10. Creating Software Applications for Children: Some Thoughts About Design

Michael Eisenberg

Department of Computer Science and Institute of Cognitive Science
University of Colorado, Boulder, CO 80309-0430 USA

Abstract. The last decade has seen an explosion in the number, variety, and complexity of software applications. While this phenomenon has been most pronounced in the world of professional software (i.e., commercial software aimed at adult professionals), it is no less true in educational software. Moreover, the trends visible in commercial application design—notably, an increase in interface complexity, and proliferation of narrowly domain-specific packages—are likewise visible in the educational realm.

This paper makes several arguments about the design of such "high-functionality" applications for educational purposes. First, it argues that these applications should be programmable, incorporating elements both of direct manipulation interfaces and of programming environments. Second, it argues that there need be no firm boundary between "professional" and "educational" applications. Third, it argues that a variety of "scaffolding elements," such as software critics and example-based catalogs, are useful experimental techniques with which to enhance the learnability of educational applications.

The paper presents three working prototypes of programmable applications, for the domains of graphics, computational physics, and charting, respectively. All three are based in the Scheme dialect of Lisp, and collectively these applications illustrate methods of putting the above-mentioned design principles into practice.

10.1 Introduction: The Advent of Educational Applications

Over the past decade, a quiet revolution has occurred in the culture of educational software. In the early 1980s, the most visible and spirited debates in the educational computing community focused on a few central questions—e.g., whether drill-and-practice programs were pedagogically valuable (cf., Solomon, 1986, ch. 2); or how best to incorporate cognitive models and artificial intelligence techniques into edu-

cational software (Sleeman and Brown, 1982); or whether (for instance) BASIC or Logo was the best programming language for children to learn (Luehrmann, 1982). While none of these issues has disappeared, the traditions of software design that they reflect—computer-assisted instruction, intelligent tutoring systems, and child-friendly programming environments—have collectively been challenged, if not eclipsed, by the advent of what might be called "educational applications." Programs such as *SimCity*, *SimLife*, *Interactive Physics*, *Kid Pix*, the *Geometer's Sketchpad*, and the *Explorer* series are representative of this trend—all might be viewed as tools, applications, or simulations aimed at elementary or high school students.

To be sure, there is a wide variety of styles and educational approaches represented even in this small set of examples. *SimCity* and *SimLife* are perhaps closer in spirit to "games" than applications, and in their nontraditional subject matter they are less "classroom-oriented" than other games (some examples like the popular *Carmen Sandiego* series are both more game-like and more classroom-oriented in style). The *Geometer's Sketchpad* is best described as a mathematical tool, while *Kid Pix* might be classified as a true graphics application for children. And of course these examples do not appear without historical precedents: Earlier programs like *Rocky's Boots* might be seen as conceptual ancestors of more recent educational applications. In any event, while a fine-grained history and taxonomy of the most popular educational software would be a desirable project, we needn't pursue the point at length here; for our purposes, we can identify several typical elements of educational software applications. These include: domain-specific simulation components (as in *SimCity* and the *Explorer* series); feature-rich direct manipulation interfaces (as in *SimLife*, *Interactive Physics*, and the *Geometer's Sketchpad*); and a focus (as in *Kid Pix*) on the creation or design of artifacts—such as drawings, compositions, multimedia "shows," and so forth—as opposed to the solution of software-posed problems. While none of these individual elements is a necessary or sufficient condition by which to classify educational software as "application software," they are representative of an overall "tool/application-based" style of work. And we can further identify these applications by noting a particular powerful element that they tend *not* to include: namely, a student-accessible programming environment (in the tradition of Logo (Papert, 1980) or Boxer (diSessa and Abelson, 1986)).

Despite the very real successes achieved within the educational-application paradigm—and despite the tremendous creativity evinced by the best programs—this last point may be seen as a troubling one. Some of the more ominous trends visible in the design of educational applications mirror those occurring in the world of commercial applications; and in both cases, these trends may arguably be traced to the absence of programming environments.

Consider, for instance, the burgeoning growth of application complexity in both the commercial and educational realms. A perusal of one's local newsstand, and

the collection of magazines devoted in large part to software advertisements and reviews, will confirm this phenomenon in the world of commercial applications. Typically, reviews or advertisements of new or updated software will focus on the description of ever-larger numbers of interface options or features:

> "XyWrite 4.0...offers improvements in performance and dozens of new commands and ease-of-use features." (*PC World*, May 1993)
>
> "Publish It! Easy is an entry-level desktop-publishing program packed with page-layout features you'd expect to find only in packages costing three times the price....[It] offers a wider range of paint and drawing tools than in previous versions...." (*Mac Computing* special issue, 1993)
>
> "The latest version of Theorist features several important enhancements....In addition to adding support for tables, the new version of Theorist provides QuickTime support....Other new features include support for Fourier transforms and Bessel functions, the ability to create scatterplots from tables and matrices, and the ability to customize the appearance of notebooks with different fonts." (*MacUser*, June 1993)
>
> "MacProject Pro is a significant upgrade to MacProject II, providing tools for better organization and entry of data, enhanced display capabilities, much-improved resource management, more-sophisticated scheduling features...." (*MacWorld*, May 1993)

It is in fact unusual to find an advertisement or review of a new release (or upgrade) of existing commercial software that fails to describe a marked increase in the number of features; and the same pattern of growing complexity can be seen in educational applications as well. The interfaces of programs such as *Interactive Physics* and *SimLife* are impressively (one might argue, dauntingly) large.[1] Even the original *Kid Pix* program—following the pattern of its commercial graphics-software cousins—has been followed by a subsequent release of add-ons to the earlier system.

