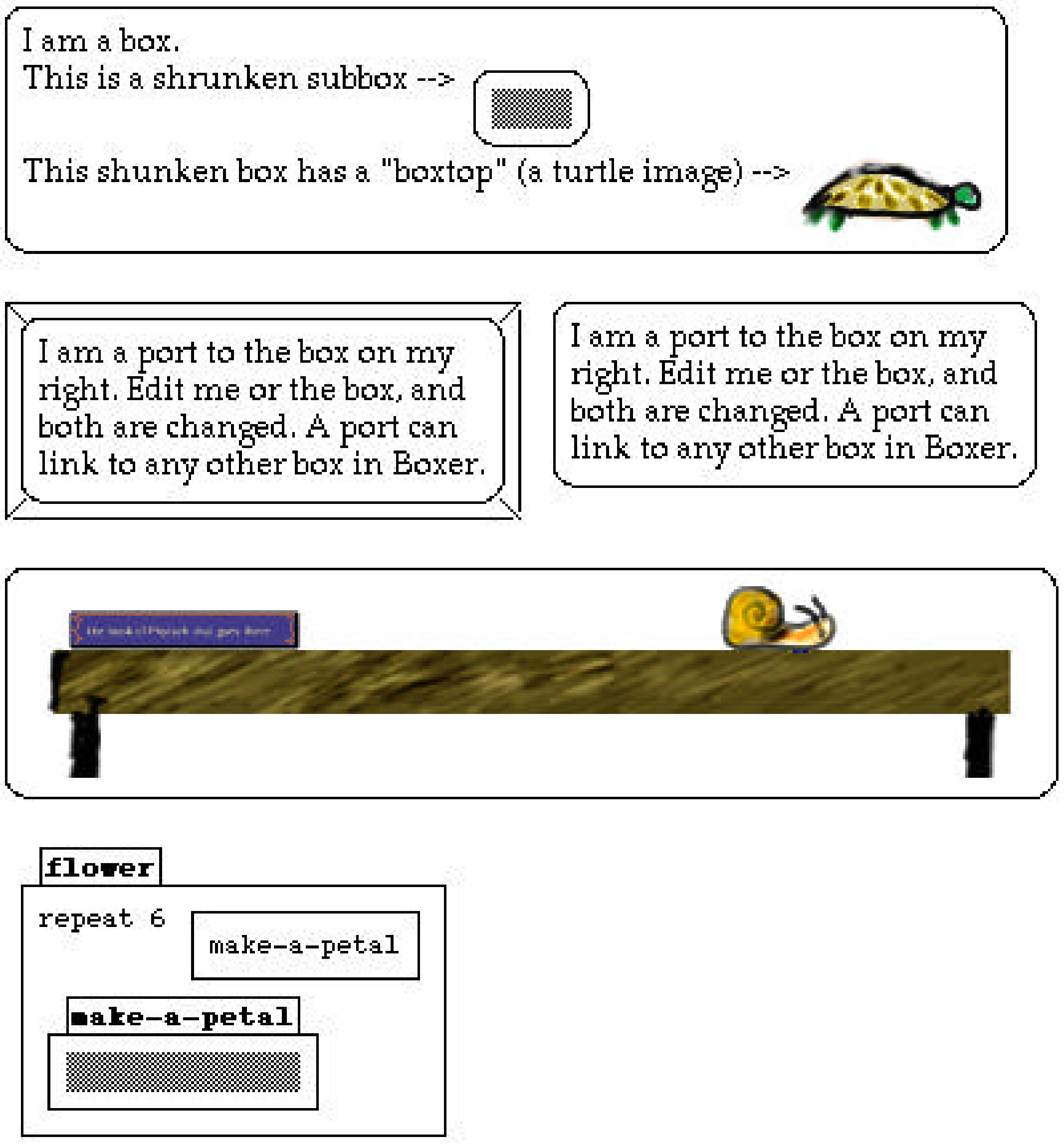
[2] In principle, the "screen-space" container and the programming interconnection container may be distinct. This is strongly true for ESCOT, where interconnections are at the Java level. E-Slate, in contrast, uses a proprietary container with a built-in interconnect interface (plugs and sockets) that browsers (including the ESCOT Runner) don't have.
[3] Since components and the internal structure of components are identical, hierarchical composition is automatically possible in Boxer. See the discussion around Figure 3. ESCOT does not have hierarchical composition capabilities, and E-Slate plans to add them at some future time.
[4] Having the *use* environment be the same as the *construction* environment also means the boundary between component and the rest of the environment is very fluid. A part of a learning configuration of boxes can be placed in its own box to become a component; a component can be opened and pieces used

The features that make Boxer distinctive as a technological environment enable a broader range of social configurations of development and use. Primarily, users with much less technical competence can become involved in technical aspects (creation, modification) of component-based learning environments. A key question is how powerful those alternate social configurations are. Social configurations of development constitute a critical set of issues. We will raise some of those issues in this report, and they are and will be discussed in many other places in our Project's writings.

The Technology

We provide here somewhat more detail on technology, per se, than in other profiles. This is for several reasons. First, Boxer is designed to show its "basic workings" to ordinary users. Thus, Boxer technology should be more comprehensible than other construction or component assembly environments. In addition, some important modes of Boxer use depend on its technology, as we will show. Finally, Boxer is an unusual system that is likely less familiar even to technologically sophisticated readers.

separately. This fluidity has advantages in terms of flexibility, and, possibly, it has disadvantages in terms of "stability." (Users can very easily change the form and functioning of the system.)


I am a box.
This is a shrunken subbox -->

This shunken box has a "boxtop" (a turtle image) -->

I am a port to the box on my right. Edit me or the box, and both are changed. A port can link to any other box in Boxer.

I am a port to the box on my right. Edit me or the box, and both are changed. A port can link to any other box in Boxer.

```
flower
repeat 6 make-a-petal
    make-a-petal
```

Figure 1. From top to bottom: A box containing text and some shrunken subboxes; a “port” to a box to its right; a graphics box containing a picture of a table, a mobile book that can slide across the table and a snail trying to outrun the book; a program (doit box) named “flower,” with a shrunken subprocedure, “make-a-petal.” See text.

Boxer can be thought of, to first approximation, as an augmented hypertext editor. Consult Figure 1. Text has a large role in the system. Text can be structured by means of boxes, which can contain more text, or, recursively, more boxes. A special kind of box called a graphics box contains images, mobile graphical elements, and supports a re-definable “point-and-poke” interaction.[5] See the table (image) with book and snail (mobile graphical entities) in Figure 1. Graphics boxes are the most common face of Boxer components.

[5] By “point-and-poke” we mean an interface drive by purely graphical, mouse-oriented means, in contrast to Boxer’s broader augmented text processor interface.

Boxes can be shrunken to hide their contents, or expanded to fill the full screen if their contents are complex enough to warrant full focus of attention. Users can define or change the graphic (the “boxtop”) that appears in case a box is shrunken. Boxes can be cut, copied and pasted, just like text.

Graphics boxes have two different views. Each graphics box may be “flipped” to provide a conventional Boxer view of whatever text and boxes the creator chooses to put there, instead of the standard graphical face. We will illustrate uses of flipping graphics boxes later.

“Ports” are special boxes that provide two-way links to ordinary boxes. The port shows exactly the same thing as its “target” box, and changes to either the port or its target result in changes to both. The second row in Figure 1 consists of a port (left) and the box that is the target of the port (right). The text contained in either the port or its target can be edited, and the other element instantly shows the changes. Ports are the standard Boxer mechanism for creating hypertext, and they are also handy computational means for sharing and communicating information among components.

In addition to being augmented by boxes and ports, Boxer is augmented over an ordinary text processor by the fact that it is a programming environment. In fact, all of the boxes that are used to structure text (data boxes) are also computational objects. If a data box is given a name, it may be accessed and changed by a program or simple command, in whole or in part. In addition, text typed in Boxer may be executed (provided it consists of commands built in to Boxer or added by user or developer). Just as text (data) may be structured by data boxes, text that constitutes programming statements can be structured by “doit” boxes, which have the same user interface properties as data boxes (e.g., they can be expanded or shrunken).

Among Boxer’s features as a programming language, it allows message passing and object-oriented programming. Boxer can pass messages by TELLing any box to execute any command in its interior, where particular procedures and data may be available. Message passing, in addition to ports, is another way to connect and control components.

Components

Boxer has a fair number of components and other tools in a standard library, which is currently distributed with Boxer itself. Components are typically created as graphics boxes that have been designed to provide a particular graphical presentation, and to have interactive properties (mouse interaction) suitable for some function. Figure 2 shows an example of a canonical component, a graph drawing utility. When you press the mouse anywhere on the component’s surface, a pulldown menu appears to show options such as “graph data” (the user then clicks on a box containing numerical data), “clear,” and “draw-by-hand” (which allows the user to draw a graph and, if he or she chooses, to get the numerical data represented by the drawn graph). Since the grapher is simply a (graphics) box in Boxer, it may be cut, copied and pasted anywhere it is needed.

