CHAPTER 6

Notes on the Future of Programming: Breaking the Utility Barrier

ANDREA A. diSESSA

In preparing this chapter I felt obliged to look at a number of other discussions of the future of programming. Few of them mention anything resembling the issues discussed here. Instead, probably the most prominent theme is the "complexity barrier"—the tremendous difficulty and cost involved in creating and maintaining huge programs. This is, no doubt, a serious and enduring problem. But it is the most serious one for professional programmers, not nonprofessionals who, at least in terms of numbers, will dominate future use of computers. More than numbers, I believe the ultimate social and cultural impact of computation will be determined to a great extent by what we can cause to happen when technologically unsophisticated users sit down at a machine. The hope I share with many others is that computation can significantly enhance intellectual development and productivity for most, if not all, people. (Two exemplary references are Winograd, 1984a, 1984b.)

What role will programming play in this? Some of my favorite antagonists in this regard feel that computers will totally disappear into the woodwork as far as ordinary people are concerned, the way electrical relays and motors and control micro-processors have already disappeared. Even if they don't disappear, certainly, it is said, the ordinary person will need to know as little about programming as about repairing

an automobile—that's a job for specialists. Then the future of programming would indeed fall back into the province of specialists and strike the complexity barrier head-on.

If one were to judge from present programming languages, I might well agree. They are clumsy, inelegant, hard to understand, harder to learn, and just don't do very much very easily. So I am saying the future of programming in the large-scale scheme of things is not evident in what we currently have, but will depend largely on future forms and contexts for programming that we invent for nonspecialists.

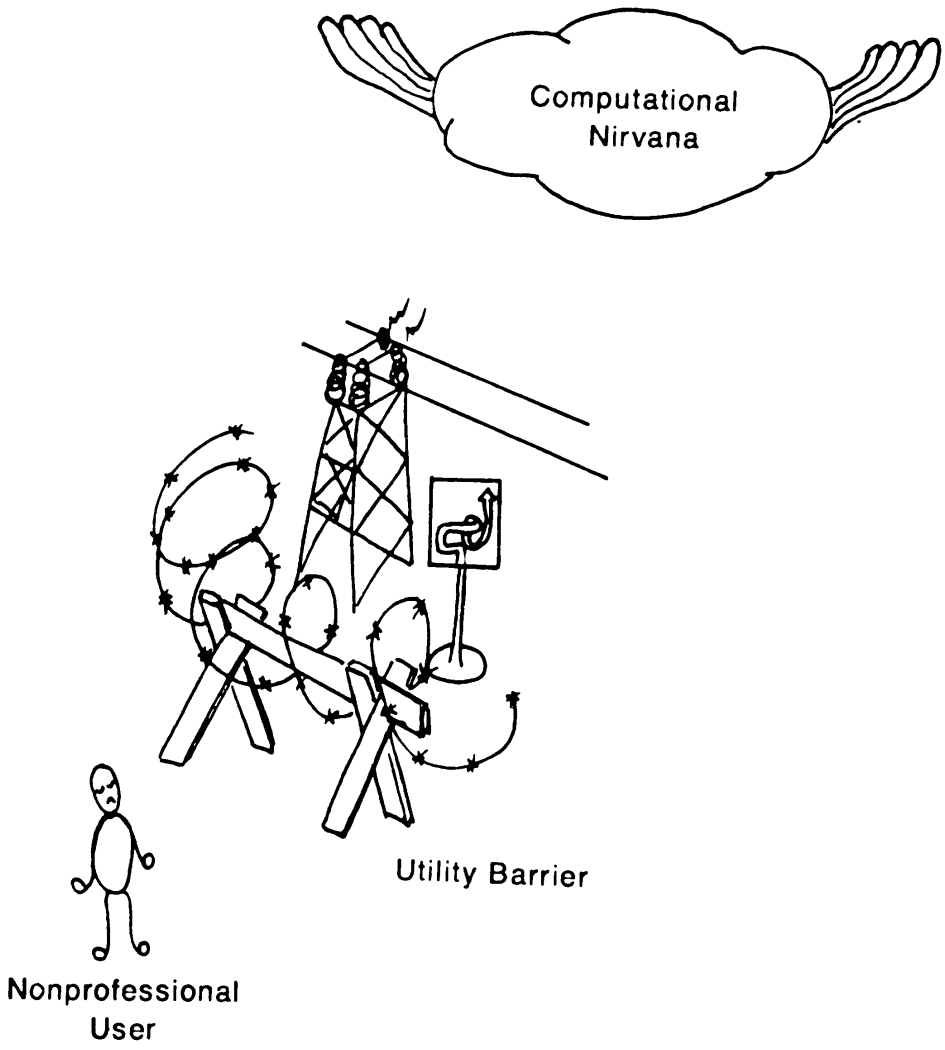
What might computers do for nonspecialists? What should programming be like for them? If I were to pick a single image, it would be the computer as an interactive, constructible and reconstructible medium. "Interactive" deserves a great deal of discussion, but it does not need it. The degree of interactivity is the well-recognized key difference between computation and all previous media. "Constructible and reconstructible," however, are just as important, but much less discussed. The written word would not be half as powerful as it is if most people couldn't write. As a tool for accomplishing information related tasks, for thinking, for inventing, for developing oneself intellectually, text would be next to useless, except as a way to convey to others what some "expert," who could afford to hire a scribe (read "programmer") wished to convey. We need in interactive media the equivalent of notes to yourself in margins, crossings out, personal summaries, and your own essay or even book—little things in bits and pieces that can be put together into a masterpiece, but don't need that scale of effort to warrant doing. Even great masters of interactive media, the equivalent of a great author or poet, would be limited unless they could play, tinker and create in pieces, rather than waiting for the long-loop to the scribe and back. Surely there will always be the equivalent of technical editors and publishing houses, but just as surely, there needs to be the equivalent of pencils and paper for everyone.

So programming will mean being able to construct and reconstruct in interactive media. This is why the automobile engine metaphor is an inappropriate way of thinking about all computers. If a machine is to serve one very specialized role, such as providing mechanical power, it ought to be hidden and of no concern. But if the purpose of the machine is flexibility and personal adaptability, we had better figure out how to give users maximum control. Expressive power and nuance are incompatible with invisibility and inaccessibility.

Programming languages viewed in this way will certainly be, in many ways, much different than those of today. In the first instance they had better be far superior to present languages at interaction. Long, silent, invisible algorithms are much less the point than being able simply to

arrange to observe and control an ongoing process. Languages had better make modification of existing structures and processes very easy. They had better accept bits and pieces of whatever the medium supports, certainly text, programs, and pictures, to be recombined easily into a new product. They had better respond to the complex, interleaved activity structures of humans much better than any set of isolated programs can do. See the chapters in the section on User Activities, Section IV of this book.

There will, however, be many similarities between present and future languages as well, at least for some time to come. There are certain things one simply needs in order to describe a complex process. Representing state, controlling availability and flow of information, deciding what happens when: These seem very likely to remain salient, in one form or another. The meaning of programming will change, but I am certain that the means of construction and reconstruction will be powerful and complex enough that no one will be embarrassed to call them programming languages.