In both the educational and commercial realms, then, software design is characterized by a spiralling growth of feature-sets. But while most reviews and advertisements describe this burgeoning feature-expansion in unhesitatingly positive terms, it is possible to view this trend with at least a certain degree of ambivalence. To take a specific instance, a recent advertisement of the well-known *CorelDraw* program noted that its latest release contains over 100 "feature enhancements" (*PC Computing*, July 1993). One might reasonably ask whether the fifth or sixth iteration of this upgrade process will result in five or six hundred such additions; and if so, how will users ever hope to accommodate this deluge of novelty? More pointedly, will these five hundred new features really alter or improve users' understanding of graphics, or, more likely, will the users simply be overwhelmed? These questions take on even more urgency in the realm of educational software, where

[1] For instance, the *Sim Life* interface, by a rough count, has over 300 menu choices and control options (and over 200 pages of accompanying documentation).

huge feature-sets lend an especially frantic and busy tone to applications. Rather than promoting a style of usage in which students can develop patient mastery over a creative medium, these feature-sets instead encourage endless moment-to-moment exploration of designer-supplied interface features. The result is that mastering the interface of one of these applications takes precedence over mastering the purported subject matter; one text on *SimCity*, for instance, devotes a substantial proportion of its instruction to techniques for succeeding in the game itself (e.g., how to "fool" the program into ignoring special patterns of taxation (Dargahi, 1991)) as opposed to ideas related to the "real" topic of city planning.

Yet another troubling aspect of application development that is shared by both the commercial and educational worlds is the somewhat arbitrary fragmentation of application software. In the commercial world, this phenomenon is illustrated by the large number of distinct software packages that come into existence for mildly varying tasks: A user interested in graphics, for instance, may own a package for creating hand-drawn paintings, another for creating architectural drawings, another for creating geometric drawings, another for creating flowcharts, and so forth. Similarly, in the educational world, the pattern can be glimpsed in applications such as the *Explorer* series (in which, e.g., "harmonic motion," "two bodies," and "gravity" units are all marketed as separate applications), and in the popular *Geometric Supposer* series (which likewise offers separate programs in quadrilaterals, circles, and triangles).[2]

Both these trends—the growth of feature-sets, and the fragmentation of packages—may be attributed in part to natural economic motives on the part of software designers. Periodic software upgrades, accompanied by the addition of new features, can be used to resell the same software repeatedly to a community of loyal users; and multiple variations of software applications can be employed to market a basic design substrate many times over. But there is another reason for these trends, rooted in the absence of programming environments that allow users to build ideas and create extensions for themselves. If a student can write programs to extend an application—using a modeling language to create a toy ecosystem, for example—then there would be little need for the gargantuan, feature-heavy interface of a program like *SimLife*; and arguably, the student in such a case would be more focused on the subject matter of ecosystem simulation than on the act of wandering through the wilds of the *SimLife* interface. If a student can use a set of geometric primitives as building blocks to investigate a variety of distinct topics, then there is no need for a fragmented set of *Geometric Supposer* units; conceivably, the same language substrate could be used as a foundation for interchangeable, and mutually enhancing, units on triangles, circles, and so forth—much as the *Mathematica* language can be extended with procedural libraries focusing on a variety of special-purpose mathematical topics.

[2] A more recent "SuperSupposer" version of this program does in fact offer these units in combination, while the original units continue to be offered as separate programs.

Again, it is worth mentioning that the best educational application software—*Explorer*, *SimLife*, and the *Geometric Supposer* included—is often nothing short of beautiful: appealing in its interface, creative in educational ideas, and astonishing in technical execution. Nevertheless, even these excellent applications fall short of what they could achieve by the integration of interactive, application-enriched, accessible languages. In the remainder of this paper, we pursue this critique in several ways. The second section of the paper articulates several philosophical principles for educational application design. The third section illustrates the tenets of the second, presenting three programmable applications for the domains of graphics, computational physics, and charting respectively. In the fourth section we conclude with some remarks on related and ongoing work.

10.2 Desiderata for the Design of Educational Applications

While the current culture of educational software design exhibits some tremendous strengths—as argued above—there are three principles of software design that could arguably improve that culture, and that might well be found controversial within that culture. This section presents the case for these three principles:

- A focus on "programmable application" design, integrating direct manipulation interfaces with interactive languages;
- A productive blurring of the distinction between "professional" and "educational" software;
- A variety of "scaffolding" elements, including software critics, catalogs, and embedded tutorials, whose purpose is to assist students in coping with the complexity of mastering the application interface, language, and subject domain.

The first of these principles is in a sense primary, in that it leads to the other two: The second principle might be seen as a corollary of a focus on programmable application design, and the third might be seen as an attempt to mitigate the most common pitfalls that occur in programmable application design. We now examine each of these principles in turn.

Programmable Applications

Programmable applications (Eisenberg, 1991) are software systems that integrate the best features of two important paradigms of software design—namely, direct manipulation interfaces and interactive programming environments. The former paradigm—popularly associated with menus, palettes, icon-based interaction techniques and so forth—stresses values of learnability, explorability, and aesthetic appeal; the latter, by providing a rich linguistic medium in which users can develop their own domain-oriented "vocabularies," stresses values of extensibility and expressive range.

Historically, direct manipulation and programming languages have often been viewed as (at worst) opposed or (at best) orthogonal frameworks for software design: Indeed, one of the early seminal papers on direct manipulation described the idea as "a step beyond programming languages" (Shneiderman, 1983). In point of fact, however, the respective strengths of direct manipulation and programming languages are surprisingly complementary. Direct manipulation techniques are ideal for those "extra-linguistic" tasks that reward skills of hand-eye coordination and that (maddeningly, to artificial intelligence researchers) seem to defy formal representation—tasks such as drawing a picture by hand, identifying a pleasing color combination, or steering a cursor around an obstacle. Programming, in contrast, is ideal for tasks that call for formal representation and abstraction—tasks such as the production of a novel intricate geometric design, or the creative modeling of a complex dynamical system. (Arguably the two paradigms support what Norman (1993) calls "experiential" and "reflective" styles of cognition, respectively.) Moreover—going a step further—direct manipulation and programming are capable of more than passive coordination: as will be argued later in this paper, the complementary strengths of the two approaches suggest numerous opportunities for creative and symbiotic combination.

In the context of educational software design, the programmable-application approach implies some points of difference with traditional notions of educational programming. First, rather than starting with a general-purpose programming language as a given and asking what can be done within this language framework, the programmable-application designer instead starts from the requirements of the application itself. Thus, rather than asking, "What can we incorporate into Logo (or Scheme, or Boxer) to represent (e.g.) musical notes?" the programmable-application designer instead asks, "What would be the best musical application to provide for children, and what kinds of language (and interface) elements does this suggest?" Asking the question in this alternative way in turn implies that we may well want to endow educational applications with a choice of several associated languages (that can be selected depending on the experience or preferences of the child); or that we might want to rethink fundamental language elements (such as the representation of parallel processes, or the provision of logical queries) to accommodate important ideas in the application domain; or that we might want to tailor language implementations for particular applications (e.g., operations that write to screen locations might be implemented differently depending on whether the application is intended for a single user or multiple users); or that we might want to design language features around the use of particular interface devices (such as a piano keyboard or electronic string instrument); or that we may even want to create a new application-specific language for a particular purpose.