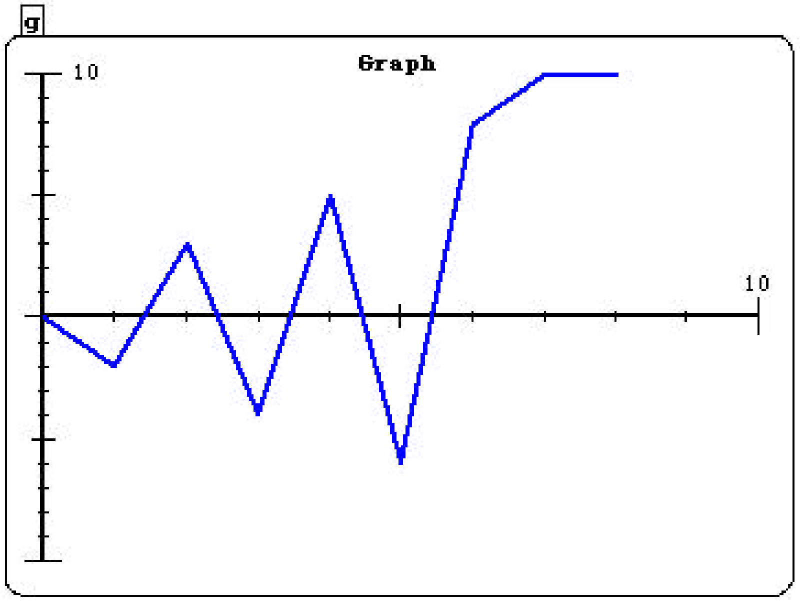
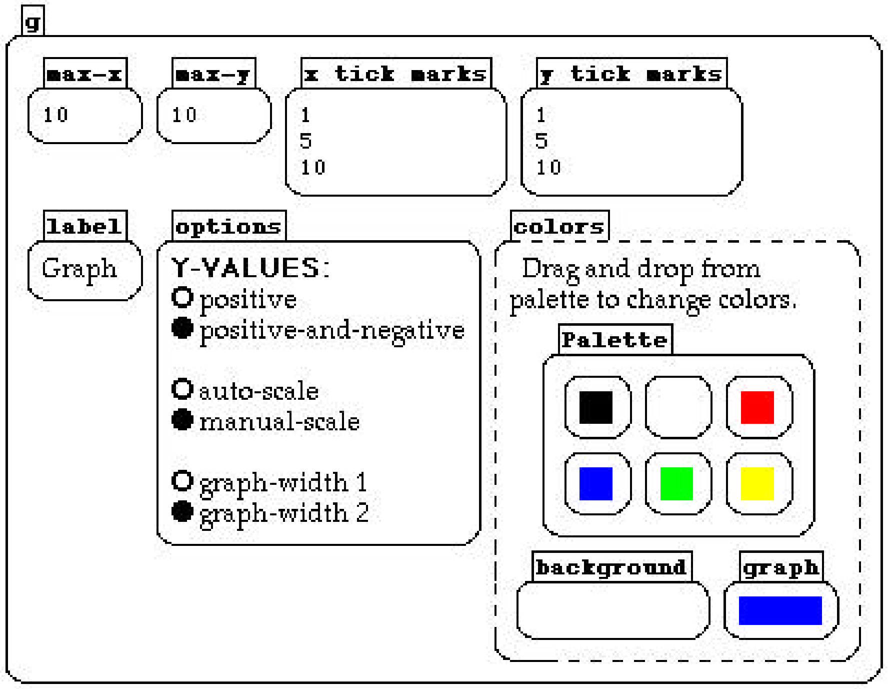


Figure 2. A graph drawing utility component.

Figure 3 shows the “flip” side of the grapher’s graphics box. Notice that a generic Boxer function that most users will already know—flipping a graphics box to reveal an alternate presentation—is a part of the way this component functions. In this case, and typically, the flip side is used to contain parameters to set: parameters like color and width of the graph, maximum values on axes; options such as the layout of the axes; and modes such as “automatically set the vertical scale.” The flip side may also contain documentation, etc.



max-x: 10
max-y: 10
x tick marks: 1 5 10
y tick marks: 1 5 10
label: Graph
options:
Y-VALUES:
○ positive
● positive-and-negative

○ auto-scale
● manual-scale

○ graph-width 1
● graph-width 2

colors:
Drag and drop from palette to change colors.
Palette: ⬛ ⬜ 🟥 🟦 🟩 🟨
background: ⬜
graph: 🟦

Figure 3. The “flipped” grapher reveals user-settable parameters.

Figure 3 also shows one of the advantages of Boxer as a container system. Components can easily be hierarchical. That is, the check-box "clickers" and the drag-and-drop color palette are, themselves, simple components that the developer of this grapher simply cut and pasted into it while constructing it. In addition, the pull-down menu used as part of the interface to the grapher is a very slightly modified copy of another component in the standard Boxer set of tools.[6]

The grapher in Figures 2 and 3 is scriptable. In fact, scripting capability is automatic for any Boxer object: Recall that any box, including this grapher, can be told to execute Boxer commands. The grapher responds to commands like "plot" (a single datum, so that real-time graphing of simulation parameters is possible), "plot-data" (plot a full dataset), "clear," etc. No special scripting commands are necessary in order to set parameters, as generic "change this variable" command does the trick.[7] One can, for example, execute

```
change label to: speed vs time
```
,

which accomplishes the creation of a new label for the graph. [The "Graph" label near the top of Figure 2 is controlled by the "label" box in Figure 3 (left-most item in the second row of boxes). "`Change label` …" changes both internal variable and external presentation.] One can similarly change the color of the graph in preparation for a new graph or an overlay graph.

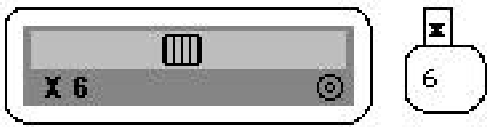


Figure 4. A slider component, which is connected to and adjusts a variable, X.

Figure 4 is another generic Boxer component, a slider. In this case, it controls the variable, X, to its right. One can press on the "target" icon (concentric circles on bottom right of the slider) and drag and drop to connect the slider to any other box. Just as with the grapher component, above, the flip side of the slider contains parameters that can be set directly or by scripting. The flip side of the slider contains parameters such as max and min values, and "action" boxes in case the slider needs to take some actions when its value is changed.

Connecting components

There are a very wide range of ways of connecting Boxer components.

Wiring: Drag-and-drop is a convenient interface for components that need explicit connection to other components. (For example, drag and drop is used in Figure 4 to connect a slider to the variable it controls. See also Figure 10 and immediately prior textual description.) It is also an interface that is easy to understand and enact for

[6] The box structure that defines the pulldown menu is hidden away in a special place we call the "closet." Every box has a closet. If one opened the closet of the grapher, the pulldown menu would be visible, and, as a user or developer, you could directly edit the names or actions that define the menu.
[7] There are many ways to hide data or commands so that casual users don't have access to low-level or critical elements of box/components.

teachers, students and developers who are not technologically sophisticated. The drag-and-drop interface typically will place a port to a component, or to a piece of it, inside another component. In that way, information available in one component is available to be observed or changed in the connected component.[8] Drag-and-drop is only one among many ways to "wire" components together; for example, a program can perform the same function as a drag-and-drop gesture, thus connecting components. Developers may permanently wire some components together, and their connections will be maintained on cutting and pasting, and on moving components around.

Scripting: Scripting, of course, is always available in Boxer, and it makes a useful mode of interconnecting components. Scripting typically involves Boxer's TELL command, to remotely execute procedures within a particular box. An overall program can script components, or components can script each other. A simulation can plot quantities in real time by scripting a grapher.

Inheritance: A third paradigm of component interconnection is that a component can simply make some of its own data or procedures available by name when it is pasted into an environment (i.e., into a box). For example, developers frequently package a set of capabilities and resources in a "tool box." If one pastes a particular tool box into another box, all of the resources in the tool box become available in the new context. An existing Boxer tool box adds the ability to create constrained geometric constructions (ala Sketchpad or Cabri) to graphics boxes. Another tool box adds the capability to control QuickTime movies and to create clickable index items, which start the movie at a particular time. A box that functions as a tool box may or may not have a graphical interface. For example, a grapher—which does have a graphical interface—can be configured to make available commands such as "plot" or "clear" to any other component within the same containing box.[9]

Components as first-class programming objects: Boxer is unique in allowing visible and mouse-able components to serve as first-class programming objects. This means that cut, copy, and paste-able objects with their own user interface can serve as inputs, outputs, or, indeed, as named data objects to be modified or inspected by programs as well as by user direct manipulation. In Figure 5, graphical vectors—where the vector "arrows" can be adjusted with the mouse—serve as inputs to the command "add." (The flip side of vectors shows component numbers.) The output of the sum appears as a new vector object, although one can also set a vector-variable (a named vector) to the result of sum. Interweaving of components and programming has been very important in educational work that teaches students by having them write simple programs. For example, students can learn about motion by constructing simulations or video games using the vector components. Students as young as sixth grade have used vectors in learning about physics (diSessa, 1997).

[8] Because Boxer can observe and modify all of its own structures, a drag-and-drop gesture can, for example, add completely new capabilities to a component, or change the capabilities already there.
[9] Fancier organizations of inheritance are possible. Any component that "expects" that certain resources may be available can check if they are, and if so, use them. So, for example, a simulation can check for the presence of a grapher, and if it is present, the simulation can graph whatever parameters it chooses. Of course, designers need to think about potential conflicts of multiple components using common resources. However, various forms of interconnection via inheritance can be completely transparent and effortless for users, at least in the best of circumstances.

