KIDSIM: *Programming Agents Without a Programming Language*

David Canfield Smith

Software agents are our besf hope during the 1990s for obtaining more power and utility from personal computers. Agents have the potential to *participate actively* in accomplishing tasks, rather than serving as passive tools as do today's applications. However, people do not want generic agents—they want help with *their* jobs, *their* tasks, *their* goals. Agents must be flexible enough to be tailored to each individual. The most flexible way to tailor a software entity is to program it. The problem is that programming is too difficult for most people today. Consider:

- How can ordinary people program agents? Most people today would say they cannot.
- How can ordinary people understand what agents are doing? Will they turn dozens or hundreds of agents loose in their computers if they cannot? Or even one?

Unless these problems are solved, agents will not be widely used.

The End-User Programming Problem

How can people tell agents what to do? More generally, how can ordinary people, who are not professional programmers, program computers? This problem—the "end-user programming problem"—is an unsolved one in computer science. In spite of many previous attempts to develop languages for end users, today only a small percentage of people are able to program. Why are most people unable to do it, in spite of all the attempts to empower them? Is programming inherently too difficult? Or does the fault lie with computer scientists? Have we developed languages and approaches best suited to the skilled practitioner, languages that take months or years to master? The authors take the latter view: computer scientists have not made programming easy enough. Consider the following evidence:

First, observe that most people can follow a recipe, give directions, make up stories, imagine situations, plan trips—mental activities similar to those involved in programming. It seems well within the capacity of humans to construct and understand concepts like sequences (first add rice, then add salt), conditionals (if the water boils too fast, turn down the heat), and variables (double each quantity to serve eight).

Can we make programing as easy as giving directions?

Second, notice that most people can use personal computers. Today, over 100 million people use them to write letters and reports, draw pictures, keep budgets, maintain address lists, access databases, experiment with financial models, play games, and so forth. Children as young as two years old can use a mouse and paint with programs like KidPix (a child's painting program, at one time the world's best selling application) or explore worlds like "The PlayRoom" (a child's adventure game). So computers are not inherently unusable. The key observation is that most of these applications are *editors:* with them, users produce an artifact by invoking a sequence of actions and examining their effects. When the artifact is the way they want it, they stop.

Can we make programming as easy as editing?

Let us define the term "end users" to mean people who use computers but who are not professional programmers. Such people are typically skilled in some job function, but most have never taken a computer course. They use programs ("applications") written by other people. They cannot modify these programs unless the designer explicitly built in such modification, and then the modification is typically limited to setting preferences. Perhaps 99% of the 100 million computer users can be classified as end users. If we could empower these people to program computers, the impact would be enormous.

In the past two decades there have been numerous attempts to develop a language for end users [21]: Basic, Logo, Smalltalk, Pascal, HyperTalk, Boxer [7], Playground [8], etc. All have made progress in expanding the number of people who can program. Yet as a percentage of computer users, this number is still abysmally small. Consider children trying to learn programming. When they are in class, most children will learn anything. But do they continue to program after the class ends? Today programming classes are characterized by the "whew, I'm glad that's over!" syndrome. As soon as children do not have to do it anymore, they go on to something that's actually fun.

We hypothesize that fewer than 10% of the children who are taught programming continue to program after the class ends. This is based on personal experience and observation. Surprisingly there are no published studies on this issue, to our best knowledge. Nevertheless, we expect that most readers will agree with this hypothesis. Soloway (private communication) states "My guess is that the number . . . is less than 1%!!! Who in their right mind would use those languages—any of them—after a class?"[1] Single-digit percentiles indicate that the end-user programming problem has not yet been solved.

As a step toward solving this problem, we will describe a prototype system designed to allow children to program agents in the context of simulated microworlds. Our approach is to apply the good user interface (UI) principles developed during the 1980s for personal computer applications to the *process* of programming. The key idea is to combine two powerful techniques—graphical rewrite rules and programming by demonstration. The combination appears to provide a major improvement in end users' ability to program agents.

[1]Private communication.

Good User Interface Principles for Programming Environments

Why have previous attempts to develop a usable EUP language not been more successful? The authors themselves have developed languages for end users, none of which were successful. When people are not making progress on a problem, it is often because they are asking the wrong question. We decided that the question is not: what language can we invent that will be easier for people to use? The question is: should we be using a language at all? This was the starting point for the work described here. We have come to the conclusion that since all previous languages have been unsuccessful by the criterion described here, language itself is the problem. It does not matter what the syntax is. Learning another language is difficult for most people. The solution is to get rid of the programming language.

But if we do, what do we use instead? The answer is all around us in the form of personal computers. Today, all successful personal computer applications and many workstation applications follow certain human-computer interface principles that were developed in the late 1970s [20] and codified during the 1980s [1]. The most common embodiment of these principles is the so-called graphical user interface (GUI) consisting of windows, menus, icons, the mouse, and so forth. The principles that make this interface work can and should guide computer scientists in attacking the end-user programming problem. We will briefly describe a few of these principles. However, we want to emphasize that we did not invent these principles in the work reported here. We are merely applying them to programming. Furthermore, the description here is by no means complete; many books have been written on these principles. See, for example, [2, 11, 12].