The complexity barrier has surreptitiously changed into the utility barrier. The challenge for the future is to make programming into something that is simple enough and useful enough that everybody will want and be able to learn how to grasp and stroke with this new pencil. Complexity is still an issue. But it is complexity with an emphasis on a different range of the spectrum. In contrast to battling huge programs, if we can allow a broad range of simple but useful things to be done transparently, we will have won more than half the battle. And complexity is by no means the full story. Please note that utility is the ratio of value to effort expended. Utility can be increased by decreasing the denominator, with which complexity is intimately involved, or by increasing the numerator. We need to worry as much about value, about the uses of programming, as about making it easier to do.

I do not wish to play futurist here and wax poetic about possibilities we can barely imagine. Instead, I wish to look conservatively to the near future—in some cases, the recent past seems not yet to have been noticed—and make the simple remark: From an engineering point of view, it is clear that, far from reaching an equilibrium, we have at best crossed a threshold in terms of making computation accessible and useful to everyone. If we wish to, we will be able to make rapid changes in what programming looks like, and in the context for learning construction and reconstruction in an interactive medium.

> *Putting programming in terms of utility rather than as good in itself is a position that might surprise some who know the history of our work. "We" are the Logo Group at MIT, and its descendants. The language Logo, our best known product to date, was developed in the aura of general intellectual skills that can be developed in computational environments, and of deep mathematical and scientific ideas that can become part of everyday experience through programming. Thus, many might expect me to write this note about designing programming languages on the basis of what powerful ideas one could build into them. But I have come to believe several things that move utility to first place.*
>
> *First, as a matter of fact, we have had much more success building microworlds, computational environments for open exploration of particular clusters of ideas, on top of a programming environment rather than into it. Interest and the many things that we want to teach people are mostly at a different level than the generalities of programming per se. Second, the need for a lingua franca of interaction is great.*

> *Multiple microworlds, each strongly tuned to its own particulars, may be too disparate to allow any economy of learning the interactive medium. In fact, the pressures of commonality go beyond the individual. Designing a new interactive medium will certainly be a long term and gradual social process in which what eventually solidifies must serve the everyday purposes of many, yet be tailorable gracefully into technical domains. Natural language, for example, is useful for everybody, but easily accepts technical terms and conventions of exposition for specialists. Third, no matter how valuable it might be to have learned programming, it will be harder, and in other ways less enticing a job, without perceived utility along the learning route. This is a major theme of this chapter. Finally, it seems to me that the good things intrinsic to programming are very robust and will not go away if we put utility in a more prominent position as a design goal.*

I consider three dimensions of change in programming. Each can bring us noticeably closer to breaking the utility barrier.

> *Since complexity is a problem (bad) and utility is a goal (good), perhaps a more parallel wording would be the "uselessness" barrier. But complexity is bad like sex in a Puritanical society (interesting, nonetheless), and uselessness is bad like yesterday's garbage. So I'll stick to the utility barrier.*

PRESENTATION

The first type of change is *presentational.* These are not modifications to the underlying structure of programming, but only to how it is visually (and, potentially, through other senses) presented and manipulated. Presentational changes will not, in general, affect the ultimate perceived power of a language, but most certainly they can dramatically reduce the effort involved in learning and understanding it. My points will be in the form of examples.

Beyond Logo

Logo's roots are in the teletype interfaces that were available when it got its start. The communication format between user and machine is linear and "conversational." The user says (types) something, and the machine says something back. One small problem with this format is

simply that you cannot even point to something you "said" a few lines ago in order to "say" it again. The trace of what was done is, in computational terms, a useless artifact and cannot be, for example, turned into a procedure. Initially, things even got worse with the demise of hardcopy terminals; in addition to having to type over tried out commands, users had to remember what they just tried if it scrolled off-screen. Of course, versions of *concrete programming* (making a program essentially by doing, step by step, what you want to have happen in the program) can be built in Logo, albeit somewhat clumsily. Various popular versions of what I started calling "instant" (single key activation) programs a number of years ago incorporate this feature. But this is not the ordinary way one programs in Logo.

A second disadvantage of conversational interaction is that large-scale structures are difficult to notate and manipulate as a whole. The Logo END command is really no command at all, but a syntactic marker of the boundary of an object (program). Unfortunately, END can easily be confused with parts of the program because it must look just like another piece of the conversation and cannot be connected visually with the previous part of the conversation, the start of the procedure definition, with which it should really be connected.

A more profound but subtler problem is that you cannot directly see and manipulate the state of the system on the screen. Instead, you must send a request to see some state (e.g., PRINTOUT), and if you want to change it, you must send another request (e.g., MAKE or TO). Recent microcomputer implementations of Logo incorporate a screen editor to ameliorate these problems to some extent, but this is only a patch. One still must make a major mode switch, entering the editor, if you want at all to pretend the screen shows the state of the system. And the details of the relation of the editor buffer, the definition process and the defined state of the workspace are both invisible and subtle.

> *Here's a hack to show the subtlety. Suppose you want to delete all but one procedure from your workspace. It is painful to type ERASE APPLE, ERASE BEAR, ERASE CAT, ... So what is frequently done is to load the editor with the whole workspace, EDIT ALL. Then use a few simple edit commands to delete everything but the wanted procedure. Then exit the editor, clear the whole workspace, reenter the editor (the buffer is not lost) and reexit again, which redefines the one procedure you wish. This is the very opposite of a direct manipulation system.*

The bitmap display, a pointing device, and enough memory can solve all these problems. In Boxer (it's all boxes!), the language we are designing as a successor to Logo, we believe we have done this. Computational objects such as programs are visual units, boxes, and are trivially manipulable as a whole, somewhat like a large character in a text editor. The program that appears as a box in Figure 6.1 can be deleted by pointing to it and pressing the rubout key. Similarly, it can be picked up and moved around as a unit.

In Boxer we have changed the conversational interaction paradigm to "looking at and directly altering the state of the system." Boxer is "editor top level": You are always able to change directly or use anything you (or the computer) have put on the screen. This automatically gives you a simple form of concrete programming since you can simply select a set of lines you have typed and executed, and box them to make a program out of them. More fundamentally than that, once one learns the few basic editing actions to point, pickup, move and delete, one can inspect, construct, or modify anything in the system without learning any more commands. The programs in Figure 6.1 can be edited just by moving the cursor into them, deleting a few characters and adding some new ones. The editor constitutes your feet and finger tips for moving around and building your computational world.