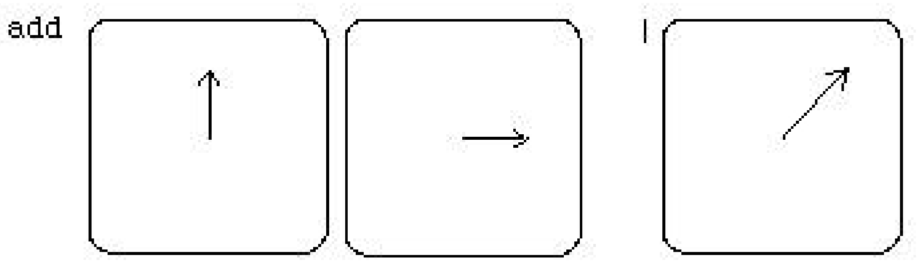


Figure 5. Mouse-able vectors serve as inputs to a programming command, “add,” and produce a new vector as output.

Another use of components as first-class programming objects is to build “factories” that create specialized components to suit. For example, an existing Boxer utility creates active push button components to specification. Another such factory can automatically build an interactive database inspection tool, specialized to particular purposes.

Modifying components

Boxer has a uniform structure and interface, both inside and outside components. As a consequence, any person, user or developer, can flip a graphics box component, manipulate what appears there, or delve into the “works” of the component. A responsible component developer will typically hide the parts of a component that won’t concern most users (e.g., in the box’s closet, see footnote 2), although elements of the “works” that are often changed may appear directly on the flip side of the box.

Boxer’s ability to provide modifiable components and modifiable configurations of components according to principles that most Boxer users will learn (cut and paste of familiar box-structure, simple programming, ports, etc.), moves the possibility to configure and modify components and component systems down the ladder of technological expertise.

Non-component resources

Because Boxer is a full-fledged programming system with rich generic resources, it offers a number of other paradigms of use, which can be mixed and matched with component-styled software.

Generic functionality: Because Boxer’s native environment constitutes a rich container, many generic capabilities are automatically provided to users and developers. For example, a student “notebook” may simply be a box into which users paste results of experiments (e.g., a graph) and type text explaining their work using always-available editing actions. Developers can structure the notebook to guide students, including, for example, suggested ordering of stages of work, hints and specific questions at each stage, etc. Students might create a full-blown hypertext report document, including analysis tools, datasets, and so on, mainly by cutting and pasting. Sending and receiving any box structure as mail is a built-in capability of the Boxer “container.” Therefore, a learning community of students, teachers and others can easily share their Boxer constructions and commentary about them. Since dynamic objects are as easily mailable as plain text, those who receive Boxer mail may review the report, but they can also re-analyze the data, or use tools included in the document in their own experiments.

A powerful capability that is very easy to build in Boxer is macro construction. For example, user interface actions in some educational environment can simply append their textual representations to a box. Then users can open that box, edit it, or “variablize” it (turn specific values into variables) for more general use as a procedure. High school students create geometric constructions by pointing and clicking, and then variablize the construction to create a general procedure, for example, to construct a line perpendicular to any given line. One advantage of Boxer with respect to typical macro systems is that most Boxer users learn at least simple programming, so macros in Boxer are familiar objects, which users know how to modify and use in conjunction with other resources.

“Libraries”: While components represent a powerful paradigm, some other paradigms that serve very nearly the same function can be easily mixed and matched with component computing. Libraries of specialized procedures have long been part of large-scale, modular programming systems. For example, one can build a library of specialized procedures that manipulate and display statistical data. Library capabilities might be combined into a component or multiple components, but they need not be. For convenience, a library in Boxer might be contained in a “tool box,” with or without graphical interface (as mentioned above with respect to connecting via inheritance).

As we introduced earlier and will discuss further below, many Boxer constructions are a combination of specialized components that work with each other, together with library resources such as programming commands that extend the usefulness of the components. The vector toolset, including graphical vector components extended by specialized programming commands, is a simple but excellent example. The toolset includes library commands to add vectors, to cause objects to move with a speed specified by a vector, etc.

Tools for developers of Boxer materials frequently include more specialized procedures and fewer graphical, user-interface-complete components.

Partial list of components and tools

Below is a partial list of the components and tools that are currently available in Boxer. They are available for download at http://soe.berkeley.edu/boxer

High-level components:
Grapher
Programmable calculator
Simple drawing tool
Simple spreadsheet
Database tool (to design and manipulate—e.g., query and filter—simple databases)
Documentation tool (e.g., to provide a guide or instructions to students)
A mail utility, so student and teachers can mail messages and Boxer structures in ways similar to commercial mailing applications
Palette and color making tools (see the image processing tool set, described below)

Graphical programming objects:
Vectors (graphics boxes with mouse-adjustable arrow; the abilities to add and scale vectors, to move objects along a vector, etc.)
Image boxes and palettes (extended-capability graphics boxes that have a mouse-based interface, yet can be inputs to, or outputs from, programming functions; see the discussion of the image processing toolset in the Case Studies section.)

Interface components:
Button "factory" (generates push buttons to specification)
Clickers (varieties of cut-and-paste "check-boxes" to suit various needs, such as radio-button check-boxes, check-boxes that can be cut and pasted into programs to turn selected parts on and off, etc.)
Pulldown menu
Slider controllers of various sorts
Color selector

Components and libraries for developers or advanced users:
A toolkit to create drag-and-drop interaction among components
A toolkit to introduce "click on any object for explanation" capability for unobtrusive help and documentation in any Boxer environment
Region and segment selection utilities. For example, a region selector tool might be used to implement the ability for users to select a region of a complex graphical presentation to zoom in to.

In Appendix A, we describe the construction of a learning microworld out of components in order to illustrate how easily this may be done. The microworld itself will be described in the next section (circa Figure 11), so readers may chose to postpone reading Appendix A until the context is better set.

Patterns of development, appropriation and use

In this section, we introduce a particular style of product and pattern of product development, which the Boxer group is exploring. To begin, here are some motivating concerns:

1. Generic components may not be adapted specifically enough to a particular kind of use (say, teaching a particular topic). Trying to make components sufficiently adaptable for all uses may fail either in not anticipating the need for a feature, or in making the generic components complicated and difficult to learn.
2. The creativity of teachers and students (and less technically skilled curriculum developers) is generally poorly served by current component technology. At best, very few teachers participate in (ESCOT-like) integration teams, and students have almost no creative role at all.
3. Adaptability of most component products, after "official" development ends, is minimal or non-existent.
4. "Development of technology" perhaps should be expanded to recognize that co-development of technology and teaching practices, over an extended period of

time, is a more reliable means of producing learning materials. Co-development with teachers is also an excellent means of professional development and enhancement.

Given these considerations, we propose:

1. In addition to libraries of general components, we expect to see the development of families of tools specifically designed to work together well in addressing the needs of the particular learning area. One grapher might well not work well in teaching both physics and algebra; so it might be better to produce a modified grapher specific to particular subjects or with features that work well with some particular set of other components. This is the "open toolset" idea (diSessa, 1997).
2. Configuring components, modifying them and even making components from scratch should be made accessible for the broadest range of people possible, including (respecting levels of technical competence) teachers, curriculum experts with less technological expertise, and students. For us, this implies a rich container environment, and component-construction and component-configuration technologies that are much simpler than most current practice.
3. It should be possible that groups of teachers can work on their own, with relatively little support, over an extended period of time to produce their own materials and practices of teaching with them. This does not at all mean starting from scratch. Developers may provide toolsets and expected-to-be useful configurations of the tools. But one would expect change and even the creation of new configurations in the hands of "users" like teachers and students.

Although many variations of development practices are possible respecting the considerations above, for concreteness, we specify one particular model, which we call the LaDDER model (Layered Distributed Development of Educational Resources). The LaDDER model involves 4 "layers" (students, teachers, local developers (or "secondary developers"), and global developers (or simply "developers"), all working together to develop learning materials over an extended period of time. More specifically, the product of such a co-development system would be (1) educational materials, (2) expert teachers who understand the materials well enough to explain and even modify and extend given materials, (3) well-developed practices of effectively using the materials, embodied in the work of teachers, (4) a toolset specialized to the curricular/activity focus in question. The toolset allows easy experimentation during development, and easy "disassembly" and modification after any "official" development has finished.

The characteristic pattern of work in the LaDDER model is that technical needs or problems propagate up the technical competence hierarchy (students to teachers, to local developers, to global developers) to the point that they can be addressed. Then, however, instead of solutions, new or modified *resources* are created and propagated as far as feasible back down the hierarchy before converting those resources into solutions. The core idea is to provide as flexible as possible resources to as many levels of the hierarchy as possible. That is why more technology-sophisticated individuals in the collaborative system should, preferentially, develop tools with which less sophisticated users can solve their own problems, rather than directly fulfilling technological needs.

Typically, the global developers (technology experts) will initiate construction of a toolset, and probably provide model configurations for educational use. Yet, a lot of work—modifying, trying out new configurations and activity structures—can happen among teachers, probably in conjunction with members of a helping level, what we call a local developer. A local developer might be, for example, a member of a university community with more technical expertise than a typical teacher, but more pedagogical expertise than a “global developer.” Or, a local developer might be a particularly technology-expert teacher. One function of the local developer level of the hierarchy is to provide leadership and liaison, both up and down, in geographically localized parts of projects. Face-to-face groups, say, in a particular school or school system, might be headed by a local developer. Global developers may not need to have nearly as much direct contact with the project. Local developers serve a similar function as what Bonnie Nardi calls “gardeners” in her work on end-user programming (Nardi, 1993).

Students may not necessarily play important roles in materials development, per se, except as “test subjects.” But they might well benefit from the openness and adaptability of the materials they use, particularly in pursuing independent projects.