The programmable-application approach also implies subtly different arguments for the value of programming as an activity than those historically advanced in educational circles. Rather than arguing that programming is an important skill in its own right, divorced from any particular subject matter—an argument plausibly

derived by analogy with general literacy skills (cf., Kemeny, 1983)—the programmable-application approach instead focuses on programming as a means for creative expression in particular domains of interest to the child. Thus, the reason that a child interested in (say) music might well want to learn programming is that it provides a rich musical medium allowing her to express musical ideas to which she might otherwise never have given voice. Programming, on this view, is not an exceptionally valuable skill in the abstract—either as vocational training, or because of what it may or may not transfer to—but is rather a skill whose definition and importance evolves, and becomes revealed, within particular applications and in accordance with the interests of the child. Whether a child is well-served by the activity of programming is therefore best answered not by looking at "computer-science-specific" questions—e.g., whether the child has mastered notions of data structures—but rather by looking at what the child can express, through the use of programming languages, within her own spheres of interest.

If the programmable-application approach exhibits some changes in emphasis from traditional ideas of educational programming, it exhibits an even starker divergence from the ideas implicit in most current (non-programmable) educational applications. Rather than viewing programming as beyond the capabilities of students, or as antithetical to the design of learnable software, the programmable-application designer instead creates software that allows students to work with both a powerful (and ideally, learnable) interface and a powerful (and ideally, learnable) programming language. If properly designed, the complexity of such an application should grow with use, as students develop larger and more advanced programs within the application; that is, the complexity of the application should derive from the student's ideas, and not from the sprawling size of the interface. Moreover, by representing their ideas in a progressively elaborated linguistic medium, students are given an opportunity to grow with the application over time—perhaps, over a lifetime; the transition between "educational" and "professional" activity is seamless. This leads us to the second of the three design principles mentioned above.

Educational Applications vs. Professional Applications

Many of the most familiar and successful models of education are those for which the distinction between "educational" and "professional" activity is difficult to pinpoint. The often-cited example of apprenticeship (Collins *et al.*, 1989), in which the student works alongside a professional mentor, is one such model; though the apprentice and master have distinct roles (at least at the outset of training), the apprentice develops his skills gradually through participation in professional activity. Similarly, the Brazilian samba school eloquently described by Papert (1980) allows beginning dancers, experts, and those in between, to work together as part of a common creative project. Perhaps a more mundane example is provided by piano education: a beginning piano student certainly plays different pieces than an expert, but the pathway from "beginning piano" to "expert piano" is gradual, and the instrument played by the beginner is the very same instrument as that played by the master.

Most educational applications, in contrast, do not provide the tools that allow a student to move from beginning to expert work; as such, they implicitly reflect an educational philosophy in which educational and professional activities are sharply distinguished. An apprentice city planner, for instance, would eventually seek out modeling tools more expressive than those provided by *SimCity*; an apprentice computational physicist would move away from *Interactive Physics* toward a full-fledged programming environment. These applications, marvelous as they are, would be seen as toys within the professional community—intriguing and lovely toys, perhaps, but toys nonetheless.

In principle, there need be (and perhaps should be) no firm distinction between educational and professional applications. Rather than create an "educational physics application" or "educational geometry application," or whatever, we might instead design applications intended—like the piano—for lifelong use. On this view, designers might start by thinking about what sorts of (e.g.) physics applications would be useful for the professional community; the educational version of such an application would then be a system based on the very same framework. Thus, if "real" physicists work with modeling languages, so should students of physics.

Scaffolding Elements

The first two design principles presented in this section have argued (a) for the inclusion of programming (as well as direct manipulation) elements in educational applications, and (b) for the design of educational applications that can evolve, through use, into professional applications. But these principles, admittedly, still leave a variety of nagging and unanswered questions.

One of these questions concerns the learnability of programmable applications. Conventional wisdom in commercial software design holds that programming is difficult for users, and represents an unwelcome hurdle to the "non-programmer"; presumably, this objection would be felt even more acutely for the design of educational applications. If adult users—according to this argument—do not wish to learn programming, then how much more difficult would it be to incorporate programming into applications intended for children or teenagers? Moreover, if—according to the second design principle—we wish to design educational applications that mirror professional activity, and if the professionals themselves wish to avoid programming, then how can we legitimately argue for the inclusion of programmability within educational systems?

A second, and related, question concerns the phenomenon of "burgeoning application complexity" mentioned in the introductory section of this paper. Certainly, programmable applications will themselves be complex (though perhaps the sources of complexity will be different than those in current applications); and programming environments are hardly immune to the disease of "creeping featurism." Just as interface features tend to grow in number over time, so do language primitives—the historical development of Lisp is instructive in this regard. How, then, can

applications—whether programmable or not, whether educational or commercial—provide techniques for coping with the growth of complexity over time?

Such questions are pointed, and the responses (one hesitates to call them "answers") to these objections are tentative at best. As to the first point, regarding the presumed difficulty of programming, there is at least some reason to question the conventional wisdom: Nardi (1993), for instance, presents powerful arguments for the learnability and ubiquity of formal languages in a wide range of professional activity. Moreover—and again in contrast to the conventional wisdom—at least some commercial and professional applications do indeed seem to be moving toward the inclusion of powerful "end-user programming environments"; *Mathematica*, *Director*, many database systems, and *AutoCAD* are notable examples of this trend, as are the numerous applications incorporating various types of macro-creating and scripting facilities. As to the second point, regarding the growth of complexity, it is worth reiterating that there are different sources of complexity within applications, just as there are different sources of complexity in professional domains; and the nature of complexity derived from huge feature-sets may well be different in kind, and more harmful in its impact on creativity, than complexity derived from expanding language vocabularies.