The following are the most important principles for solving the end-user programming problem.

- *Visibility:* Make everything relevant to people's operation of a computer system visible on the display screen. This is the single most important UI principle. People have an easier time understanding what is going on and what to do next if information is visible than if it is kept internal to the program and hidden from users. Without visibility it is almost impossible to achieve an easy-to-use interface. Visibility has a couple of related principles:

Interactive vs. batch: Establish a cause-effect relationship between user actions and system semantics. When users do something, show the effects immediately. Systems that do not are confusing.

Modeless vs. modal: A "mode" is a state of a system in which user actions are interpreted differently than they would be ordinarily. Systems get in trouble when either (a) they have many modes or (b) their modes are invisible. Both confuse people, leading to (usually unpleasant) surprises at the results of actions.

Coping and modifying vs. creating from scratch: Allow people to copy and modify existing items in a system as a way to create new ones. It is often easier to start with something that works and figure out how to modify it than to create the same thing from scratch. Revealingly, this is the way most professional programmers work.

Seeing and pointing vs. remembering and typing: Allow users to point to entities on the display screen with a pointing device, instead of making them describe the entities by typing text. It is the foundation for the popular concept of "direct manipulation."

Concrete vs. abstract: Make the entities presented to users concrete. People have an easier time with the concrete than with abstract.

Familiar user's conceptual model: Cast the concepts in a system into terms the user can understand. When faced with a new situation, people try to apply their existing knowledge to understand it. This is the inspiration for the use of metaphor in computer interfaces, especially the so-called desktop metaphor invented for the Xerox Star by [20].

Minimum translation distance: One principle of utmost importance for programming environments but not so much for other applications was proposed by Sloman [18]: minimize the conceptual distance between people's mental representations of concepts and the representations that the computer will accept. In our opinion, the failure to do so is the single biggest reason that languages designed for children such as Logo and Smalltalk have not attained wider use. Time and again we have watched children try to accomplish simple programming tasks such as making a fish swim away from a shark, only to be frustrated by the difficulty in having to deal with coordinate systems and vectors. The most articulate representations are the ones that minimize this translation distance. Of course this is also a principle of good program design: create data structures and operations that are close to those in the problem domain.

In summary, the GUI eliminated command lines by introducing visual representations for concepts and allowing people to directly manipulate those representations. It has empowered millions of people to use computers. Today, all successful editors on personal computers follow this approach. But most programming environments do not. This is the reason most people have an easier time editing than programming.

Actually, some programming systems *have* adopted an editing interface, and they are beginning to broaden the community of programmers. Spreadsheets, the most widely used programming technology, have done this for years. The popularity of some user interface management systems with their "drag-and-drop" interface builders is a result of their allowing direct manipulation of interface elements. Similarly, most people can construct buttons and fields in HyperCard, which has an editing feel, but few of those same people can program in HyperTalk.

There are a few brilliant examples of programming systems that have applied *all* of these principles. Our favorite is Bill Budge's video game for personal computers called "The Pinball Construction Set." It allows people to program pinball games by directly editing the layouts, i.e., by dragging and dropping pinball elements such as flippers and bumpers. The elements begin functioning as soon as they are dropped into place. Everyone can create pinball games this way. We call this "programming by direct manipulation," and when done well, it is wonderfully successful. The problem with "The Pinball Construction Set" is that you can only program pinball games with it. The challenge is to increase the generality without losing the ease of use.

Simulations

The end-user programming problem in its full generality is a tough one. It has resisted solution for over two decades. So we decided to attack it in a domain that is more general than "The Pinball Construction Set" but more restricted than general programming, the domain of symbolic simulations. A symbolic (as opposed to numeric) simulation is a computer-controlled microworld made up of individual objects (agents) which move around a game board interacting with one another. We chose as our target audience children in the age range of 5–18 years old.

Why simulations? They are a powerful tool for education. Simulations encourage unstructured exploratory learning. They allow children to *construct* things, supporting the constructivist approach to education. Alan Kay contends "We build things not just to have them, but to learn about them." He quotes the philosopher Cesare Pavese, "To know the world, one must construct it." Scardamalia [17] argues that children learn best when constructing things. They enter Vygotsky's "zone of proximal development." Simulations such as SimCity and SimEarth allow children (of all ages) to construct unique microworlds, giving them a sense of ownership in their creations. They can observe and modify and experiment with these microworlds. Children are the "gods" of their worlds. This pride of ownership and feeling of power are compelling qualities that motivate even professional programmers.