Case studies

In this section, we exemplify the LaDDER model with experiences we have had.

Number Charts

In 1997, we initiated a project of collaborative development of curriculum with a team headed by a local developer in Florida, whom we call Dean. Dean is a moderately competent programmer of Boxer, although he has never constructed any very large programs.

Dean is on the faculty of a university, and he wanted to explore the educational possibilities of computationally enhanced “number charts.” Exploring the geometric and numerical properties of colored-in number arrays, such as shown in Figures 6 and 7, is a common elementary school mathematics paper-and-pencil activity around the world. Dean reasoned that by adding computational capabilities, many more interesting mathematical activities could be supported, with less menial labor. The patterns in Figures 6 and 7 can be created in a few mouse-clicks with an appropriate tool. They would be laborious, if possible at all, with pencil and paper.

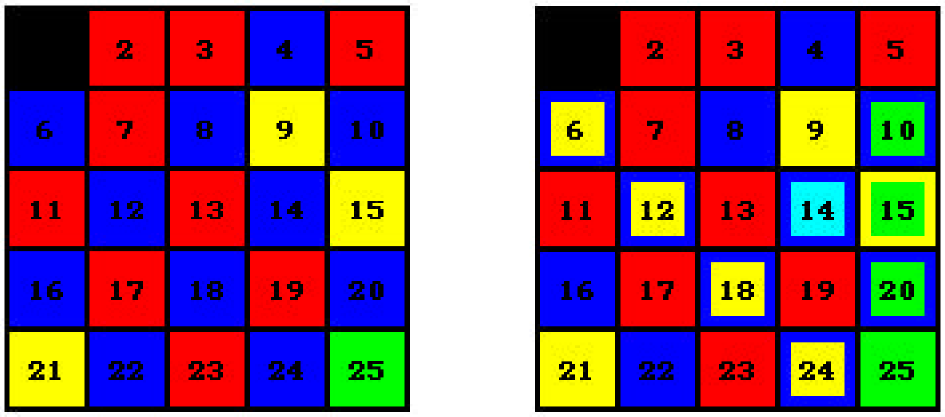


Figure 6. Sieve of Eratosthenes, to find prime numbers. (The ideas is to look through numbers, starting at 2, looking for those that are not multiples of prior numbers.) Start at 2. 2 is not a multiple of a prior prime, so it is prime and should be colored red. Cross out all multiples of 2 (blue). Moving on, 3 is not crossed out; therefore it is prime (red) . Cross out multiplesof 3 (yellow). 4 is already crossed out, and all multiples of 4 are also already crossed out. 5 is prime… Left, simple version; right, colors "stack" to show how some numbers might be multiply crossed out. For example, all multiples of 2 and 3 (blue and yellow) happen to be multiples of 6.

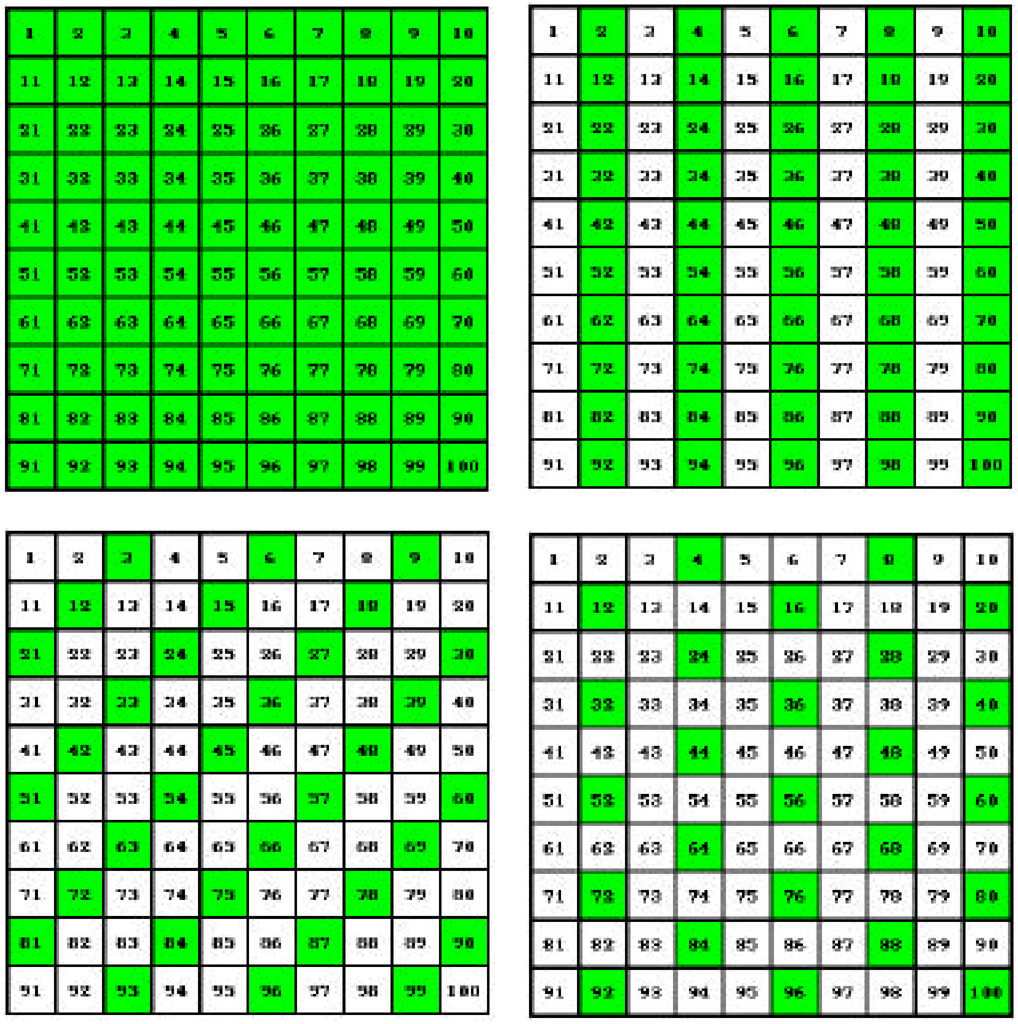
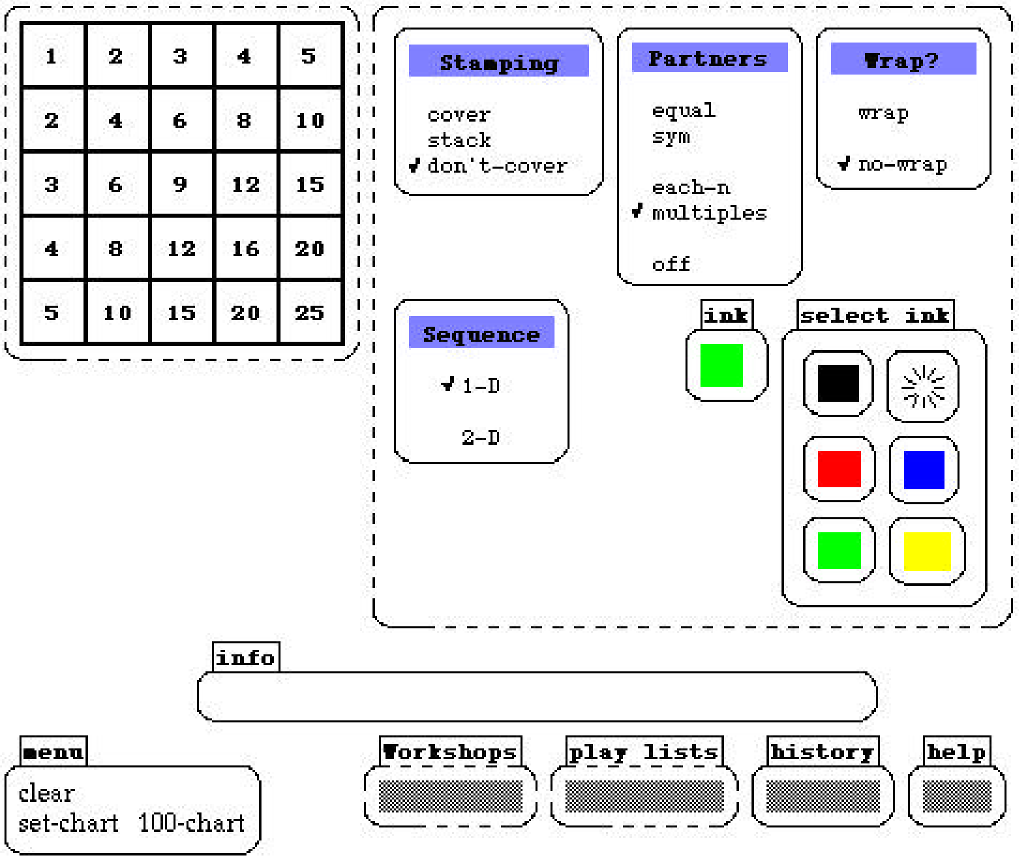


Figure 7. Multiples of 1, 2, 3 and 4. Predict the geometric pattern that would be produced by coloring multiples of any number. Classify all possible patterns thus produced.

Figure 8 shows the first version of the toolkit originally provided to Dean. At this point, modification was limited basically to spatially organizing the components (controls, number field, info box, etc.), or simply deleting components that were not needed. Dean found the initial configuration much too complex for young learners, and even for teachers. In addition, he found that it was important to make special simplified configurations for each kind of activity undertaken by students. For example, a configuration to explore patterns of multiples, as in Figure 6, might be set permanently to color all multiples of a clicked number, and it might show only the chart and a single buttons to clear the chart.