These responses, though they may have some validity, nevertheless do not exempt the designer of educational applications from considering the thorny questions of (a) how to render programmable applications more learnable, and (b) how to manage their complexity. One potentially fruitful approach to these questions is to embed a variety of "scaffolding" elements within applications—elements whose express purpose is to assist students in learning the application interface, language, and domain. Such scaffolding elements might include:

- software critics (Fischer *et al.*, 1991) that run as background routines within the application, monitoring the student's activity and alerting the student to potential problems or errors;
- browsable catalogs (Fischer *et al.*, 1992) of exemplary or illustrative work done within the application, serving as starting points for student activity;
- embedded tutorial material that the student can access in order to learn specific topics about the application, language, or domain.

Scaffolding elements of this kind are problematic additions to applications: Quite plausibly they might exacerbate, rather than alleviate, the problems of feature-expansion and burgeoning complexity. Nonetheless these new elements should be judged on an individual basis, as experiments in application design. Ideally, these experimental additions can be combined in modular fashion, and hence "mixed and matched" within particular applications; and those elements that prove successful in enhancing learnability can be selectively retained.

It is worth pausing at this juncture to summarize the three design principles advocated in this section. The first principle argues for a renaissance of the values of educational programming in the context of applications. The second argues that by pursuing programmable application design, we can realize a philosophy of education which respects the student by viewing his activity as a natural precursor of professional activity. The third argues that, in order to cope with the inevitable problems of programmable application design, we need to consider a variety of application elements geared toward enhancing learnability and alleviating the burdens of application complexity.

The following section of this paper describes three experimental programmable applications, all based on the Scheme dialect of Lisp, and all including elements of both "educational" and "professional" software. While none of these applications represents a complete realization of the design principles presented in this section, they collectively reflect an attempt to put these design principles into practice.

10.3 Three Prototype Applications

The three applications to be described here are all intended as experiments: environments in which to investigate the design principles articulated in the previous section. The first (and earliest) application, SchemePaint, is a graphics application; the second, SchemeSprings, is an application for modeling linear and nonlinear oscillators; the third, SchemeChart, is designed for the creation of charts and information displays.[3] We now describe each of these applications in turn.

SchemePaint

SchemePaint is a programmable application integrating elements of direct-manipulation paint programs with a "graphics-enriched" interactive programming environment. Figure 1 displays a view of the application. Here, the **SchemePaint** window at right is the "canvas" upon which drawings are created; the **Pen** window includes the core functionality of a skeletal direct manipulation paint-program interface; and the **transcript** window presents an interpreter for a "graphics-enriched" Scheme language. Of the menus at the top of the screen, the File, Edit, Command, and Window menus are standard MacScheme menus; the remaining menus at right are additions to SchemePaint.

The language built into SchemePaint may be viewed as a standard Scheme interpreter to which a variety of "graphics sub-languages" have been added. These sub-languages are similar in spirit to the notion of "embedded languages" advocated by Abelson and Sussman (1985): They may be thought of as independently loadable libraries consisting of new procedures, data objects, and (in some cases) interface

[3] All three of these applications were created in MacScheme (Lightship Software, Palo Alto, CA); all three run on Apple Macintosh computers.

features. Figure 1 shows the use of one of these sub-languages, for working with a Logo turtle. Figures 2 and 3 depict the use of two other sub-languages: a dynamical systems language, and an "Escher tiling language."

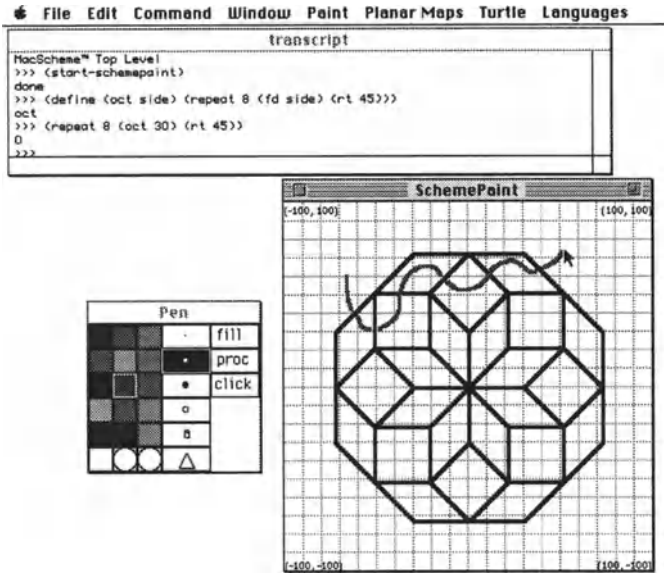

Figure 1. A view of the SchemePaint application. The **SchemePaint** window is the "canvas" in which figures are drawn; the **Pen** window contains direct manipulation paint tools; and the **transcript** window is the interpreter for a "graphics-enhanced" Scheme. Here, the user has created and used a new turtle procedure, and has added a hand-drawn squiggle.

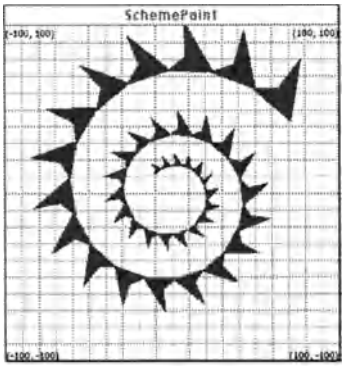

Figure 2. Here, the user employs a linear (rotate-and-scale) map by applying it iteratively to a hand-drawn polygon. SchemePaint's dynamical systems sub-language also supports the use of nonlinear maps.