However, most simulations today do not permit users to modify their fundamental behaviors and assumptions. For example, one cannot alter the fact that if one puts in a railroad in SimCity, the pollution problems go away—not exactly a realistic consequence. This inflexibility is the reason that most school teachers do not use SimCity as a teaching tool, even if they are studying city building. It does not model what they want to communicate. Simulations that do allow fundamental modifiability, such as numeric simulations built with Stella, require extensive programming skills. Few children or teachers can or want to do it.

What is needed is a way for children without programming knowledge to have more control over the behavior of simulations. What is also needed is a way for teachers to tailor simulations to support their curriculum goals. KidSim™ provides a way to do both.

KidSim

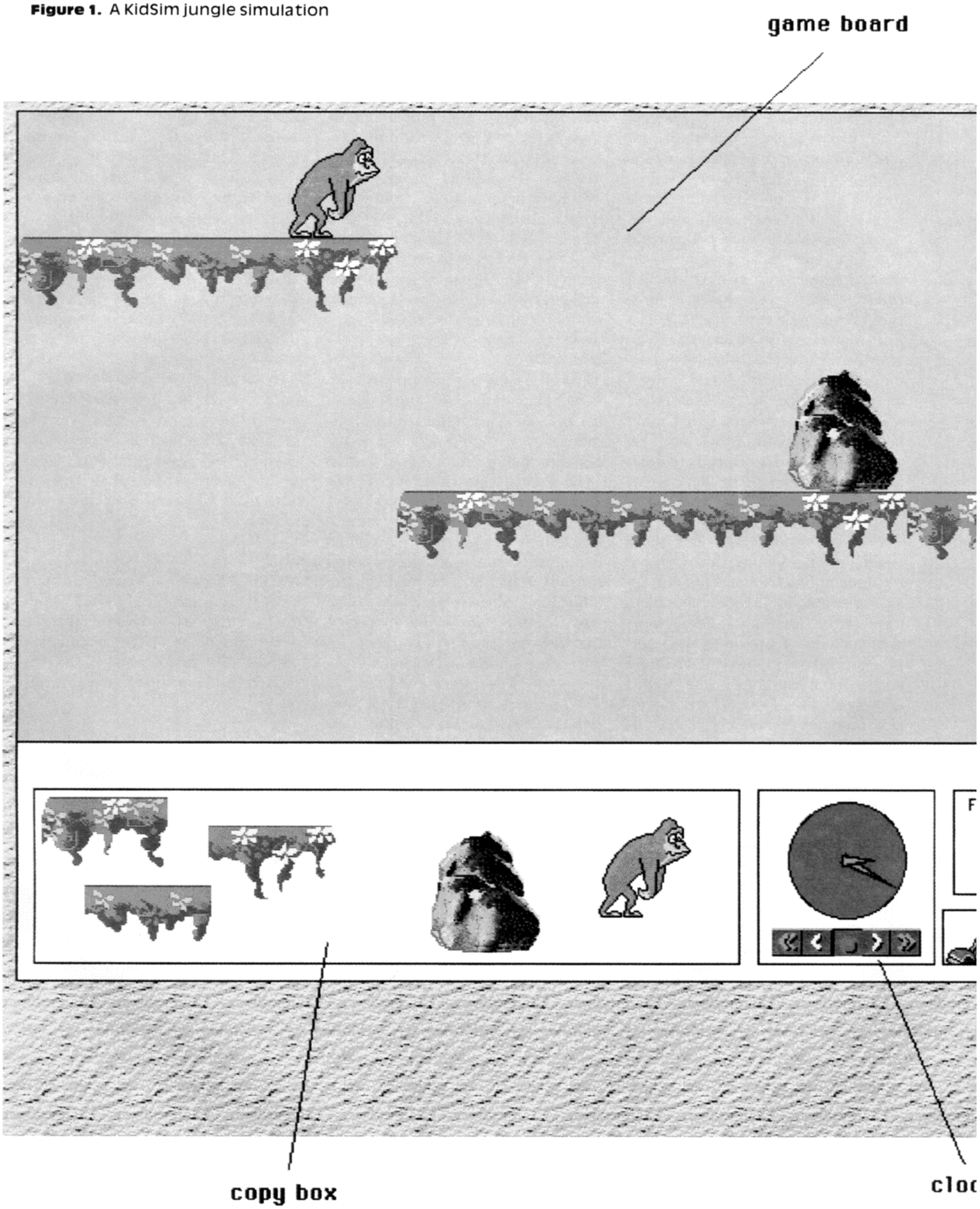
KidSim ("Kids' Simulations") is a tool kit that allows children to build symbolic simulations. Kids can modify the programming of existing simulation objects and define new ones from scratch. KidSim simulations consist of:

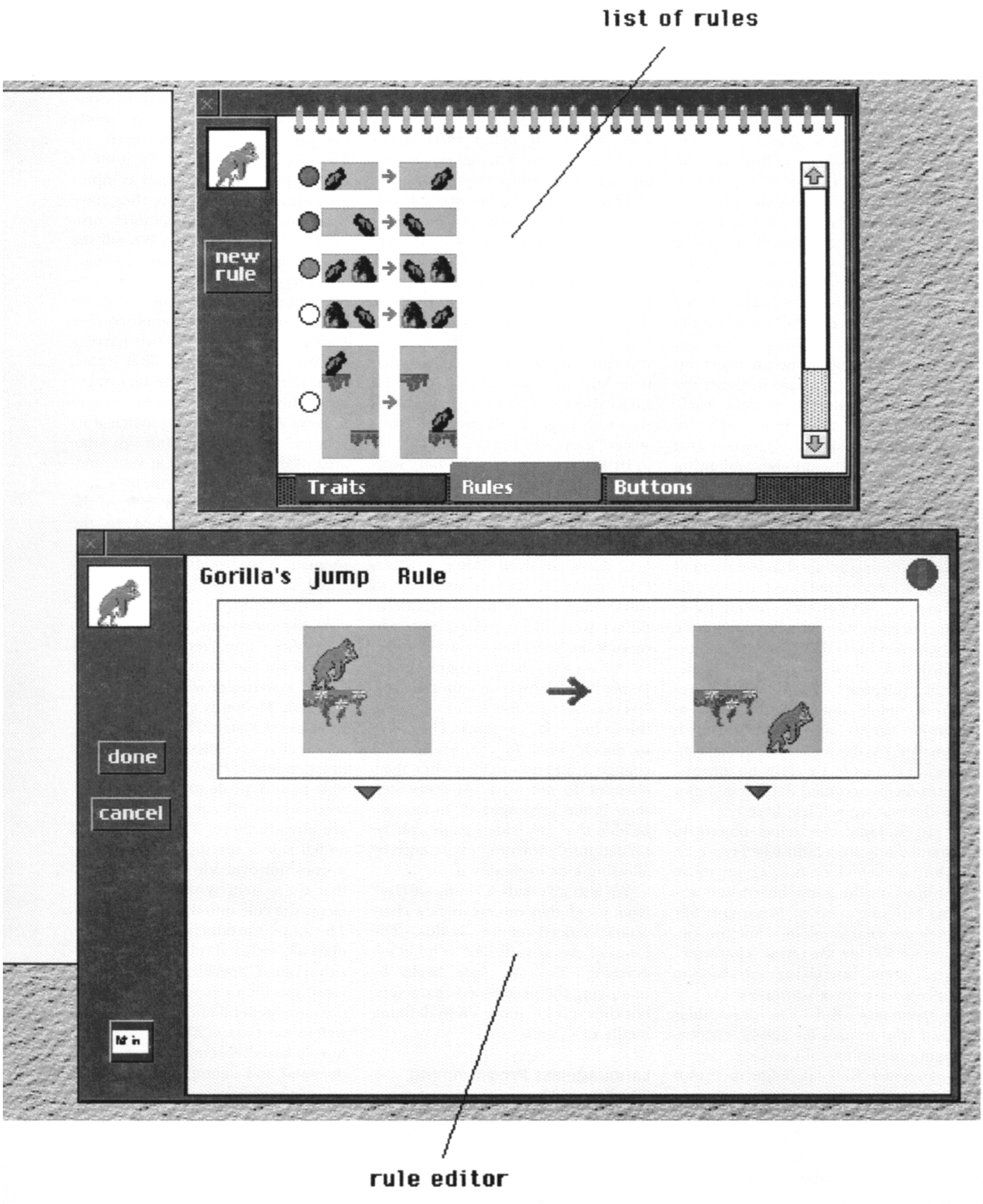
- a *game board* divided into discrete spaces, like a checkerboard
- a *clock* whose time is divided into discrete ticks
- one or more *simulation objects* (agents)
- a *copy box* which is the source of new simulation objects
- a *rule editor* where rules are defined and modified
- various other elements

In this article we will focus on simulation objects and the way kids program them.

The game board represents the simulation microworld. It is the environment in which simulation objects interact with one another. Dividing it into discrete squares makes it easier for kids to communicate their intentions to the computer. The game board in Figure 1 displays a monkey in a simple jungle scene. We will use this simulation throughout this article.

Figure 1. A KidSim jungle simulation




list of rules
new rule
Traits
Rules
Buttons
Gorilla's jump Rule
done
cancel
rule editor

The clock starts and stops a simulation running. Dividing time into discrete ticks makes it easy for kids to control their simulations. The clock provides both fine-grain control over time (single stepping) and the ability to run time backward. Running the clock backward undoes everything that happened during the previous clock tick, encouraging kids to experiment and take chances. If something goes wrong, they can just back up the clock to before that point.

The copy box is a container for simulation objects that automatically makes copies of things inside it. Whenever a child drags an object out of the copy box, the system clones the object and puts the original back. This provides an infinite source of new objects. Kids can place their own objects in the copy box, allowing them to infinitely duplicate their own objects as well.