1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

Stamping: cover, stack, [✓ don't-cover]

Partners: equal, sym, each-n, [✓ multiples], off

Wrap?: wrap, [✓ no-wrap]

Sequence: [✓ 1-D], 2-D

Figure 8. Elements of a toolkit configured as a microworld. To the right of the number chart are various controls to adjust color stamping. The "info" box provides textual feedback on what is happening with the chart. The four boxes at the lower right contain: (a) places and documentation for individuals to make their own number charts; (b) a storage box ("play lists") for favorite patterns; (c) the "macro" listing of commands for the current chart ("history"); (d) documentation ("help").

In the four years since the beginning of work, Dean has worked with dozens of teachers in many classrooms. He has, with teacher collaborators, developed an extensive set of materials focusing on many aspects of elementary mathematics curriculum, as represented in number chart activities. At the same time, the original microworld has evolved into a much more flexible set of tools.

The evolution of the number chart system and its companion curriculum provides a good example of the LaDDER model. Children used the materials, but often had ideas for different ways to use the toolkit. Teachers could sometimes help the students, but, on occasion, they asked Dean for help. Dean, in turn, could often help the teachers, but sometimes he ran into limits of the toolkit or limits in his expertise that required going back to the "experts" who developed the original toolkit. Here, we illustrate these patterns by describing one episode in some detail.

In 1998, a teacher came to Dean with a request. He wanted to have a scaffolded introduction to exploring multiples, ala Figure 7. He wanted students to select a number; have Boxer show some of the multiples of that number; have the student click on the remaining multiple; and then he wanted Boxer to give the student feedback on how s/he did. Dean could manage some of these functions, but not others. He asked for help. (See Appendix B.) In a non-LaDDER situation, the global developer might, at best, just produce the particular configuration requested by the teacher. Instead, following LaDDER principles, the number chart toolkit was extended with new resources[10] that allowed Dean to finish the project, but also opened up new possibilities for future exploration. In particular, the new facilities included:

1. The ability to attach an arbitrary action to a click on a cell in the number chart (e.g., to save student input in numerical form, instead of only coloring the chart).
2. The ability to query the state of the chart (e.g., to ask for all cells colored with a certain color).

In addition to these resources, the global developer provided direct solutions to certain needs. For example, he simply wrote a "set-difference" function to score the student's efforts. The global developer also instructed Dean in how to make interactive programs that request input from the user while they are running. In this episode, most of the new resources went to the local developer, but in other cases, they propagated back down to teachers.

It would be easy to imagine that such resources as above should simply have been built in to the toolset to begin with. However, two points should be made. First, for all their experience with educational software, neither Dean nor the global developer imagined that these resources would be useful, before the teacher's suggestion. It was a year into the project before a teacher need initiated this profitable direction. (In complementary manner, some capabilities build into the original toolset were essentially never used, hence atrophied.) Second, this case is exceptional in invoking a fairly large, systematic change in the toolkit. Other requests bubbling up the LaDDER hierarchy were more idiosyncratic and even more difficult to foresee. For example, students requested the option to "turn off" the numbers, to make their patterns prettier. As another example, the local developer requested an operator to perform palindrome computations (adding a number to the number created by reversing its digits) on chart entries. Late in the development of curriculum, Dean and collaborating teachers began exploring ways of making models of multiplication more vivid in number charts (e.g., by coloring numbers in blocks of equal size). These required new resources to be added to the toolset.

All in all, the number chart toolset has evolved progressively over 4 years of work with teachers and students. At this point (2001), Dean and his teacher colleagues are preparing their materials for publication. One notable observation is that preparing a rich set of materials often takes a long time. The LaDDER model, among other things, allows global developers to contribute at critical times without needing to be present for most of the pedagogical design work. The total amount of effort on the part of the global developer we would estimate amounted to a week to 10 days of work. Thus, the model serves to match time-scales of pedagogical work (generally long), with the much

[10] We describe the capabilities added to the original microworld generically as resources, regardless of their particular form. In general, "resources" might be a new component, a new capability of a component, new "hooks" to allow better programmability, etc.

shorter development cycles needed to enhance toolsets. Note that this flips the usual relationship for development: Typically, a long time is spent coding, and relatively little time is spent “user testing.” (See also the E-Slate Profile for a description of long durations needed to adequately determine components that are truly reusable. That profile also discusses other long-term coordination issues among different communities—teachers, primary developers, educators.)

Difficulties and possible limitations of the LaDDER model: As one case study, general properties of the LaDDER model are not definitively “proven” here. Undoubtedly this case is idiosyncratic in many ways. It is always possible, of course, that the people involved were exceptional in one way or another.[11] Furthermore, number charts are fairly simple computationally, but proved enormously rich in pedagogical value, so that multiple years development of materials were appropriate. Other computationally enhanced educational materials may not make as good use of a flexible toolset.

On the whole, the LaDDER models seems a sensible approach to developing materials, with some strong advantages compared to other methodologies. In fairness, though, there were some disappointing aspects of this case, in addition to the very encouraging ones. In particular, in the end, teachers did not do as much configuration themselves as might be hoped. Perhaps the role of the local developer is even more important than we initially thought. Perhaps early elementary school teachers are not likely candidates to configure a toolset, or perhaps we need to develop better means to foster a higher degree of technical competence among teachers.

An Image Processing Toolset

Figure 9. An electronically alterable image, created by the Hubble telescope.

Another example of the LaDDER model involved only personnel from the Boxer group. In 1997, the group began designing a toolset to involve students and teachers in image

[11] In particular, while it did not involve a lot of time or effort, the global developer was fairly responsive over 4 years time, which may be unusual.

processing for purposes of scientific investigation. The basic idea is that images in electronic form, such as Figure 9, can be adjusted and inspected in many ways to support interesting analyses of astronomical (or other) situations. The fundamental capabilities of the toolset involve storing arrays of numbers that are, in the case of astronomy, interpreted as brightnesses (although other spatially distributed data, such as temperature, might also be involved). The user specifies a linear palette of colors (a simple sequence of arbitrary colors), into which the numerical data is mapped. Typically, one chooses a “maximum” number, which corresponds to the “right-most” color in the palette, a “minimum” number, which corresponds to the “left-most” color, and numbers in between are mapped proportionally to colors in between.[12] Figure 9 involves a palette (not shown) that is an ordered sequence of shades of gray.

The core toolset involved a modified version of a graphics box that (1) contains associated numerical data, that (2) accepts different palettes, maximum and minimum settings, and that (3) can rapidly map numbers to colors for redisplay. We call these specialized graphics boxes “image boxes.” In the months leading up to a course for middle and high school students involving image processing, a subteam of the Boxer group created a richer toolset around the core capabilities. The toolset consisted of:

A fairly rich tool for creating color palettes. For example, users could select from some basic options, such as a “rainbow” palette, a greyscale continuum (from black to white), or they could linearly interpolate between any two chosen colors. Users could also concatenate palettes, change individual colors in a palette, keep a “scratch pad” of test palettes, and, when done, send their finalized palettes to a global storehouse that could be used in any image processing activity. The tool was operated by clicking buttons or dragging and dropping colors and palettes onto one another. See Figure 10.

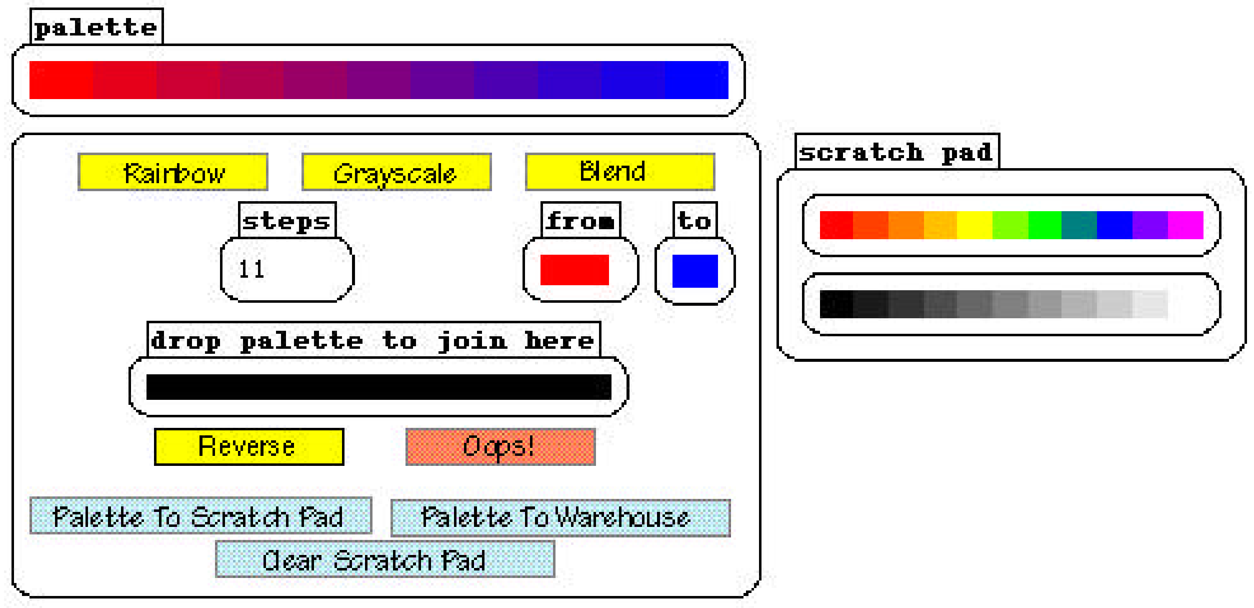


Figure 10. A tool to create color palettes.