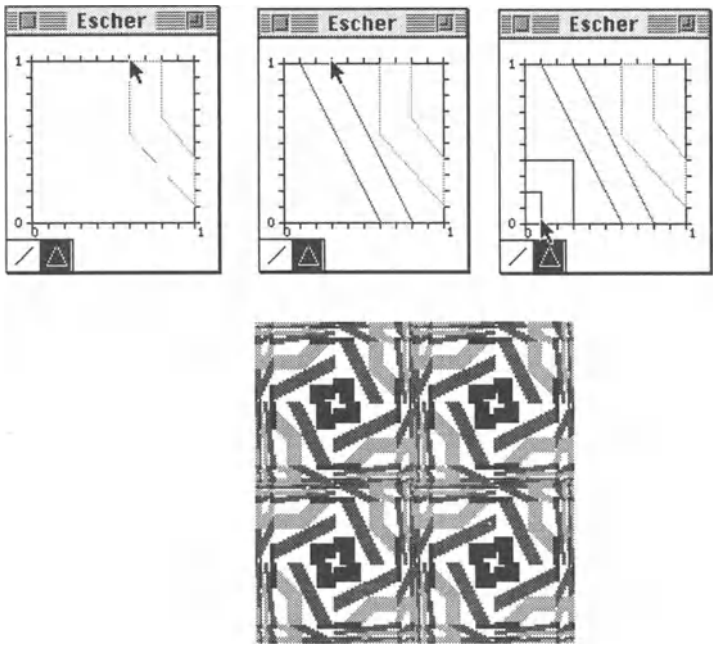

Figure 3. SchemePaint's "Escher tiling sub-language" in use. At top, in three successive stages, the user creates a "basic tile" in the **Escher** window associated with the sub-language. At bottom, the user is able to employ tiling procedures to replicate the newly-created tile.

While it would be a somewhat lengthy detour to describe all the aspects of the SchemePaint system here (more details can be found in Eisenberg, 1991), it is worth mentioning several points in the context of this paper. First, the program is designed—in the spirit of programmable applications—to facilitate both "drawing by hand" and "drawing by program." Figure 4 suggests the type of cooperation permitted by the program: the figure shows a (mostly hand-drawn) butterfly whose wings are decorated with Julia sets (Barnsley, 1988) created with SchemePaint's dynamical systems procedures.

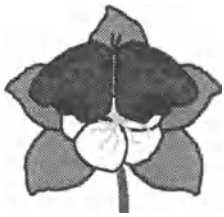

Figure 4. A SchemePaint picture.

Another point worth noting is that SchemePaint is designed not merely to permit a passive combination of direct manipulation and programming, but to facilitate a more symbiotic style of interaction between these two design strategies. Figure 5 depicts one such method employing a "programmable color" option: Here, the user has evaluated a SchemePaint expression that allows the mouse to act according to any user-specified function from x- and y-coordinates to colors. (In this particular instance, the mouse draws a color gradient from red to white based on x-position.) In the same vein, SchemePaint includes techniques for creating programmable buttons and sliders, pressure-sensitive pen procedures, and for coercing mouse-drawn points, lines, and polygons (among others) into program-accessible objects.

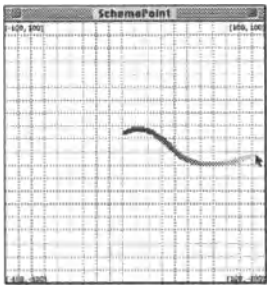

Figure 5. Here, the user has written a procedure from position (x- and y-coordinates) to color that—depending on the value of x—returns a color value smoothly varying between red and white. By assigning this "programmable color" to the mouse, the user can draw a color gradient.

Finally, because SchemePaint is presented as an application rather than as a general-purpose programming environment it is a promising venue for teaching programming to those students whose primary interests lie in graphics or computational art. More broadly, SchemePaint, like programmable applications generally, represents an opportunity for those students often described as "non-programmers" to obtain an introduction to programming in a subject closer to their own interests. The program has in fact been used in a full-semester graduate-level course to teach Scheme programming to non-computer science students; more recently, the system has been used to introduce Scheme programming to a small population of elementary and middle school students.

SchemeSprings

In the case of SchemePaint, the focus is on moving paint programs (which are typically "pure" direct manipulation systems) in the direction of programmable applications by incorporating a programming environment. By way of contrast, the SchemeSprings system seeks to augment computational physics systems (which are typically programming languages) by incorporating direct-manipulation interface tools. The SchemeSprings system, like its predecessor, seeks to combine an

interactive programming environment (in this case, centered on the domain of oscillators) with a direct manipulation interface.

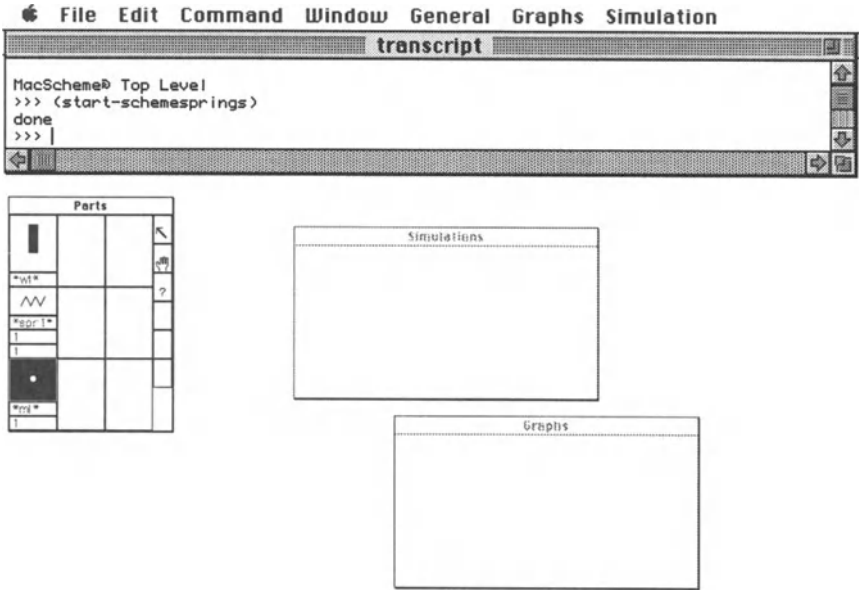

Figure 6. The initial SchemeSprings screen. Oscillators are constructed in the **Simulations** window, using elements from the **Parts** window; results can be plotted in the **Graphs** window. The **transcript** window provides access to an "oscillator-enriched" Scheme interpreter.

Figure 6 depicts the initial SchemeSprings screen. Sample parts for oscillators can be selected from the **Parts** window: these include "walls" (stationary objects), linear springs (the user can specify the natural length and force constant of the spring), and point masses (the user can specify the value of the mass). These parts may be used to construct oscillator models in the **Simulations** window; the simulations may now be run, using a fourth-order Runge Kutta integrator, and results of various kinds may be displayed in the **Graphs** window.