We call this profitable illusion that one sees and directly manipulates the system on the screen *naive realism*. It is, in a sense, an old principle, but it is rarely applied to computational structure. The principle has many implications. It makes the system easier to use in that you can always see what you are doing in changing or adding to the system. A small set of editing commands can replace all the separate structure creating and modifying commands. If you can remember or imagine

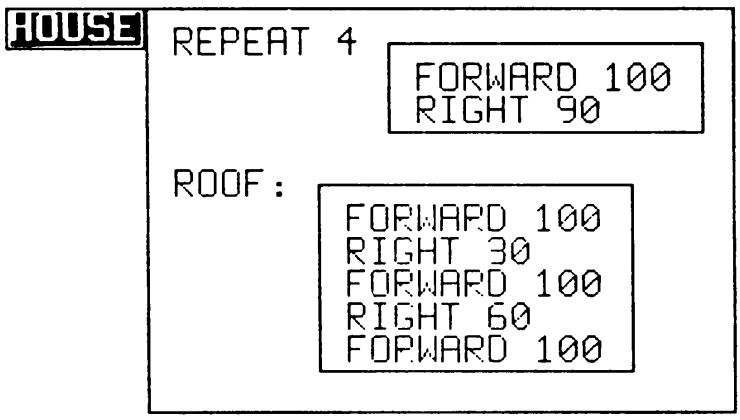


FIGURE 6.1. A simple program definition in Boxer. The tab is the procedure's name. The box labeled ROOF is a subprocedure written in place.

what some structure should look like, you can create it. The visibility that comes with naive realism also makes many aspects of the system easier to understand, particularly its large scale organization. For example, procedure/subprocedure relations may be made absolutely explicitly (Figure 6.1).

Over long time scales, seeing the structure of the environment rather than only a trail of a recent "conversation" should pay handsome dividends, especially for those who may have difficulty keeping system organization in their heads, children for example. In order to find something in the system, you may simply wander around looking for it. (See also Figure 6.4 and surrounding discussion.) Note that the ability to make an audit trail is not lost from Boxer. All it takes is the discipline to type rather than point. More elaborate and automatic audit trail mechanisms, of course, may be programmed.

Boxer makes another contribution toward presentational change that goes hand-in-hand with the naive realism just described. The geometric configurations one sees on the screen express fundamental semantics of the language. I call this the *spaital metaphor*. For example, containment implies inheritance. Procedures and variables that are defined in a box are accessible only in that box, and in recursively contained boxes. Thus, the important modularity constructs of environments and namespaces are plainly visible. This use of regions to represent namespace environments is a concretization of the "contour model," which is frequently used to explicate these ideas.

We have also taken pains to make sure that dynamic as well as static structure of the language is visible. If one chooses, the execution of a procedure can be watched. The essential features of this visibility are very simple. Commands making up a procedure are highlighted successively as they are executed; returned values appear in place of the procedure that returned them; and one has the rule that to watch the execution of a procedure called by name, a copy of the procedure definition will replace the name as the first step in executing it. (Details can be found in diSessa, 1985). Compare Figure 6.2 to the little man model shown in Figure 10.1 in Chapter 10 in this volume.

An Iconic Presentation

Let me give one other example of potential presentational changes in programming. Quite some years ago, Radia Perlman and Danny Hillis constructed a device known as a slot machine. Children programmed by inserting cards representing commands into rows of slots, which represented programs. Not only could one see and directly manipulate the programs and their pieces, but also the sequential activation of

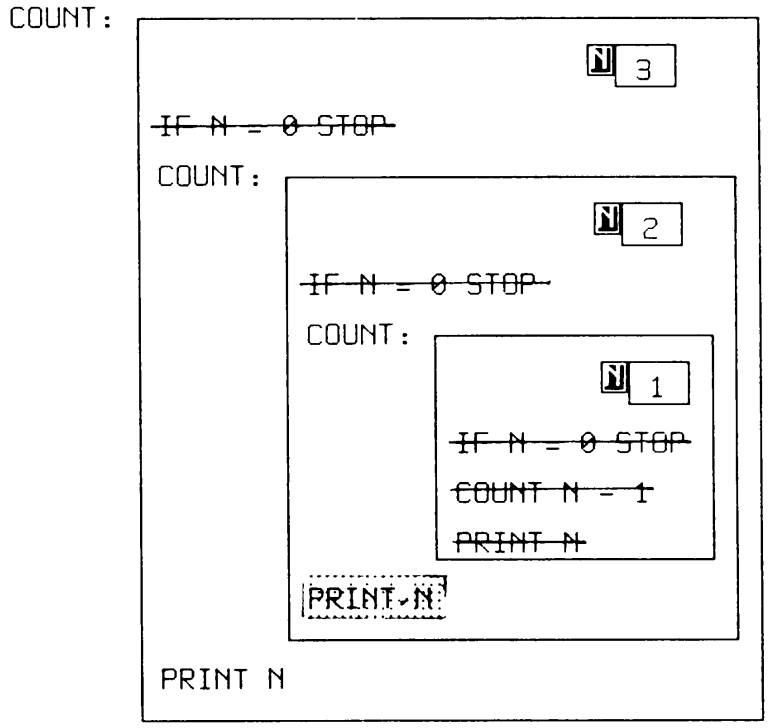


FIGURE 6.2. The display of a procedure stopped in midstream.

program steps, subprocedure calls and returns, could be directly observed as lights lit up under each card when that step was executed. Apart from avoiding typing, and adding concreteness to computational objects, the slot machine provides substantial help in making a mental model of the operation of a computational system like Logo.

With the advent of high resolution bitmaps and graphical objects like sprites, one can easily implement a slot machine on the video screen, moving icons around like cards with a joystick or mouse. The screen slot machine can be freed of many of the limitations of the physical one. On a screen, it is easy to invent spatial/graphical representations for general input parameters and conditionals which did not exist on the slot machine. Adding symbols, e.g., by typing a new word, is trivial in the screen version whereas making a new, physical card is not so easy. Most important, the process of abstraction can be represented easily on the screen—e.g., a spatial sequence of card-icons forming a program can slide together over one another like a spread out deck of cards being pushed back together and be given a top "cover" icon, becoming a single unit like all the supplied primitive card-icons. There does not seem to be any reason that nearly the whole of a computer language like Logo could not be presented in this graphical, concrete way. Though it has limitations that will be discussed in the section on direct manipulation, it is a very attractive introductory presentation to programming.

In passing I note an interesting theoretical issue. This iconic form of programming represents an attempt to use spatial and object knowledge to replace the linguistic means Logo uses to promote comprehensibility (see Chapter 10). Looking at the set of bugs and difficulties students have with such a system as compared to present Logos should provide an insightful comparative study.

STRUCTURE

The second type of change in programming is change in *structure*. This goes beyond the presentational changes mentioned above and significantly alters the underlying computational mechanisms and structures. While it might not be apparent to the user because of syntax or other presentational issues, the structure of a system provides some bedrock characteristics.

Traditional motivations for changes in structure have been increased modularity, preventing and catching bugs, precision of expression, mathematization of programming, and the like. In view of mounting an assault on the utility barrier, priorities become reordered. Looking back at the ratio of value to effort, one comes immediately (in the numerator) to expressive power—how quickly and simply does the language describe situations users can immediately perceive in terms of existing goals and needs. And one comes (in the denominator) to understandability. To be sure, ease of use plays a role in defining the work needed to accomplish something, but I have been constantly impressed with how much effort humans will expend if they value the result and understand what they are doing. In this section, I largely ignore ease of use. As long as it is not confused with understandability and perceived value, ease of use may decide which product succeeds in the marketplace, but not which paradigm of computation will succeed. (Compare Norman's remarks on first-and second-order issues in Chapter 3.)

When it comes to expressive power, one must talk about the tasks a user wishes to accomplish. Structure can be important. While all programming systems may be formally equal in power (Turing equivalent), some may be radically better adapted to some purposes than other systems. However, I defer discussion of this to the section on context, except where it is unavoidable.