Agents in KidSim

Let us define an *agent* as a persistent software entity dedicated to a specific purpose. "Persistent" distinguishes agents from subroutines; agents have their own ideas about how to accomplish tasks, their own agendas. "Specific purpose" distinguishes them from entire multifunction applications; agents are typically much smaller. (As demonstrated by the articles in this issue, this is by no means a universally accepted definition, but it is the one we will use here.)

In KidSim, the active objects in simulations are agents (see Figure 2). During each clock tick, agents move around on the game board interacting with one another. Metaphorically they are characters in a microworld, and we will use the terms "character" and "agent" interchangeably. KidSim agents have three attributes:

• *appearance:* Kids can draw their own appearances for agents, encouraging metaphorical thinking.
• *properties:* Kids can define their own data and characteristics for agents. Typical ones for a monkey character might be "name," "age," "height," "weight," "sex," "hunger," "fear," and "climbing ability." Properties are name-values pairs. They serve the same function in KidSim that variables do in traditional programming languages. Properties have no inherent meaning to KidSim. They have meaning only if kids use them in rules.
• *rules:* Kids can define rules of behavior for agents. The set of rules for an agent constitutes its program.

Thus KidSim agents are full objects in the object-oriented programming sense. They have state (properties), behavior (rules), and an appearance. While there is no inheritance between agents, there is a way to give every agent the same rule.

KidSim agents are similar to those in Logo Microworlds. The difference is in how they are programmed. In Logo Microworlds, kids program objects with Logo. In KidSim, kids construct "graphical rewrite rules."

In KidSim, kids usually start with several predefined characters in various microworlds. The kids can play with these microworlds immediately, as with ordinary video games. This gets them involved. After a while, they typically want something to work differently. At this point KidSim differs from video games. Kids can modify the way the characters work, by changing their programming. Pedagogically this is an important difference, because the act of changing things forces kids to think. They have to decide what to change, how to change it, and how to fix it when their changes do not work. At every step their brains are engaged. In fact, we believe that any video game can be turned into a learning experience by allowing kids to modify it.

We also give kids a "lump of clay" from which they can create new characters, indeed entire worlds. The lump of clay is sufficient to build everything. Typically kids begin by modifying the predefined characters, but they quickly move on to defining totally new ones.

Languageless Programming, the Key Idea

The main innovation in KidSim is the way in which children specify the behavior of agents. KidSim does it without a programming language. Instead KidSim combines two powerful ideas:

1. graphical rewrite rules
2. programming by demonstration

Each has been tried before in isolation by various researchers, including the present authors, and each has been found insufficient by itself to enable people to program computers. This is the first time they have been combined in a general programming environment. We call the result "languageless programming."

A graphical rewrite rule is a transformation of a region of the game board from one state to another. (See Figure 3.) It consists of two parts: a "before" part and an "after" part. Each part is a small scene that might occur during the running of the simulation. A rule is said to match if its "before" part is the same as some area of the game board at some moment in time. When a rule matches, KidSim transforms the region of the game board that matched to the scene in the "after" part of the rule. (Actually a recorded program is executed, as described later.)

Rewrite rules or "if-then rules" or "production systems" are well known in artificial intelligence [5, 14, 16]. They form the control structure for expert systems, of which OPS5 from Carnegie-Mellon is an example [13]. Rule-based systems have some marvelous characteristics. Since rules are independent of one another, it is possible to add a rule to an existing system without affecting the rules that are already there. This assumes the added rule is specific enough so that it does not override other rules and that the system is smart enough to factor the rule into the correct order. The Lisp70 production system automatically factored rules using an algorithm called "specificity" in which the more specific rules were tried before the more general ones, which worked well in most cases [22]. Furthermore, in rule-based systems it is easy to understand and debug each rule by itself, without having to be concerned with the other rules. Of course a good rule tracer and stepper are essential, as in any programming language.

Graphical rewrite rules are two-dimensional versions of rewrite rules. They, too, have been applied to the