[12] Typically, data values above the maximum are mapped into the right-most color, and numbers under the minimum are mapped into the left-most color.

A tool for creating colors. Another tool, similar to the palette tool, allows experimenting with individual colors.

Slightly modified generic sliders to control maximum and minimum settings. These sliders can be connected to image boxes with a drag and drop gesture.

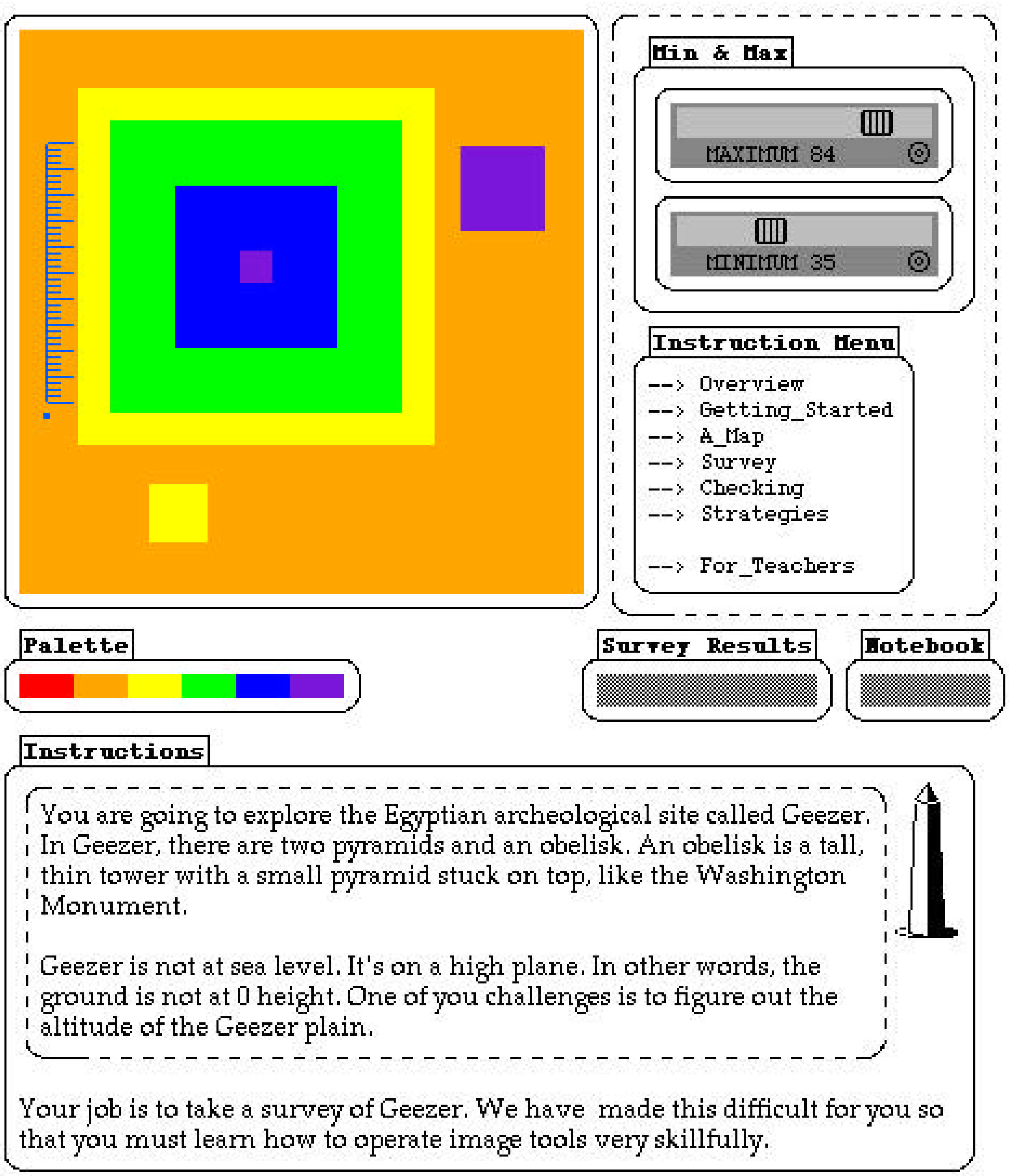
A tool to view the numerical data in a small, square patch of the image box. The user drags a square cursor around in the image box; the corresponding numerical array appears in a data box.

A "slice tool" that allows users to drag out a linear "slice" of an image, and the tool graphs numerical data along that segment.

With this augmented toolset, the full Boxer group and the teacher for the course worked over about a month to create a large number of tool configurations to support particular activities that were used successfully in the course. The salient points are (1) that a large amount of materials and activities were created over a short time with the toolset; and (2) not a great deal of technological expertise was involved. While all members of the group were familiar with Boxer, only a few had done much programming. In addition, very little programming was actually done by anyone at this phase; most work consisted in configuring a particular collection of tools, making minimal modifications (e.g., via scripting). This phase corresponds roughly to local developer level of technical expertise. In this case, it was important to have a significant group of people making materials, much more than we could have accomplished if "global developer" skill levels were required at this stage. A partial description of the materials and activities is given below.

1. An early introduction to the idea of image boxes and palettes, where only a few (large—on the order of 1/4" square) data elements are used so students can directly see the map between numbers and palette colors.
2. An exercise microworld in which students are invited to go on an expedition to explore an imaginary archeological site in Egypt, where pyramids, obelisks, etc., are shown only by color-coded altitude over the site in an image box. See Figure 11. The construction of this microworld is described in some detail in Appendix A.
3. Another exercise microworld where student import electronic images of themselves, and try to perform actions such as adjusting minimum and maximum to bring out as much detail in their faces as possible, or to try to color all their hair purple (which is generally impossible—and a good lesson on the meaning of color map).
4. A configuration of tools so that students could explore images of the moon, and, from measurements of shadows, determine relative altitudes of crater edges, mountain peaks, etc. A novel part of this task was that students first developed hypotheses about how to determine height information from images, then built their own mechanical models of a "landscape" with particular lighting, captured an image of their landscape, imported the image into Boxer, and tried out their strategies using tools supplied to them.

5. A microworld where students inspect images such as Figure 9, trying to distinguish stars from galaxies, and near-field stars (in our galaxy) from parts of background galaxies, etc.
6. A configuration of images and tools that invited students to examine two complex star images, one of which contained a single deviation corresponding to the appearance of a new supernova. Students were led to understand how a “difference image” (pixel by pixel difference in values) would highlight any changes between the images. For the conclusion of the activity, the difference image of the two introductory images was computed and displayed, vividly showing the supernova.
7. A simple extension to an image box that invited students to program various strategies to “hill climb” to find the brightest parts on an image.



You are going to explore the Egyptian archeological site called Geezer. In Geezer, there are two pyramids and an obelisk. An obelisk is a tall, thin tower with a small pyramid stuck on top, like the Washington Monument.

Geezer is not at sea level. It's on a high plane. In other words, the ground is not at 0 height. One of you challenges is to figure out the altitude of the Geezer plain.

Your job is to take a survey of Geezer. We have made this difficult for you so that you must learn how to operate image tools very skillfully.

Figure 11. An introductory activity, constructed of image processing tools and generic Boxer resources. (a) In the upper left is an image box containing the graphical display of numerical data, representing altitudes over an archeological site. The "ruler" (left side of the image) is a generic Boxer mobile graphical element, which took only a few seconds to add to the image. (b) Top right: Sliders, modified from generic sliders so that they know how to connect themselves to a particular *part of* the image (graphics) box. (c) Under the sliders: A menu of steps (which bring up new instructions and modify the configuration by scripting) to guide students in the activity. This is a very simple modification of a generic Boxer tool. (d) Middle row, left. A simple palette (part of the image processing basic toolset). (e) A box, Survey Results, into which students are intended to place their measurements of the Geezer site. (f) A Notebook,

consisting box for students to write down their personal observations in working through problems posed. The Notebook also automatically collects all the results students type into “answer boxes” to questions asked in the Instructions. (The Instructions contains ports to boxes in the Notebook.) (g) On the bottom are the Instructions for the particular stage of the activity the students are at. In this case, Instructions contains overview text. In other cases, Instructions contains specific question, hints, and “answer” boxes where students type their answers for later inspection or for instructor monitoring/grading of student work.

The curriculum development described above illustrates the creation of many activities and materials in a short time by a group of pedagogically competent (but technologically limited) individuals—roughly at the “local developer” level. In addition, this episode illustrates a number of points we made in motivating and describing the LaDDER model.

Generic tools were not up to the task. The generic graphics box needed to be modified to support needed image functionalities, such as quick redisplay of colored images, “understanding” the meaning of palettes, etc. Even generic sliders were adapted for this context. For example, we made special sliders whose drag-and-drop method, when connecting to an image box, did not adjust the contents of that box, per se, but linked to a particular subbox of that box (i.e., the maximum or minimum variable). This and similar experiences supports our general contention that modifiability is extremely important, and also supports the contention that general tools frequently need tuning to work well in a particular context, with particular other components.