When it comes to understandability, the structure of a language does not stand in isolation, but relies in general on presentation as a major channel of communication to the user. Thus the shift from print- to heavily graphics-based interfaces can precipitate a shift toward visual–spatial programming languages, as exemplified by the icon

programming system. But there, no change in structure was implied. With Boxer, however, we have found many small, and some not so small, ways that the structure of programming can be profitably changed to mesh with new presentational means, to which we turn briefly.

Structural Innovation in Boxer

Variables are different in Boxer than in any other languages for reasons of understandability and usefulness. The meaning of variables in Boxer includes the familiar set and fetch protocol, but variables are turned into genuine places in which different things can be stored. The place metaphor cannot be supported very well by the simple association of a name with a data object. For example, a Boxer variable can be shared not only by having another procedure use the same name, but also by an actual reference to the place of the variable. This reference is what we call a *port* in Boxer. Even if you change the name of the variable, the program will still share its value through having a port to it. Ports as part of data objects give a hyper-text functionality. This is as valuable concretely, as it is in programs. For example, it allows one to create cross referenced and non-hierarchically organized documents. One can think of ports as analogs of traditional pointers, except objects shared with pointers do not have a unique place of existence like the target of a set of ports. Instead, objects shared with pointers belong equally to all owners of a pointer to them. To get a grip on what this slight change in structure means to comprehensibility, imagine what a bizarre world it would be if possession were not largely synonymous with physical location, if several people could hold, look at, and even change the same object at the same time. (To be fair, Boxer does not eliminate this kind of thing. But it makes it an advanced topic rather than an entry level one. One needn't program with ports. At the same time, the issue is better presented visibly.) Figure 6.3 shows the display of a port. Though in this case the target of the port is on screen, that need not be true.

FIGURE 6.3. A port and its corresponding target box.

We could continue the list of subtle, but not insignificant differences between Boxer structures and those of other contemporary languages. For example:

1. Boxer does not have the same QUOTING mechanism as Lisp, Logo or Smalltalk, but instead has one with a simpler interpretation. The equivalent of quote is a type marker, *data*, and data objects are simply unevaluated by the interpreter. The quote is not stripped in evaluation.

2. Boxer does not distinguish intrinsically between atomic and compound data objects in order to eliminate a class of type bugs beginners have.

3. Actor-oriented programming with class and subclass hierarchy is not primitive in Boxer as it is in Smalltalk, but can be built with box structure.

Each of these structural changes was motivated by utility, making the language simpler and more powerful. But they are not dramatic. I don't believe Boxer will succeed or fail on the basis of its innovative structures—presentation and context (to come) are its strong points.

Beyond Procedural Languages

So far only procedural programming languages (Basic, Logo, and soon, we hope, Boxer) have made significant inroads with nonprofessional programming. To be sure, spread sheets border on offering general computational power in a new form. See Kay's article (1984). But at this stage it is more tease than substance. Visicalc is more than usually flexible for an applications program, but not yet a really programmable medium. Actor-oriented programming, as represented by Smalltalk, has some significantly different structures than run-of-the-mill procedural languages. But, Smalltalk's strengths are really at the scale of systems—it is the first genuine example of a fully integrated medium taking advantage of high resolution graphics. Its aim as a personal dynamic medium puts it squarely in the line of development of interest here. But, by and large, the designers of Smalltalk have attacked the complexity barrier rather than the utility barrier. The basic computational mechanism is more complicated in Smalltalk than in Logo. Actors, classes, and instances are undisputably useful, but they are levels of organization on top of functions and variables, not simplified replacements. The things one can do quickly and easily with the basic

system and unprofessional skills are not advanced enough to be a popular medium in the sense of this chapter. Thus, Smalltalk is more a meta-medium than a medium.

Prolog is also not a breakthrough in understandability. Its logical semantics seem quite orthogonal to advances in visual presentation that motivated Boxer and, in a different way, Smalltalk. But Prolog really does offer a potentially dramatic shift in the kind of thing done with programming, so it will warrant a second look later.

Let me turn to a "new," as yet to be fully defined, example to illustrate another class of near future possibilities for changes in structure.

Device Programming

Device programming is motivated by the image of an electronic or mechanical device consisting of a number of components of a few classes, like resistors, transistors and capacitors; or pipes, pumps and reservoirs. These components each have relatively simple behavior and achieve the functionality of the device by being hooked together at their terminals into a network. Computationally, we want to have graphical components that can be manually assembled into devices by connecting their inputs and outputs with "wires" (lines drawn on the screen). The wires communicate messages of an arbitrary symbolic kind, which could, for example, represent flow of substance or electricity by passing numbers representing amounts. Each component knows when it gets a message at an input and can compute and send output messages as it sees fit. Device programming is structurally a significantly different form than contemporary procedural languages because of its explicitly parallel nature of computation, and its explicit representation of data flow rather than sequence. On the other hand, device programming can simulate a function as a component with one input and one output. Activation of the function amounts to simply giving it an input. Furthermore, a component can be built out of very little more than a procedural programming language in which to express the actions to be taken to compute outputs from inputs.

Naturally, it would be important to have a general abstraction mechanism so that a network of devices could be made into a component. Some set of free inputs and outputs in a device could "extend beyond the boundary" of the device to act as terminals of the abstracted component. Visually, the parts of the device-become-component can shrink and/or acquire a new surface form to hide detail. Likely one would like the surface form to show some small part of the internal state of the device.

Device programming is attractive because it has such a simple and graphic method of combining elements to make compound things. There is reason to believe it can have some intuitive accessibility that the hidden data flow and complex sequencing of pure procedural programming does not. Lastly, it opens doors to more easily simulating an important class of physical computations, thus aiding in our understanding of the world. One can even engage in the delightfully recursive task of constructing a computer out of computationally implemented components, emulating every level of abstraction of a physical computer. All of these characteristics of device programming—intuitive accessibility, important and interesting application, and even ease of use—are illustrated in the contemporary computer games *Rocky's Boots* and *Robot Odyssey*.

Device programming, like most new ideas, is not entirely new by any means, but a crystallization in a new context of a collection of old ideas. In fact, the first system I know of that substantially followed this outline was made 20 years ago by Sutherland (1966). The reason the idea is timely again is that powerful processing and high resolution graphics, along with the particular concern for creating an easily accessible popular medium, define a niche into which device programming may well fit nicely. It is interesting to remark on how this niche redefines past conceptions similar to device programming. UNIX pipes, by their very name, suggest the right topological metaphor. They are also a parallel processing system. However, pipes are structurally one dimensional and presentationally nongraphic. They provide no easy opportunity for output, let alone input, at intermediate stages of the pipe. Pipes are not intended to be a means of implementing programs on a small scale, but rather, they are a way of combining existing rather large chunks. Most other parallel processing constructs, *Simula* coroutines for example, are at best intended to control graphical objects, not to be graphical, nor do they make use of device topology to define the communications network in a program.