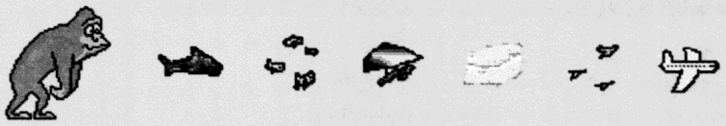


Figure 2. Examples of agents

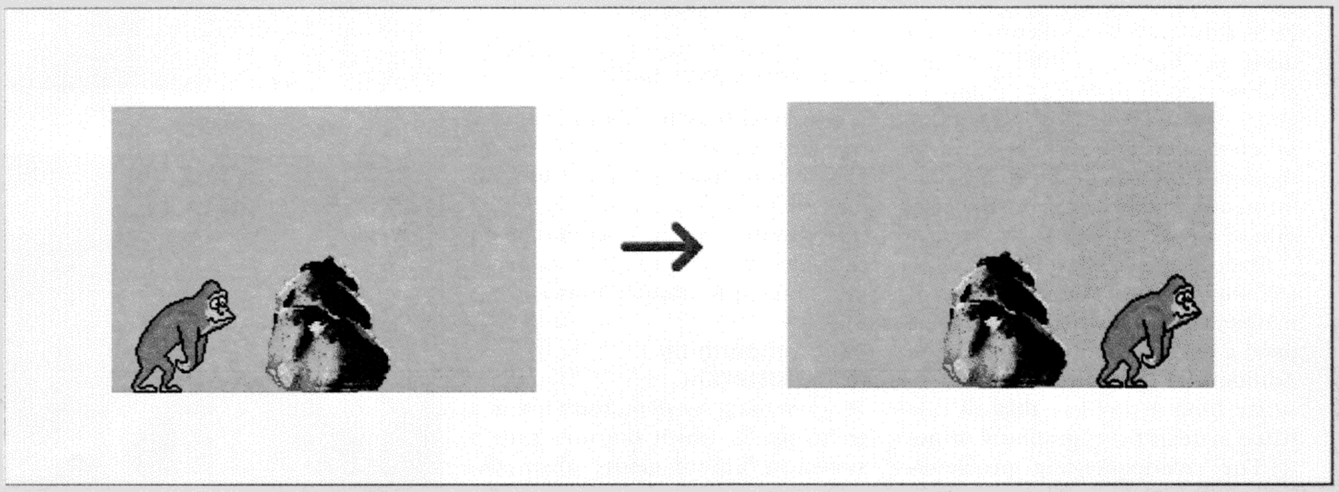


Figure 3. A graphical rewrite rule

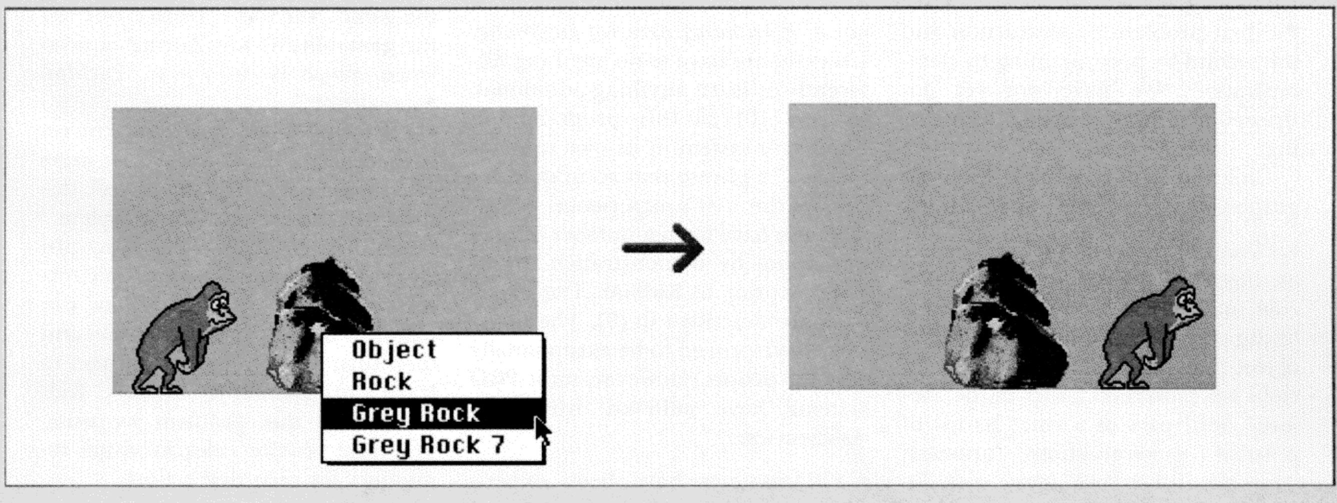


Figure 4. An example of picture abstraction

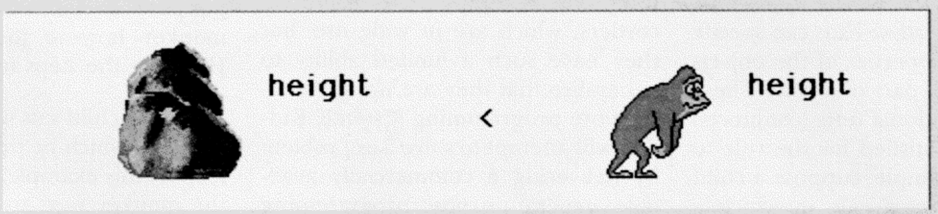


Figure 5. An example of property abstraction

end-user programming problem by several researchers [9,15]. (See also A. C. Kay, *Tableau,* 1988, unpublished.) While they work well for simple tasks, they have encountered two problems that have limited their utility for complex tasks: (a) The "rule-generality" problem—pictures, being inherently literal, are difficult to generalize to apply to multiple situations; and (b) The "rule-semantics" problem—it is difficult to specify how the computer is to perform the transformation from the left to the right side of a rule.