In addition to the direct manipulation tools that allow for the construction of linear oscillator systems, there is an "oscillator-enriched" Scheme language (again, available through the **transcript** window) that allows the user to construct a variety of "specialty" springs and masses. As a brief example, we can create a Scheme procedure to make cubic (nonlinear) springs:

```
(define (make-cubic-spring natural-len cub-coeff sq-coeff
lin-coeff)
 (make-pure-difference-spring
  natural-len
  (lambda (diff) (+ (* cub-coeff diff diff diff)
    (* sq-coeff diff diff)
    (* lin-coeff diff)))))
```

This procedure employs the SchemeSprings primitive `make-pure-difference-spring`, which creates spring objects whose force vectors depend only on the distortion of the spring from its natural "rest" length, and whose direction is opposite to that of the distortion. We can now use this procedure to create a new cubic spring object (this particular spring is "soft" in the sense that the cubic force term acts to make the spring's restoring force less than linear for small distortions (Abraham and Shaw, 1982):

```
(define soft-spring-object(make-cubic-spring 20 -0.03 0 2))
```

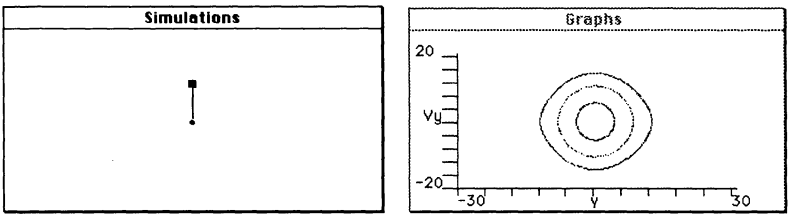

Figure 7. An experiment with a "soft" cubic spring. The **Simulations** window shows a wall-and-spring system in the course of a simulation; the **Graphs** window shows the results of three simulations (including the current one), in which the initial y-position of the mass is the only factor changed. The flattened elliptical shape of the position-versus-velocity graph (for larger initial distortions) is typical of soft springs. (Abraham and Shaw, 1982)

Figure 7 now depicts the use of this spring object: We have created a simulation in which the mass and wall are linked by a cubic spring, and have graphed the mass velocity against its position for several thousand integration steps.

From this starting point, it is also not difficult to create more complex simulations of various types. We can make springs with time-varying force constants; springs with "noisy" force constants (varying with a certain amount of randomness); masses whose value varies as a function of time or position; and many others. (cf., Andronov, Vitt, and Khaikin (1987) for examples along these lines.) Figure 8 depicts one additional experiment: a simulation of a cubic spring with a sinusoidal forcing function (a Duffing oscillator), resulting in chaotic motion (Guckenheimer and Holmes, 1983).

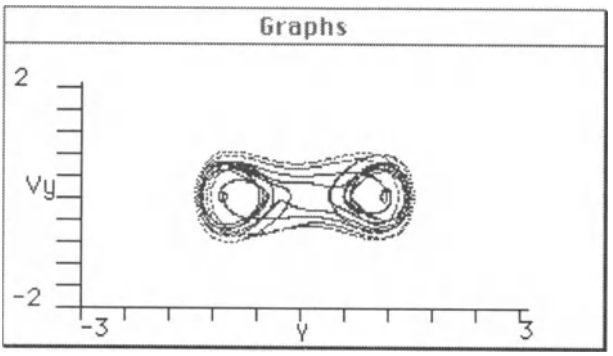

Figure 8. A chaotic position-versus-time trajectory resulting from the simulation of a forced Duffing oscillator. The parameters in this experiment were taken from an example in Guckenheimer and Holmes (1983).

SchemeChart

The third application to be described, SchemeChart, is in some ways a recent elaboration of the SchemePaint system, focusing on the creation of charts and information displays. For the purposes of this discussion, the key development in SchemeChart is the attempt to incorporate the sorts of scaffolding elements mentioned in the previous section. Figure 9 shows a scenario of the SchemeChart system in use. Graphs are created in the **SchemeChart** window; the **Paint Tools** window (like the **Pen** window in SchemePaint) permits the use of standard direct manipulation graphics tools; the **Samples** window provides a "coarse-grained" selection among chart types; the **Charts** window provides finer-grained selection based on an iconic catalog of SchemeChart examples; and the **transcript** window is the application-specific Scheme interpreter.

In the Figure 9 scenario, the user has selected the "bar chart" choice from the **Charts** window; when this selection is made, a variety of specialty bar charts are provided in the **Samples** window. The user selects "trapezoidal bar charts" (bar charts with non-horizontal upper lines, for depicting ranges of values) from the **Samples** window; now, by making a menu selection, the user is able to see a variety of relevant Scheme procedures for creating charts of this type, as presented in the window labelled **Trapezoidal Bar Chart Examples.**

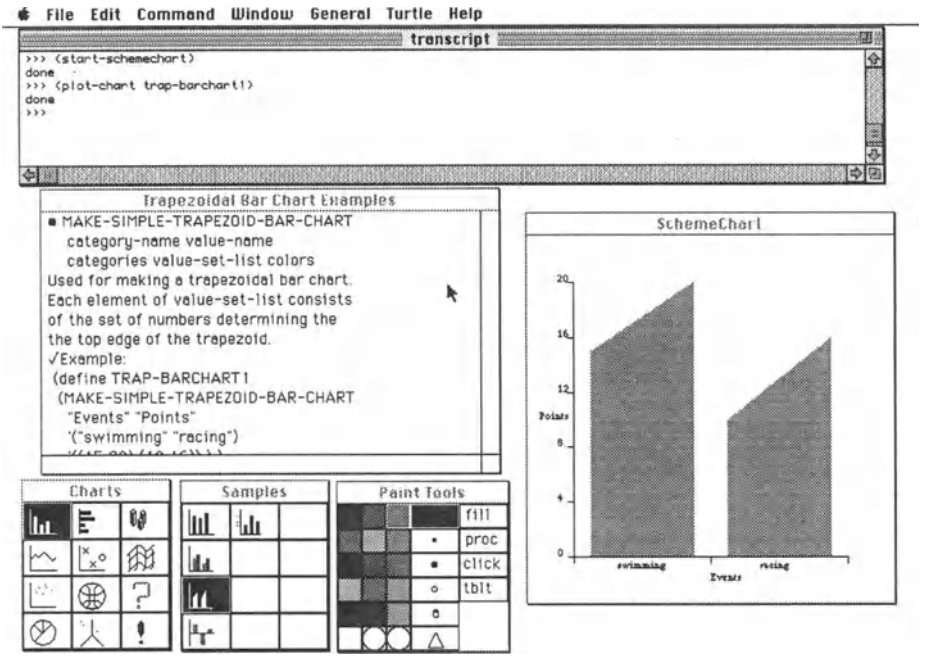

Figure 9. A screen view of the SchemeChart application. The user has created a "trapezoidal bar chart" from the sample procedure provided by the application; see the description of this scenario in the text.