CONTEXT

Continuous incremental advantage. The third dimension of change in programming is context, what you do with your programming system. Turtle graphics, whereby drawings may be created by issuing commands to a mobile graphics cursor, is a crucial part of the advance of Logo over previous languages. It is motivating; it allows children to set goals immediately that they understand (drawing pictures), yet it

can evolve naturally and slowly into a medium of contact with profound mathematics (Abelson & diSessa, 1981; Papert, 1980). Again the old story of natural language tells the tale. It is an incredibly complex and large learning task which, nonetheless, nearly every child masters because it can be mastered one tiny bit at a time, and is useful to the child at every step along the way. The steps are small, but the range is large: A child gets a cookie by learning how to ask; a university professor gets recognition not only for his ideas, but also for presenting them well. The principle really deserves a name—*continuous incremental advantage*. If we want anyone to master any complex but powerful tool, it had better offer continuous advantage to the learner for learning more, and the bits of learning had better be in manageable (incremental) chunks. The importance of the principle is not only having small steps in learning, and motivation for those steps, but so that at each stage the learner gets structured feedback on competent performance, feedback in terms of achieving understood goals.

In learning Logo we have found some blocks, or at least apparent plateaus in continuous incremental advantage (Papert, diSessa, Watt, & Weir, 1980). For very young children, procedures seem to constitute a small barrier. They are more difficult than necessary to master, and don't do all that much beyond packaging the picture the child is drawing. Many children prefer to type out a stereotyped set of commands over and over rather than to make them into a procedure. This is one case where more work is done to avoid a not firmly understood but more efficient process. For older children, procedures are more productive, but variables seem to stall them for a while. Effective use and understanding of list processing has similar problems.

Table 6.1 shows how Boxer compares to Logo with respect to transparency and incrementality in the important early activity of making a definition.

Data objects can be built in a similar way and turned into a variable simply by adding a name. The value of the variable can be changed at any time under program control, or directly with the editor.

Boxer Data Worlds

With regard to continuous incremental advantage, Boxer's concreteness not only provides smaller, more understandable increments, but it also enlarges the scope of programming's context to include text production and manipulation, and the organization and manipulation of many other sorts of data. If a programming system is literally also a child's book and pencil (text editor in modern parlance), and if he can, bit by bit, modify, extend, and personalize not only what comes in his book, but

TABLE 6.1
COMMANDS

Logo	Boxer
1. Type a set of commands to try them out.	1. Type a set of commands, or select and collect from previously typed text.
2. Type TO <name>. Screen goes blank. You're in the editor.	2. Mark the commands (push a button and draw pointer across). Then press a key to make a box.
3. Recall or type over from pencil and paper the list of commands tried out.	3. You may now point to the box to execute it.
4. Exit editor.	4. If you like, add a name to the procedure.
5. Type the name to execute.	5. Type the name to execute.
	6. If you like, shrink the box to hide its contents. Or you may move it entirely off the screen which is the currently visible part of the Boxer world.

also the form of the presentational medium, then programming becomes a learnable-in-tiny-increments and constantly useful extension to written communication, something with which children are in constant contact. A simple example of such modification is to add a new editing command, or to use a variable as a means of keeping around a template for electronic mail messages or other "forms." Hierarchical structure, boxes in boxes, which Boxer makes so prominent and easy to generate, can be used by children to organize and reorganize their personal computational world (Figure 6.4). Note that there need not be any such things as files and filing commands. Such use may seem trivial in view of the the power and subtlety of variables and hierarchical structure in the hands of expert programmers, but simple instantiations provide for continuous incremental advantage. More advanced use of Boxer's degree of integration are also possible. Because everything in the system is a computational structure, computing on any text (say, computing a reorganized format for a notebook) is simple to arrange.

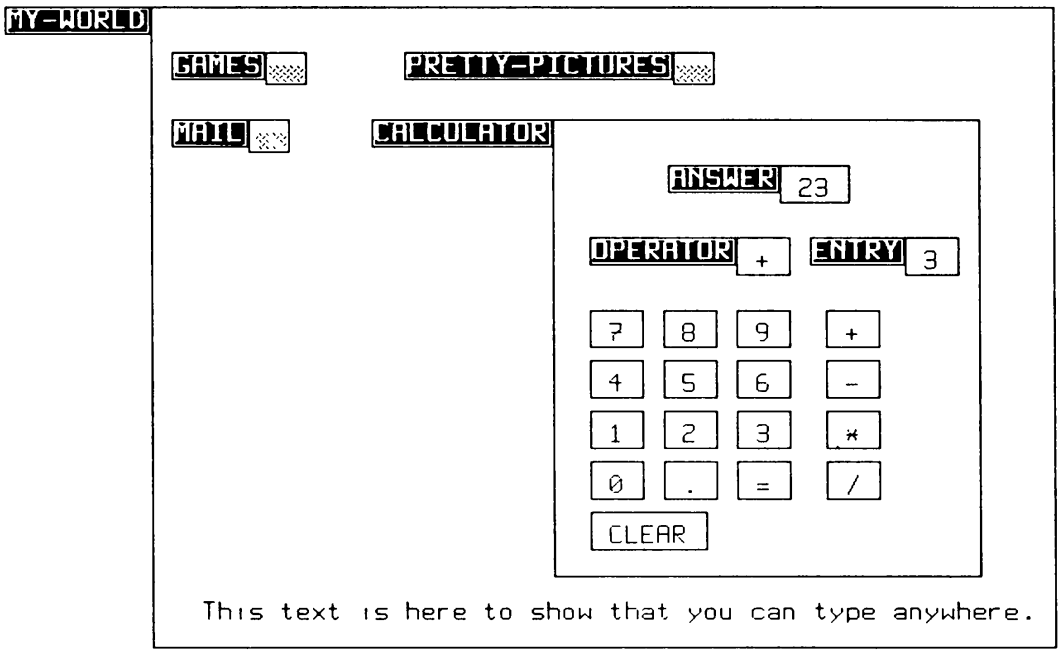


FIGURE 6.4. The top level of a Boxer world. Small gray boxes have been shrunk to hide detail. A subbox may be "entered" by expanding it (with a keypress) to full screen size.

In comparison to iconic programming or programming by direct manipulation, Boxer's text orientation may seem conservative, if not reactionary. But the main reason we chose to move in that direction was to promote a synergy between the written word, including all the incremental advantage it offers, and the advantage that programming and programming structure offers to written communication. Our judgment is that text will not fade away as a dominant medium, but will be transformed and improved by computation and programming.