Some systems have applied AI techniques to try to infer the transformation, but to date no one has developed a general method for doing so. Additionally, graphical rewrite rules suffer from a problem that all rule-based systems have, graphical or not: (c) The "rule-sequencing" problem—it is difficult to specify a series of transformations, i.e., do rule A then rule B then rule C, since rules by definition are independent of each other. KidSim's graphical rewrite rules solve the first problem by abstraction and the second by programming by demonstration. We have not yet addressed the third problem, sequencing.

Children may generalize KidSim's graphical rewrite rules in two ways:

• *Picture abstraction:* Kids may select an object in the "before" part of a rule, and a pop-up menu will appear listing possible generalizations of that object (Figure 4). In this example, a child has clicked on a rock in the "before" (left) part of a rule. Its list of possible generalizations appears: "this particular rock (grey rock 7), any grey rock, any rock, or any object." The child may specify that the rule is to apply to any of these types of objects.

• *Property abstraction:* Kids can specify tests on the properties of the objects in the "before" part of a rule. These constitute additional tests (conjuncts) that must be satisfied for the rule to match. For example, suppose a child wants to restrict a rule in which a monkey jumps over rocks to rocks that are less than the monkey's height. Adding the property test in Figure 5 to the rule does this. If a child buttons down on the < symbol, a pop-up menu of operators appears showing the allowable tests on numeric properties ($< \leq = \neq \geq >$). Text properties have other operators. A child may choose any operator.

Now we can fully define a graphical rewrite rule in KidSim:

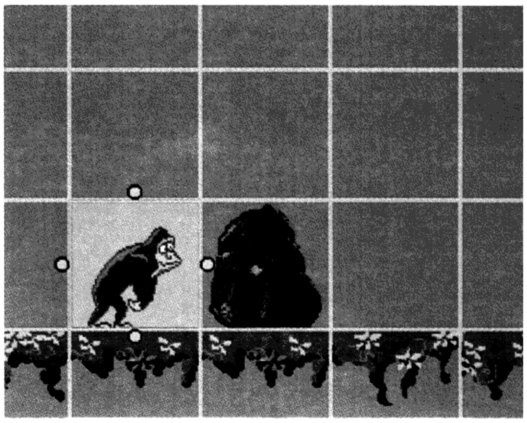
A **graphical rewrite rule** consists of a (possibly generalized) visual image and zero or more property tests. In order for a rule to match, its visual image must conform to a situation on the game board, and all of its property tests must evaluate to true.

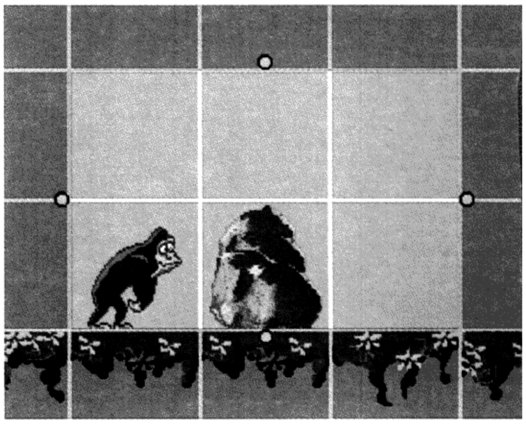
Programming by Demonstration

Programming by demonstration is a technique in which the user puts a system in "record mode," then continues to operate the system in the ordinary way, and the system records the user's actions in an executable program [4, 19]. The key characteristic is that *the user interacts with the system just as if recording were not happening.* Users do not have to do anything differently or learn anything additional. Halbert [10] calls this "programming a software system in its own user interface," a phrase that accurately expresses the user's experience.

There have been a number of programming by demonstration (PBD) systems prior to KidSim. The major ones are described in [3]. These systems have proved to be exceptionally easy for people. However, most PBD systems have suffered from two deficiencies:

• PBD systems have been experimental, used by small numbers of people for simple tasks but not by large numbers of people for complex tasks. The exceptions are macro recorders, which are in wide use, but they have such a limited ability to generalize that they are not general-purpose programming systems. KidSim will attempt to solve this problem by delivering a commercially available, general-purpose programming product. It is designed to be powerful enough to enable children to construct simulations as complicated as





the game "PacMan." In fact, our test for generality is not Turing equivalence, which is easy; it is "PacMan equivalence."

• PBD systems do not represent recorded programs in a way that users can understand. We might call this the "PBD representation problem." Often they show programs as scripts. But it does not work to let people record programs in a way they can handle—by demonstration—and then turn around and force them to learn a programming language! KidSim solves this problem by using graphical rewrite rules as visual reminders for recorded actions.