The key element of this scenario is that the user is able to access language examples from an existing iconic catalog; and moreover, that these examples are based on typical tasks that the user might wish to accomplish. The Scheme procedures presented in the new window may be evaluated directly, or edited to produce new variations; this allows the user to obtain an introduction to programming tasks by example modification, as suggested by Lewis and Olson (1987).

SchemeChart includes several other techniques intended to assist the user in dealing with the complexity of the application. Figure 10 shows one such technique in a scenario similar to that shown in Figure 9. Here, the user has edited a standard color bar-chart procedure to create a chart in which the three quantities being graphed are relatively close in value. When the new graph is created, a "critic" mechanism in the application is alerted: the exclamation point icon in the **Charts** window blinks several times to inform the user that a potential problem has been spotted in her graph. The user can now (again, through a menu choice) bring up a window in which the system's critic message is passed on to the user, as shown in Figure 11; if

the user wishes she can simply ignore the message, or reedit the original expression (e.g., by plotting a different set of quantities, or trying a different type of chart).

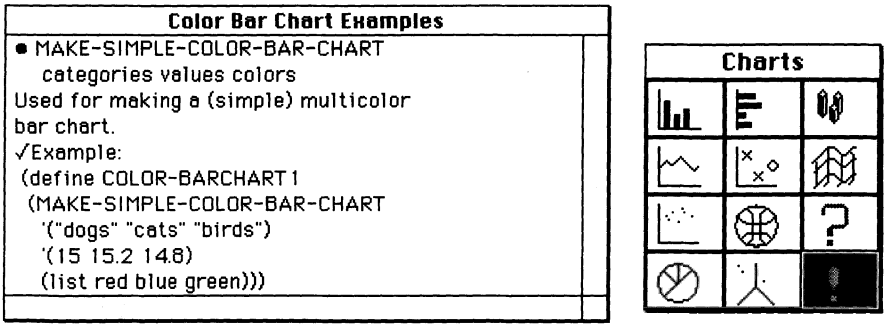

Figure 10. The user edits the sample expression in the window at left to create a new bar chart; in response, the "critic alert signal" (the exclamation point in the **Charts** window) flashes several times to indicate that a potential problem has been spotted with this newly-created chart.

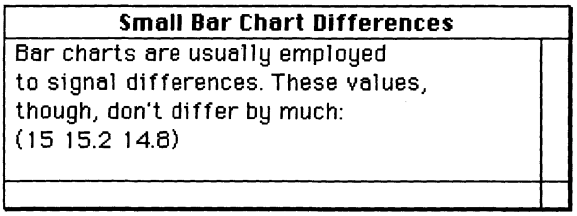

Figure 11. The critic message for the graph created in Figure 10.

One other element of SchemeChart worth noting is its "query-mode" facility, represented by the question-mark icon in the **Charts** window. In this mode of operation, the user can select—via mouse—portions of newly-created graphs (such as the axes, or axis-labels); once selected, the user can access the names of various procedures relevant to the creation or alteration of the given graph portion. By viewing "query-able objects" in this fashion, the user can begin to work with the system's language facility by association with newly-created objects.

Each of the three applications described in this section represents only a partial realization of the design principles articulated earlier. SchemePaint is perhaps the best illustration of the first design principle—as a programmable graphics application, it focuses on the integration of direct manipulation and programming

techniques. It does not, however, include any scaffolding elements, and thus provides the user with little assistance in learning either Scheme or, for that matter, the domain of graphics itself. SchemeSprings illustrates the second design principle: it is neither an "educational" nor a "professional" application *per se*, but could conceivably be employed either as an educational system in which students could simulate simple oscillator systems or as the basis of a much more advanced simulation project; indeed, the system has been used to simulate the complex behavior of Shaw's (1982) nonlinear "dripping-faucet oscillator model." Nevertheless, this application, too, provides little in the way of domain-oriented assistance to its users. SchemeChart, finally, is geared toward investigating the incorporation of scaffolding elements; but it does so at the cost of some added interface complexity.

10.4 Related and Ongoing Work

The principles and applications described here are similar in spirit to recent efforts in the development of Logo environments. Lego/Logo (Resnick, 1989), for example, is a compelling example of the integration of programmability into the domain of design and robotics; and the work described by Resnick (1993) in "programmable brick" development is a continuation of this effort. Similarly, the programmable application concept has been greatly influenced by the emphasis of the Boxer community (diSessa and Abelson, 1986) on programming as an expressive medium, by Abelson and Sussman's (1985) focus on creating domain-specific "embedded languages," and (more recently) by the work of Fischer and colleagues on domain-specific design environments (Fischer and Lemke, 1988; Fischer *et al.*, 1991). Myers *et al.* (1992) provide an excellent overview of strategies for end-user programming; Nardi (1993) advances many strong arguments in favor of programmability; and a growing and impressive body of work in programming-by-example (Cypher, 1993) might also be viewed as a principled attempt to rethink the inclusion of programming within applications and to make programming more accessible to users.

The work described in this paper diverges from each of these related efforts along various dimensions. In its emphasis on integrating programming with direct manipulation, and on embedding programming within particular stand-alone applications, there are some differences between the systems described here and the Logo/Boxer efforts; likewise the inclusion of programmability represents a departure from Fischer's earlier work in design environments. In contrast to the focus of most end-user programming efforts on "modifiability" or "tailorability" of applications, the systems described here essentially treat programming as a tool with which to rethink application domains; thus the emphasis is less one of perturbing application interfaces, and more a matter of employing programming languages as means of creative expression (Eisenberg, 1991). Finally, in contrast to the concerns of programming-by-example advocates, the work described here makes no attempt to shield users from standard constructs of programming such as iteration, recur-

sion, compound data structures, and abstraction; rather, the design strategies espoused here present the user (whether child or adult) with full-fledged programming environments. This decision is motivated by the belief that programming-by-example has yet to exhibit the level of expressiveness of "true" programming languages; and that the perceived (and vastly overestimated) difficulty of programming can be greatly alleviated by embedding task-oriented languages within applications (cf., Nardi, 1993).