A more elaborate example of the need for modifiability and creating toolsets, rather than more generic components, was what happened to the generic graphing component. We began using the grapher by adding a one-line script that converted the generic grapher component (plus segment selector tool; see list of generic components, above) into a slice tool. Preliminary work suggested this was not sufficiently adapted to be really useful in this context. So, a number of additional features were added. (a) The graph “remembered” the particular slice and which image had been sliced to create its current graph. Dragging a vertical red control bar across the graph “lit up” the slice and showed the particular place on the slice that corresponded to the current x position of the bar. (b) During this drag, the numerical value of the pixel quantity (y value on the graph) corresponding to the x position of the control bar was also displayed. (c) Provision was made to easily save “thumbnail” images of graphs, which could be dragged and dropped back on the grapher to reinstate a previous slice.

These modifications involved one of the “global developers” getting back into the act, modifying the graphing tool after the first-year’s edition of the course in preparation for improved and extended materials for the second years’ edition of the course.

Generic functionality of the container environment served many purposes well. While not uniquely connected to the LaDDER model, generic capabilities of the Boxer environment were frequently used. Text to guide students was typed directly into the environment and edited by anyone who chose to. A “ruler” tool was added to the Egyptian archeological site image box simply by dropping a mobile graphical element (in the shape of a ruler) into the image box. A “notebook,” which was little more than an empty box, was entered into some environments.

The teacher made significant contributions, and modifiability was frequently exploited by him and others. The classroom teacher had significant input into the design of materials. Typically a local developer would produce a quick prototype, and the teacher provided suggestions for change. In addition, the teacher essentially always edited the text in the microworlds to suit his sense of what was important and how to draw students' attention to it.[13] Often this editing resulted in reordering activities into a more pedagogically felicitous sequence.

Openness and modifiability supported student creativity. Because all of the tools were open for inspection, disassembly and change, students' creative initiative was significantly supported. For example, one student had the idea of a "color zoom" tool. The idea is that the user could click on a particular color in an image, and the minimum and maximum of the color map would be reset to the minimum and maximum values that were displayed *in that color*. Thus, all of the colors in the palette were concentrated to show detail in precisely the data range that previously showed no detail at all—the range that showed as only one color. On another occasion, the palette maker tool was used by students entirely independently of the image processing tool set, to investigate perceptual properties of colors. See Friedman and diSessa (1999) for further examples of student creativity by extending, modifying, and using image processing tools for unanticipated purposes. The image processing toolset is available on the Web.[14]

Difficulties and possible limitations of the LaDDER model

Curriculum designers' (local developers') work was not always as smooth as might have been suggested in the descriptions above. One curriculum designer used a slight modification of a slider to implement the ability to scale the size of image boxes. (The boxes did not have the possibility to scale simply by resizing the box. But a programming command did the resize, and this designer used a slider as an easier interface than a programming command.) The problem was that resizing took a long time, so the slider action was very jerky. To avoid this problem, a new element was added to the toolset to change image size by clicking on "increase" or "decrease" buttons. In general, local developers may not have superb interface sense. More generally, good tools never obviate the need for good ideas and design sense. On the other hand, good tools can facilitate quick and excellent deployment of a good design, and they can, in some instances, help foster better design by demonstrating it, and by incorporating it into existing components.

Another curriculum designer noted, but did not know how to fix, a problem that occurred with image boxes involving large data values: Graphs were slow to appear. The problem was that thousands of tick marks were being drawn, since the smallest tick-mark interval had not been changed. One might adjust the tool so that the selection of tick-marks is automatic. But this brings its own problems, limiting manual control

[13] The teacher in this case also happened to be a superb writer, which, of course, is not always—or even frequently—the case. However, the ESCOT Project also noted that teachers made strong contributions in terms of creating text. For "full disclosure," we note that this teacher could also program in Boxer, although he left essentially all programming to colleagues in the production of these materials.

[14] One path to the materials is via the Boxer Hub, which is directly accessible as a "net box" in the "!Start Exploring Boxer Here!" box that comes with Boxer release. Download the Boxer plus documentation and demos from http://soe.berkeley.edu/boxer.

(say, you want fixed tick-mark interval to make a collection of different graphs more easily comparable). Instead, this example probably points to a class of problems that the toolset idea doesn't address: Users (including local developers) usually don't read documentation; no tool, no matter how simple, is transparent in the way it works; reading code might in principle be a good way to understand a tool, but, like documentation, it is often ignored, and so on. Still, the problem of slow-to-draw graphs was trivial to fix, given the social context. When a global developer was asked about the problem, he quickly discovered the issue and instructed the curriculum designer about adjusting tick mark intervals. This last illustrates that one important function of global developers is simply to teach on occasions where the teaching is directly related to the task of the "learner." Many episodes in the number chart LaDDER case worked in this way, as opportunities to teach.

Future Work

As an unfunded project, future component work by the Boxer group is uncertain. However, we note one promising possibility. Boxer's technology is probably most distinctive in offering an extremely rich, flexible, and relatively easy to use container environment. Other component projects could take advantage of these resources provided Boxer could host fairly generic Java components. So, for example, Java beans might be imported into Boxer, then serve as cut-copy-and-paste components. A minimal interface with Boxer and other components would be to make bean methods and data available via Boxer scripting. Similar to "flipping a graphics box," "flipping a bean" could provide documentation and direct access to local methods and data.

The point of such a development would be to enable straightforward test of the usefulness of the rich container model. If essentially the same components, but embedded in a rich container, really enhanced development and, particularly, enhanced the ability of less technologically sophisticated individuals to contribute to development and adaptation, then the rhetoric in favor of rich containers might be converted into empirical results.

Summary and Highlights

This profile of Boxer components highlights a few core ideas. First, we highlighted an orientation toward component computing that is unusual: We advocate moving the possibility of contributing directly to technical development as far as possible down the technology expertise hierarchy. Advantages of this possibility include:

- broadening the size of the development community,
- making materials that are much more adaptable to local needs and concerns by users,
- promoting teacher professionalization, and
- facilitating social structures of development that
    - rely less on programmers,
    - rely more on people with more pedagogical expertise, and

- do a better job of matching different time-scales of development between different communities. (See the "lemon tree" example of the E-Slate profile for more on the latter issue.)

In pursuit of reducing the technical expertise needed to configure, modify or construct components or component configurations, we introduced several key ideas. First, we advocated the use of richer container environments. Second, we discussed the possibility that generic components would not efficiently serve the needs of particular curricular areas, hence motivated the need for a "toolset" collection of components that were specially configured to the use in question and to use with each other.

Finally, we introduce the LaDDER model, which is a social configuration for development consisting of students, teachers, local developers, and global developers. The LaDDER model seeks to recognize the strengths, limitations and constraints of each layer to produce an optimal development collaboration. In it, problems and needs propagate up the technology expertise hierarchy, and resources (rather than solutions) propagate downward to expand the most creative avenues available to even the least technologically sophisticated participants.

Summary difficulties and limitations

Although there are many potential concerns and difficulties with the models introduced here (e.g., the rich container model; the LaDDER model), we limit this discussion to global issues that have not had much play up to this point. In this regard, probably the most obvious difficulty for these models and paradigms is the use of non-standard technology. The fact is that browsers are standard and ubiquitous, and systems like Boxer are not. Even if the advantages of a rich container and the LaDDER model are empirically proven, the path to wide-spread realization is not obvious.

Although the data we have analyzed are very encouraging, the assumption that significant advantage can be had from teachers', local developers', students' and other less technologically expert individuals' participation in the creation of computational resources for learning is not widely accepted. Experiments, such as proposed under "future work," will help resolve the issue.

Acknowledgements

This work was supported by a grant from the National Science Foundation, number REC-9973156. The opinions expressed here are those of the authors, and not necessarily those of the Foundation. We thank members of the Boxer group who contributed to the formulation of ideas and provided feedback on early versions of this report, including Flavio Azevedo, Orit Parnafes, Jeff Friedman, and Joe Wagner. Don Ploger's work within the LaDDER paradigm was inspirational.

References

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

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.

diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.

Friedman, J. & diSessa, A. A. (1999). What should students know about technology? The case of scientific visualization. *International Journal of Technology and Science Education, 9*(3), 175-196.