To solidify this notion, consider a thought experiment. Imagine Boxer as a future publishing medium for educational materials. A student might buy a digital optical disc containing a Boxer book. Figure 6.5 shows a "page" (box) from that book. Box structure allows hierarchical presentation with details suppressed at each level by shrunken boxes. The table of contents of the book may *be* the book, viewed from the top level. Within a "section" (box), graphics boxes provide illustrations. Graphics boxes are like most boxes in that they can be made with a keystroke, expanded and shrunk, moved around or deleted like a large character. They are computational objects. They can be named, ported to, and referenced in programs. But graphics boxes also come equipped with a set of primitives for making and modifying pictures. As an example, generalized turtles—movable, touch sensitive (to the mouse and to each other) graphical objects—may be created in graphics boxes. (A paint program in Boxer is not

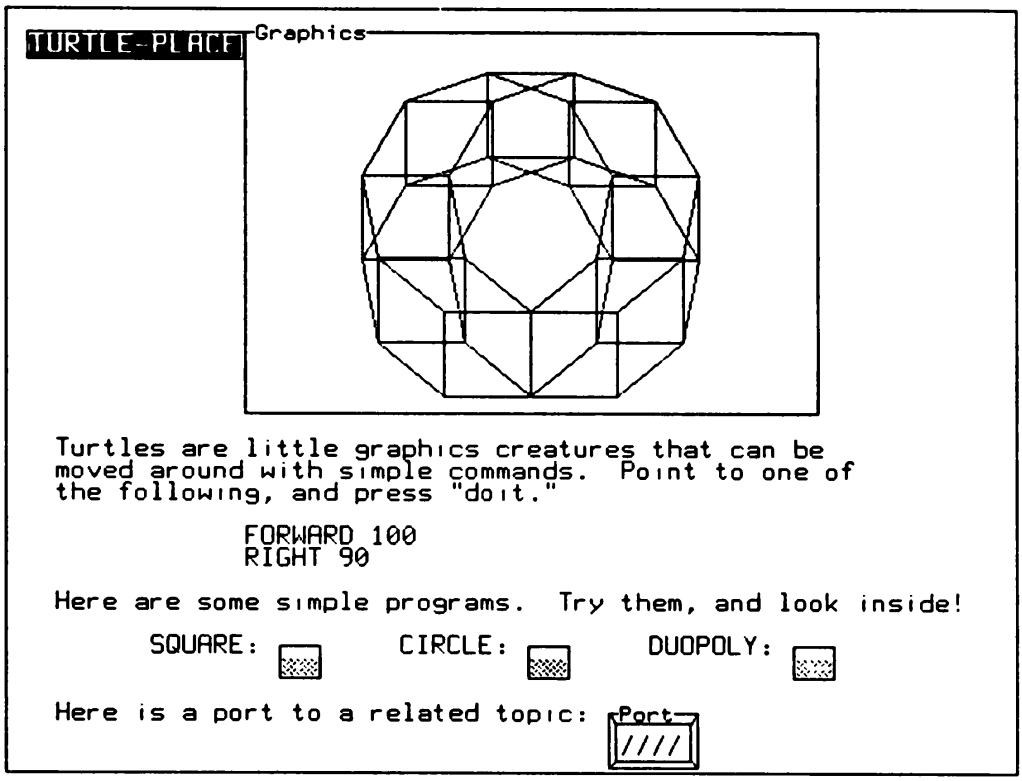


FIGURE 6.5. A page from a Boxer book.

viewed as defining graphics capability, but as an optional program written to simplify construction of a certain kind of data object, a graphics box, the way one might write a program to simplify construction of programs or more familiar data objects such as records.) So the illustrations may easily be dynamic, controlled by programs written into the text. The simulation programs, of course, are inspectable, and changeable, so students may experiment freely, learn from the way the simulation is written, and they may even clip pieces of the program out of the book for their own purposes, such as game program writing.

Citations to other sections of the book may be made through ports. Students may write annotations into the book. They may provide their own summary view of the book by collecting ports to selected sections into a box, together with their own notes. Any tables written into the text are also automatically data bases, available to be used for further analysis or computation.

Note how much work it would take to produce such a book. Basically, the author would just type in the text (including box organization), assemble pictures and type the programs that control the graphics or that provide particular facilities not built into Boxer. Instead, or in addition to programming, the author might hire a programmer or take programs from public domain Boxer books.

It is easy to imagine teachers making fragments of Boxer books, dynamic worksheets, in the course of a day's work. Undoubtedly, professionally produced books would be published containing fragments of programs and ideas the same way worksheets or lab kits that constitute a limited, but potentially very significant, part of today's learning culture for teachers.

To the level that I have described this hypothetical book, the standard Boxer interface is the means of construction, and, as well, the means of consumption and reconstruction. That this is true is an important characteristic of the kind of medium we want to have. One can step easily from the role of consumer to that of producer. The fact that Boxer has this property is in large part due to its visibility and principle of naive realism. Constructing an interface to an application is often trivial because, for example, output from a program can be produced simply by putting a program variable or port to one on the screen. Input can be equally trivial, since typing into that variable changes it.

The constructable interface. Naturally, for special purposes, one may want input and output farther from the default paradigm. We are in the process of designing a slice of Boxer that I call the constructable interface through which more specialized interactions can be made with incremental programming. Indeed, this is a new context of continuous incremental advantage for programming. In the past, programming environments like Logo, Basic, and Pascal have been conceived of as relatively or absolutely fixed, not seriously user adjustable. In Smalltalk, the user interface is totally redefinable, but the cost of this flexibility is that fiddling with the interface often can involve fiddling with complex, highly refined system code, or it might be a huge project of supplying a complete replacement for that code. Our hope is to steer a middle course that avoids these extremes and provides more realistic incremental advantage.

The constructable interface in Boxer at present has two parts. The first consists of "bells and whistles" and adjustable parameters on boxes with user interface functionality. For example, keystrokes and mouse clicks can be redefined to run arbitrary Boxer functions when performed within a given box, boxes can have demons associated with them to take definable actions on entry or exit of the cursor, and boxes can be frozen at any size (overridable by explicit "expand" or "shrink" commands) to allow selectable display of the interior of a box, not just all or none.

With definable mouse clicks alone, it takes about 10 minutes to make a screen-based calculator from scratch, where you point and click

to press the buttons (boxes). An arbitrary amount of the internal structure of the calculator, registers and so on, can be shown. The surface form of the calculator shown in Figure 6.5 involves no graphics at all; it was simply typed in using the Boxer editor. More generally, we imagine users building or modifying little Visicalc-like interfaces for doing jobs like data presentation (graphing) and analysis (easy entry and sorting of data fragments).

The second part of the constructable interface implements more extreme changes in the interface. For this we use the interior of graphics boxes, which structurally do not allow precisely the same "text editor" interface as the rest of the system in any case. Thus, the interior of a graphics box will not have the very well-elaborated and universal interaction of the rest of Boxer, but default behaviors and a few appropriate high-level primitives to build interactions. Examples of the former are: (a) Graphics objects are automatically highlighted (selected) as the cursor passes over it; (b) clicking "expand" on an object expands it to show the data that define its state (position coordinates, special procedures to define behaviors like animation, or other change of visible presentation); (c) any Boxer code can be activated on selecting an object and pressing a mouse button or key. Examples of the latter include the process of following the mouse cursor, changing default behavior, etc.

I would like to close this section with two small, less parochial notes on the important topic of context. The first is about Prolog, or, more generally, about logic programming. Much has been made of the elegance and conciseness of such languages in expressing certain facts and relationships in a form that is executable. Just as much has been made about this as a proper way of using the high performance, multiple-processor hardware that is soon to be available. In relation to the complexity barrier, this is appropriate. In relation to the utility barrier, it is less so. I have already stated that Prolog does not obviously accept present technological advances that can improve understandability, such as integration of visible properties with semantics (e.g., Boxer's spatial metaphor) or presenting images of the computational mechanism in process. Arguments have been presented, in fact, to the contrary: that the invisibility of mechanism and relative uncontrollability of Prolog makes it difficult to program and, especially, to debug. But Prolog offers the glimmerings of a substantially different and exciting context for programming.