In many respects, the work described here is preliminary: Certainly, in the short term, each of the three sample applications is the focus of continuing effort. SchemeSprings and SchemeChart, in particular, are still in early stages of development; a variety of direct-manipulation graphing tools are in the works for the former, and a new Lisp-based version of the latter is in preparation. In addition, two new Scheme-based applications—for the domains of chemical kinetics and diffusion-limited aggregation—are currently being developed.

In the longer term, there are exciting possibilities for expanding the notion of programmable applications to make use of newer interface devices such as DataGloves, speech recognition systems, "virtual reality" interfaces, and so forth (Foley, 1987). Perhaps the most fully-realized version of the design strategies described in this paper would work toward the development of complete "application machines" in which special-purpose hardware, interfaces, and programming languages could be employed concurrently for educational and professional applications. Thus, rather than thinking in terms of a "graphics application," whether for children or adults, we might consider instead the creation of a full-fledged "language-based graphics studio"—complete with programmable electronic easels, wall-sized screens, 3D viewing devices, and so forth. In keeping with the principles advocated here, such an environment would reflect a spirit of integration: of language and hand-eye coordination, of "abstract" and "reactive" thought, of professional and educational work, and of adults' and children's experience.

Acknowledgments

Thanks to Gerhard Fischer, Julie Di Biase, Chris DiGiano, Gina Cherry, and members of the Human-Computer Communication Group at the University of Colorado for their contributions to the ideas in this paper. Special thanks also to Hal Abelson, Andy diSessa, and Gerald Jay Sussman for their mentorship. This research was supported by NSF grants RED-9253425, IRI-921034, and by a Young Investigator award (IRI-9258684).

References

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

Abraham, R. H. and Shaw, C. D. (1982) *Dynamics: The Geometry of Behavior*, Santa Cruz, CA: Aerial Press, Inc.

Andronov, A. A., Vitt, A. A., and Khaikin, S. E. (1987) *Theory of Oscillators*, New York: Dover

Barnsley, M. (1988) *Fractals Everywhere*, Boston: Academic Press, Inc.

Collins, A., Brown, J. S., and Newman, S. E. (1989) Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics, in L. Resnick (ed.), *Knowing, Learning, and Instruction*, Hillsdale, NJ: Lawrence Erlbaum

Cypher, A. (ed.) (1993) *Watch What I Do*, Cambridge, MA: MIT Press

Dargahi, N. (1991) *SimCity Strategies and Secrets*, Alameda, CA: Sybex Inc.

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

Eisenberg, M. (1991) Programmable applications: Interpreter meets interface, MIT AI Laboratory Memo 1325

Fischer, G., Grudin, J., Lemke, A.C., McCall, R., Ostwald, J., Reeves, B.N., and Shipman, F. (1992) Supporting indirect, collaborative design with integrated knowledge-based design environments, *Human-Computer Interaction*, 7/3, 281-314

Fischer, G., Lemke, A.C., Mastaglio, T., and Morch, A. (1991) The role of critiquing in cooperative problem solving, *ACM Transactions on Information Systems*, 9/2, 123-151

Fischer, G., and Lemke, A.C. (1988) Construction kits and design environments: Steps toward human problem-domain communication, *Human-Computer Interaction*, 3/3, 179-222

Foley, J. (1987) Interfaces for advanced computing, *Scientific American*, 257/4, 126-135

Guckenheimer, J. and Holmes, P. (1983) *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, New York: Springer-Verlag

Kemeny, J. G. The case for computer literacy, *Daedalus*, Spring 1983

Lewis, C. and Olson, G. (1987) Can principles of cognition lower the barriers to programming?, in G. Olson, S. Sheppard, and E. Soloway (eds.) *Empirical Studies of Programmers: Second Workshop*, New Jersey: Ablex

Luehrmann, A. (1982) Don't feel bad about teaching BASIC, *Electronic Learning*, September 1982

Myers, B, Smith, D. C., and Horn, B. (1992) Report of the "end-user programming" working group, in B. Myers (ed.) *Languages for Developing User Interfaces*, Boston, MA: Jones and Bartlett

Nardi, B. (1993) *A Small Matter of Programming*, Cambridge, MA: MIT Press

Norman, D. (1993) *Things That Make Us Smart*, Reading, MA: Addison-Wesley

Papert, S. (1980) *Mindstorms*, New York: Basic Books

Resnick, M. (1989) Lego, Logo, and life, in C. Langton (ed.), *Artificial Life*, 397-406, Reading, MA: Addison-Wesley

Resnick, M. (1993) Behavior construction kits, *Communications of the ACM*, 37/7, 64-71

Shaw, R. (1984) *The Dripping Faucet as a Model Chaotic System*, Santa Cruz, CA: Aerial Press, Inc.

Shneiderman, B. (1983) Direct manipulation: A step beyond programming languages, *IEEE Computer*, 16/8, 57-69

Sleeman, D. and Brown, J. S. (eds.) (1982) *Intelligent Tutoring Systems*, London: Academic Press

Solomon, C. (1986) *Computer Environments for Children*, Cambridge, MA: MIT Press

Software

AutoCAD Autodesk, Inc. Sausalito, CA
CorelDraw Corel Corporation. Ottawa, Ontario, Canada
Director MacroMind, Inc. San Francisco, CA
Explorer Sunburst. Pleasantville, NY
The Geometer's Sketchpad Key Curriculum Press. Berkeley CA
The Geometric Supposer Sunburst. Pleasantville, NY
Interactive Physics II Knowledge Revolution. San Francisco, CA
Kid Pix Broderbund Software, Inc. Novato, CA
MacScheme Lightship Software, Inc. Palo Alto, CA
Mathematica Wolfram Research, Inc. Champaign, IL
SimCity Maxis, Inc. Orinda, CA
SimLife Maxis, Inc. Orinda, CA