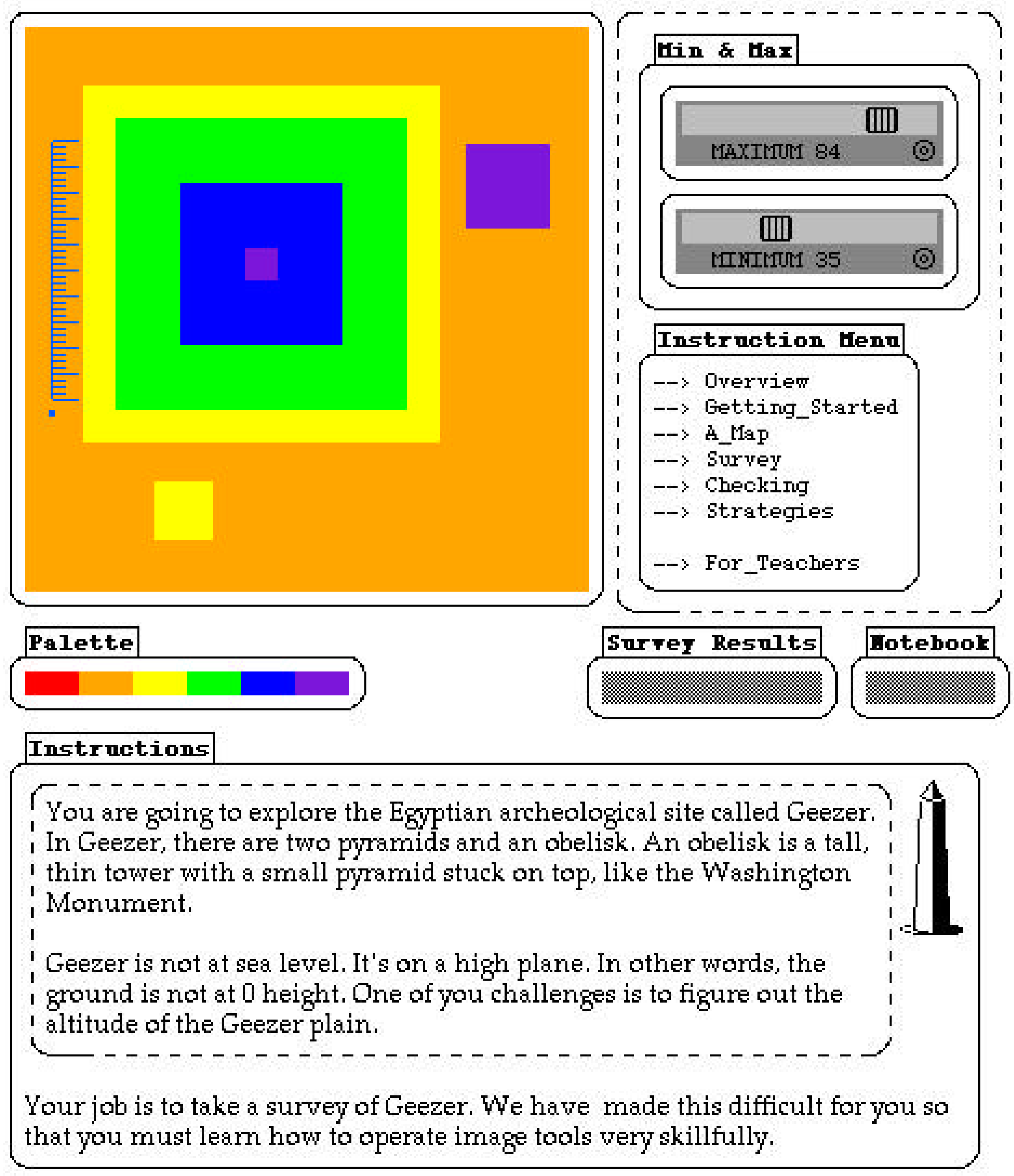
Nardi, B. (1993). *A small matter of programming*. Cambridge, MA: MIT Press.

Ploger, D. & Della Vedova, T. (in press). Dynamic Charts in the Elementary Classroom. *Learning and Leading with Technology*.

Appendix A:

This appendix gives a blow-by-blow description of how to create the configuration of components and contents that define the activity microworld illustrated in Figure 11. The figure is repeated below for the convenience of the reader. The point is to demonstrate how components can be assembled and modified in Boxer. More generally, this scenario illustrates how the technical aspects (that is, “coding”) of creating an educational piece of software can be dramatically reduced in time with appropriate components and tools. In particular, neglecting the design of the activity and inputting text (jobs that could be accomplished by a pedagogically expert, but technically naïve individual), creating the software in Figure 11 could easily be accomplished in less than a half hour. In practice, of course, building the “software” was done incrementally as the activity was designed and new software features were needed.

We proceed by explaining how to create each element of the Figure 11, in sequence. We note, in advance, that most of this work, which involves copying, pasting and typing, could easily be done by Boxer novices, such as teachers. A part of the work requires modest programming, probably doable by a secondary developer. Realistically, a primary developer might have to act as a consultant since at least one component was slightly modified in this construction.



You are going to explore the Egyptian archeological site called Geezer. In Geezer, there are two pyramids and an obelisk. An obelisk is a tall, thin tower with a small pyramid stuck on top, like the Washington Monument.

Geezer is not at sea level. It's on a high plane. In other words, the ground is not at 0 height. One of you challenges is to figure out the altitude of the Geezer plain.

Your job is to take a survey of Geezer. We have made this difficult for you so that you must learn how to operate image tools very skillfully.

(a) *In the upper left is an image box containing the graphical display of numerical data, representing altitudes over an archeological site.*

The image processing toolset contains a command to create image boxes, which are specialized versions of the generic Boxer graphics box that understand image processing commands. Creating the image box involves deciding on its dimensions and the default value for each "patch" (pixel) in the image. Actually typing the command and executing it takes only a few seconds. The resulting box can be cut and pasted wherever it is desired.[15] **[time: < 30 seconds for "coding" (excluding design)]**

Actually creating the landscape data (pyramids and obelisk) might involve writing a short program that sequentially adds square arrays of constant data (i.e., a new layer to the pyramid or obelisk) one layer at a time in appropriate places in the image. Although writing such a program would take about 10 minutes for a competent Boxer programmer, in point of fact, we already had a tool to do this kind of construction with a graphical gesture. Both programming from scratch and the graphical tool employ a command to which image boxes respond: adding an array of numbers (pixel by pixel) at a specific location in any existing image box. **[time: up to 10 minutes]**

The "ruler" (left side of the image) is a generic Boxer mobile graphical element.

Adding a draggable graphical element to a graphics box amounts flipping the graphics box and selecting the "sprite" option in the Boxer "Box" pulldown menu. Thereafter, the shape of the ruler (scale hatchings) can be created with a simple repetitive "turtle" program. **[time: < 5 minutes]**

(b) Top right: Sliders that adjust the mapping from numbers to colors.

The image processing toolset contains sliders as pre-made components. One just makes a copy of the slider and pastes it into the position where it is desired. Then a drag and drop gesture links the slider to the image. The range of adjustment of the slider may need to be set, which amounts to flipping the slider and typing in new values. **[time: < 2 minutes]**

(c) Under the sliders: The "Instruction Menu" is a menu of steps (which, when clicked, bring up new instructions).

Boxer's generic toolkit includes a simple cut-and-paste tool to do this. Ordinarily, the menu is hypertext, and a new page is displayed when links (-->) are clicked. In this case, the teacher wanted more stability in the menu, and a ten second change resulted in a menu that did not change, but which caused changes in the "Instructions" box (bottom row in the figure). Operations involved: Copy and paste the documentation tool; edit the name of the box to be changed by links (in the closet of the tool). This action might require a primary developer level of expertise **[time, excluding typing in text for each link, < 1 minute]**

d) Middle row, left. A simple palette.

This is part of the image processing basic toolset. Copy and paste. **[< 1 minute]**

[15] Technically speaking, the basic image processing toolset is a library of commands added to Boxer as a Boxer extension (like Macintosh OS extensions). Then, a file of graphical components using those resources can be read into Boxer for developers to copy for pasting into new creations.

(e) Middle row, “Survey Results,” in which students place the results of their measurements.

This is simply a generic box, essentially instantly creatable with a Boxer menu item. Survey Results contains a number of subboxes for particular results, e.g., width and height of the obelisk, uncertainty of measurement, etc. **[time: inconsequentially more (a few keystrokes) than deciding on what should be measured and typing in appropriate labels]**

(f) Middle row, right, a notebook.

Like “Survey Results,” this Notebook is just a generic box, which takes a few seconds to create and label. Similar to Survey Results, the Notebook also contains empty subboxes for answers to particular questions asked in the instructions. In this case, the creator of the text for the activity placed a port in the instructions to the appropriate “answer box” in the notebook, so that students don’t even need to find the relevant part of notebook when they are working on particular issues raised in the instructions. (Recall, anything typed into a port also appears automatically in its “target.”) The time to accomplish these port links is little more than a keystroke or menu selection for each box or port, and is easily done by a Boxer novice. **[time: inconsequentially more than deciding on relevant questions and typing them out]**

(g) On the bottom are the instructions for what students should be doing at each stage of the investigation.

As we mentioned, “Instructions” is a generic box that would take literally only a few seconds to create, place and label. The “Instruction Menu” changes the Instructions’ contents by inheritance (i.e., the “Instruction Menu” component expects and “talks to” a box called “Instructions”); no explicit linking is necessary. **[time: a few seconds]**

Notice that no text input facilities need to be programmed at all. Students just type text using Boxer’s always-available editing capabilities.

Appendix B:

We present a letter (edited only to remove identifying data) from a “local developer,” describing his experience in working with a teacher, and requesting help from a “global developer.” Note (1) the local developer is trying to support creative involvement by a teacher. (2) He is capable of handling most of the teacher’s request. (3) Following the LaDDER methodology, he requests tools, not solutions.

> A. [global developer],
>
> This request came from M., the teacher of the 4th grade class where I will be doing much of the work (and the class where R. is student-teaching).
>
> The great news is: M. is even more impressed than before with how much Chartworld helps children at both end of the spectrum of mathematical talent. Children who are struggling in school math often become fascinated by their

success with Chartworld. And, of course, very talented kids create truly impressive projects.

Another bit of really great news is that M is finding Chartworld interesting in its own right.

Therefore, when he asked for a microworld, I decided it would be good to support his interest.

He wanted to have Chartworld make part of the multiples pattern, but then have it change the color so that the student could complete the pattern. The final verification stage would show all the multiples, and provide feedback on the student's accuracy.

I made a version that basically works. It would be great if you could give me a few hints to spiff it up. The best thing would be for you to help me out with a few big tools (and also some interface ideas -- how do you interrupt a program to allow the student to click, and then resume to verify the work)?