In Prolog, one makes relational assertions in a data base such as "John loves Mary," or "John is-a man." Rules of general value may be written, such as "If X is-a man, X is-a person." Then the data base may be queried by making variabilized statements like "X is-a person,"

which returns all values of X that make the statement true. Thus logic programming is in substantial ways about expertise, and the hope can be that Prolog or future versions will allow incrementally learnable advantage in "expertise about something" over common-sense reasoning or noncomputable "writing down facts and generalities." Ennals and Kowalski are pursuing this possibility where children may, in the end, play with and even write their own tiny expert systems (Ennals, 1983; Kowalski, 1979).

Finally, to close an open loop, anyone who develops a general device programming system may well provide an exciting new set of contexts in which to learn and appreciate programming. (This is not to slight present versions, like *Robot Odyssey*, but their aims are quite limited with respect to being a general medium.)

DIRECT MANIPULATION—WHERE'S THE PROGRAMMING?

The concept of direct manipulation as a future kind of programming has acquired a good deal of support in recent years, particularly because of its more attractive features in simplifying the programming process. (Chapter 5 by Hutchins, Hollan, & Norman discusses these hopes in its first sections. Readers would be advised to look over those sections before continuing here.) I would like to comment on the status of these hopes.

The plan for this section is to understand direct manipulation by situating it along the three dimensions of change in the meaning of programming: presentation, structure, and context. The overarching question is what exactly does direct manipulation have to do with programming as defined here? By and large, I shall assert that direct manipulation does not define or even suggest any major change in computational structure or context beyond present programming. So my major discussion will be on how direct manipulation can re-present programming. We shall find that direct manipulation has difficulty with particular aspects of functionality intrinsic to programming. There are limits that may not be evident at first sight on what we can expect direct manipulation to do for programming. The section concludes by rediscovering direct manipulation in what, given the perspective on the future of programming in this chapter, is a proper relation to programming.

Direct Manipulation of Standard Programming Objects

To begin, it is important to distinguish between tasks accomplished by direct manipulation. The task might be to operate on computational objects, which might be more or less conventional objects in a programming system. If this is the case, then what we are talking about is not a change in structure, but a change in presentation. Boxer is a good example of a system that permits direct manipulation on more or less conventional computational objects. Boxer makes all computational objects visible and manipulable in a uniform way. Adherence to naive realism assures that the visible representation and the underlying reality are minimally disparate. And the uniform text representation of all objects assures a rich manipulation language (the Boxer editor). Visible reality and a rich manipulation language bode well for a successful direct manipulation system in the straightforward sense of those terms. Naturally, I see an important future to such systems. But Boxer is not what direct manipulation advocates have in mind. It is structurally more or less a garden-variety, general purpose, computational system. It contains a lot of "code," and has such things as syntax errors.

One may, of course, try to turn procedural structure more or less directly into nontextual forms, like iconic programming, slot machine style. But in consideration of a simple-minded "conservation of complexity," this tack will have limited success compared to the prospect of eliminating error and providing instant accessibility to first-time users. Arguments below detail other limitations of this general line.

Manipulation as a Presentation of Programs

On the other hand, the task accomplished by manipulation might be not to construct computational objects, but to demonstrate or represent them. This is a considerably more optimistic image of what direct manipulation might bode compared to replacing words on a one for one basis with icons. But, again, this does not necessarily offer substantial changes in structure, but only presentation. If one considers paradigms such as procedural programming, logic, and constraint-based programming, then it appears that manipulation as a programming language belongs firmly in the procedural camp: The basic programming metaphor is demonstrating a sequence of actions.

How much of the procedural paradigm can direct manipulation carry? Here, the programming language community is in the midst of substantial research. One can point to such examples as Gould and Finzer's programming by rehearsal (Gould & Finzer, 1984), where

routines are taught (programmed) by manually running through the sequence of actions that one wishes to teach. The actions are essentially all selecting from among preprogrammed actions of a troupe of computational actors. The system gets much of its character and power from the set of initially supplied actors. In its present state of development, the system doesn't feel like a general programming system; not even its implementers chose to build the supplied troupes with the system itself. My suspicion is that this will not give those who learn only the rehearsal level of the system enough flexibility for it to be widely adopted as a style of programming, that is, a reconstructible medium.

Halbert's (1984) programming by example system and Lieberman's Tinker (1984) attack more directly some of the fundamental problems of direct manipulation as a representation of programs. To understand what these are, we need to look at some of the fundamental functionalities that define procedural programming. This will put us in a better position to make general comments than examining particular systems. It also seems appropriate in a paper on the future of programming to say something in detail about what the essence of programming is, and what are the invariant structures we will find, in one disguised way or another, in any near-future programming system.

I will use a description of a general computational system provided by Newell (1980, recharacterized and slightly reorganized). Table 6.2 shows a taxonomy of functions. The four major areas, abstraction, representation, computation and I.O. appear in column 1. Column 2 contains the operators Newell uses to exemplify key subfunctionalities, and column 3 contains my characterizations of the subfunctionalities from column 2. The bracketed enumerations mark functions that may be problematic for direct manipulation, and they are discussed below.

Abstraction. Abstraction is the first functionality. In its simplest interpretation, abstraction means elevating from instances to classes. Variables and the literal/computed distinction are very simple mechanisms that provides much of this functionality to contemporary programming languages. But these pose a fundamental problem for direct manipulation. How do we express in the manipulative paradigm, which selected items are to be regarded as instances, which as classes, and along what dimension of generalization are we to define the class, if variabilization is intended? Much of Halbert's work is to solve this problem. Even this work assumes that the objects manipulated are the items to be abstracted, rather than the potentially much more difficult problem of abstracting on procedures. How does one say, "Look, do what I just did, but such and such a part of that was just an example of what might be done."

TABLE 6.2
TAXONOMY OF FUNCTIONS

Abstraction	*Quote*	controlling reference; variable/ literal distinction **[1]**
	Assign	naming **[2]**
	Copy	supporting isomorphism as well as identity
Representation [3]	*Read* symbol at position *R* from an expression	fetch
	Write to position *R* in an expression	mutate
Compute	*Do*	sequence
	Exit if ...	Control **[4]**
	Continue if ...	
I.O.	*Behave*	output
	Input	input

Naming. Abstraction also means having complex entities as units, substantially functioning objects in the system. Naming is at the core of this functionality. Here, direct manipulation is on a slippery slope. Pointing can be quite effective in some circumstances, but in order to have broad access to many entities, which is often needed in present computational systems, one would like to have some easily reproducible symbol. Once one allows this, however, text becomes such a powerful competitor, that it is unlikely any gestural or other method of specification could effectively compete. After all, text evolved over eons to serve precisely this role. Once one allows textual names, and has any sort of written representation for programs, then it seems to me a text-based programming is not far behind.

Even a seemingly strong point of manipulation, demonstrating sequence, begins to appear problematic without suitable abstraction mechanisms like mnemonic naming. For debugging purposes, reproduction of the gestures made to demonstrate a program is a very weak replacement for all the intention and context that existed in the head of the direct manipulation programmer. Not being able to use the expressiveness of language within a program seems a dubious improvement. Not being able to annotate beside the "code" seems unnecessarily

restrictive. Not having a visible notation seems fatal. Debugging was a weak point of an early and ambitious piece of work that mixed manipulation with static program structures to represent programs (Smith, 1975). For reasons of debugging alone, I have much more hope for a hybrid text and direct manipulation system than for any attempts at pure manipulation systems.

Symbolic representation. If one thinks of representation (in the sense of building complex objects that model noncomputational systems through attributes or assertions) as a separate functionality, then again one seems to slide directly into symbolic presentations for the same reasons as for naming. It is plausible to construct these representations and write programs that manipulate them by direct manipulation, but easily reproducible symbols and variabilization are still issues. A strongly hybrid system is again the likely outcome.

Control. Control is a serious problem for manipulation. I know of no workable gestural indication of conditional branching. Compared to a simple IF ... THEN ..., it seems unnecessary to try. So once more, it seems some symbolic representation enters into the system with this functionality.

Direct Manipulation as a New Context

As far as contexts are concerned, direct manipulation does not by itself suggest anything new. It, of course, motivates a particular class of domains, in particular those where substantial interaction or processing can be assembled by organizing visible units. But that, by itself, is a weak heuristic.

Where have the high hopes for direct manipulation gone? Could it be that direct manipulation has nothing much to do with programming, but is instead a very general, thus weak, heuristic for constructing interfaces, or a trend in applications programs to draw icons in high resolution, to allow pointing, poking and moving those icons around in order to do a few things, but certainly not to program?

Let me sketch a class of ways to achieve most of what I believe realizable from direct manipulation while maintaining a true programming medium. To start, we would like a system which can support an "object metaphor" in the sense that graphical objects with a standardized set of manipulations on them such as *copy*, *move*, *connect* and *recognize a pointing operation*. This is meant to be the surface level of most "programs" written in the system. However, beneath this level must exist a full computational system to define the semantics of the objects and specialized meaning of their object manipulations. For example, the

objects could open into data structures, programs or complete environments that contain multiple programs and data. So far so good, but the key is to establish a rich set of connections between the computational semantics and the object manipulations so that object manipulations are, in fact, reconstructing the computation. One must be able to bind interface operations to computational ones: touching to activation, connection to data flow, copy to copy, movement to context switching.

If all this sounds familiar, it should. This is precisely the intent of the constructible interface of Boxer, particularly its graphics box component. So, what I have in mind is opening up graphical objects, the generalized turtles described earlier, into Boxer structures that define them. Some of the connections between interface operations and computational ones are already in place in our current implementation. So the plausibility of these arguments on the relation of direct manipulation to programming will soon have a strong test. If we can do our design responsibly, Boxer should become, in part, a medium for building, using and modifying what everyone will recognize as direct manipulation systems.

Summing up. Direct manipulation does not appear to offer any substantial new structure or even, of its own, any new context for programming. Even as a different presentation of procedural or actor-based forms, there are several areas of functionality where text or other symbolic presentations are so strong, that at best we should expect a general purpose language to be strongly hybrid in ways akin to the relation of Boxer to its constructible interface. Indeed, I believe that direct manipulation of symbolic computational structures will become standard, especially for nonprofessionals. But except for a few very special cases, I do not believe direct manipulation will soon become even an acceptable substitute for symbolically presenting or representing computational structures.

I cannot claim any finality to this brief critique of direct manipulation. The line of argumentation presented here is subject to at least three criticisms, which I will very briefly counter. First, the functionalities abstracted may be abstracted too directly from present day programming so that the conclusion that form is hard to change is essentially built in. Independent of details, however, I believe these functionalities are indicative of a class that could be the basis for refined argument. Second, one may ask what difference it makes if direct manipulation is not programming. Flexibility and other virtues that we need in a medium are not synonymous with programming. Here, the more experience with particular systems, the better. When we have examined the empirical constraints on flexibility, etc., of many systems,

I think it may well be easier to convince the direct manipulation enthusiast that those problems fall into categories like those above.

Finally, invention has a way of revamping our assessments of possibility. Two examples of invention that might have impact here are the advent of quite substantial machine intelligence and radically enhanced input devices. Intelligence might, for example, figure out a programmer's intended level of generalization in pointing to an object, and represent it, even for the programmer's use, in other ways. Gestural input at a level significantly beyond pointing and poking (in concert with voice, intelligence, etc.) might give new meaning to manipulation. In comparison to the rich manipulation language of the physical world (consider a watch-maker's or a machine tool-maker's craft), the manipulation available with present machines is a travesty. See Buxton's contribution, Chapter 15. But these innovations are quite beyond the conservative, near future stance I took in this chapter.

SUMMARY

This chapter centers around the notion that a cluster, including ease of use, comprehensibility and perceived value—in short, utility—will be a major factor determining the extent to which programming will enter the lives of most people in the future. Although most chapters in this book deal, appropriately, with questions of "how" with respect to interfaces, I have attempted to bring to center stage the prior question of "interfaces to what?" The image of computation as an interactive medium of unprecedented breadth and power, with programming as the means of construction and reconstruction in this new medium, establishes a context in which we may set our goals and judge the results of our work.

The concept of utility that must be employed in this context is not simple. To succeed in enticing individuals and society at large to learn a complex and subtle device, that device must offer "continuous incremental advantage," motivation to take each step forward at each stage and at each level: from day one, to expertise; from immediate gratification to the noble goals of intellectual advance of civilization. In this regard, no individual or group of designers can pretend to substitute for the experiences and judgment of a hugely diverse society. But, while computer professionals are rapidly crystallizing computational environments out of their needs and aesthetics, few are attempting to give nonprofessionals a chance to experience a medium of minimally constrained possibilities. We must do better.

In changing programming from its present to future forms, we have three major dimensions of change at our disposal. We can alter the way

we present computation to the user; we can alter the structures of computation themselves; or we can alter the contexts of application of these structures to do different things. While they are conceptually distinct, these are not independent dimensions. Each can influence the other. Means of presentation may select different structures as optimal; different structures may make new applications possible; and important applications may carry with them suggestions for presentation.

This chapter has looked at a span of near future changes to the meaning of programming. This yields some expected conclusions, and some more surprising. Programming is a relatively complex task that is not likely to become trivial through any near future change in any of these dimensions. On the other hand, there are several promising moves we can make that offer improved utility with changes along all three dimensions. I believe we can look forward to an exciting future for programming as a popular medium that can extend the reach and grasp of us all.

ACKNOWLEDGMENTS

This paper contains many ideas that have been developed and honed by a group of people associated with Logo and Boxer, a group too large to enumerate. Discussions with Brian Silverman about device programming have been provocative. The constructible interface in Boxer is being developed jointly with Jeremy Roschelle. The discussion of direct manipulation was stimulated by suggestions by Don Norman, Ed Hutchins, and Jim Hollan. The presentation of this chapter has been improved by suggestions from the Asilomar group, particularly Bill Mark, Don Norman, Steve Draper, Clayton Lewis, and Mike Eisenberg.