An Example of Programming in KidSim

Suppose a child wants to teach a monkey how to jump over rocks. Here are the steps involved:

1. The child sets up the simulation situation which he or she wants to affect. In this example, the child places the monkey next to a rock. KidSim allows children to define rules only when the actual simulation situation exists. This makes defining rules a

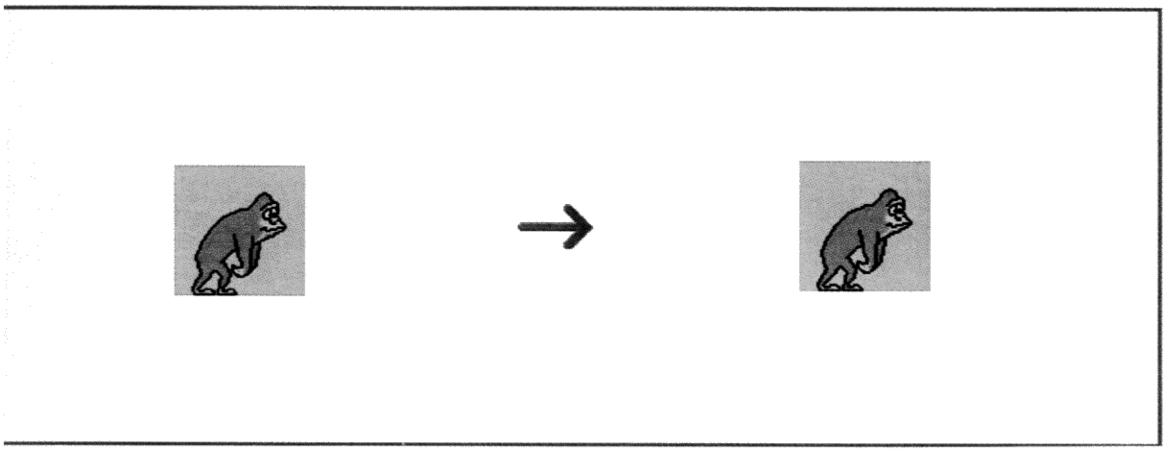
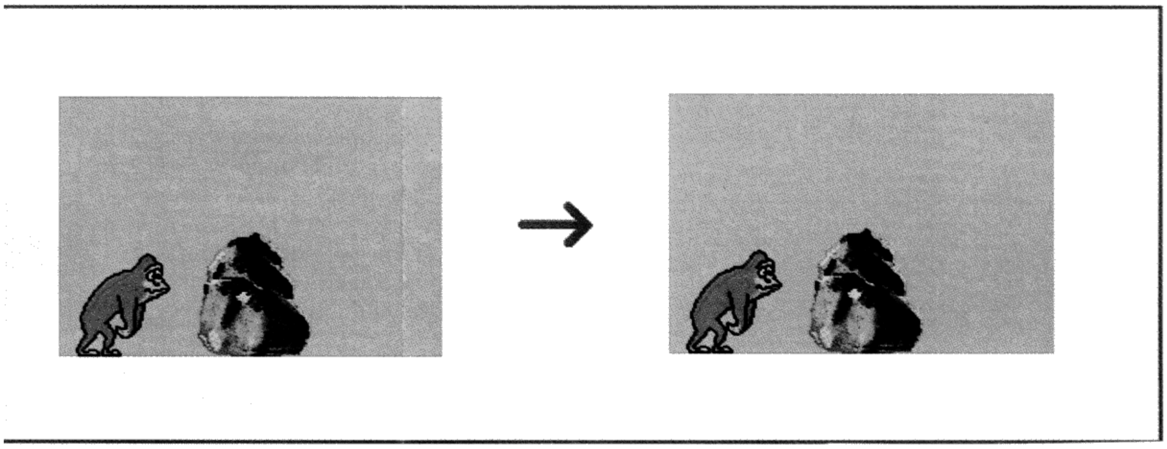


Figure 6. Defining the context for a rule



concrete process, reducing the need to visualize simulation states abstractly. The child can be sure the rule will work at least for this one example, and the child can generalize it to a wider class of situations later.

2. The child specifies the region of the game board with which the rule is to deal (Figure 6). This is the region that will be pattern-matched against the game board when the simulation runs. The child specifies this region by direct manipulation, by dragging the border of a "spotlight" which appears during recording. The "before" and "after" pictures in the rule copy the "spotlight's" area.

3. Initially the "before" and "after" parts of rule are identical, i.e., each rule begins as an identity transformation. The child defines the rule semantics by editing the "after" picture to produce a new simulation state. (See Figure 7.) First the child places the cursor (a small hand) over the monkey and drags it to the square above the rock (Figure 8). Then the child drags it to the square to the right of the rock (Figure 9). Done. That's all there is to it. Nowhere did the child type "begin . . . end," "if

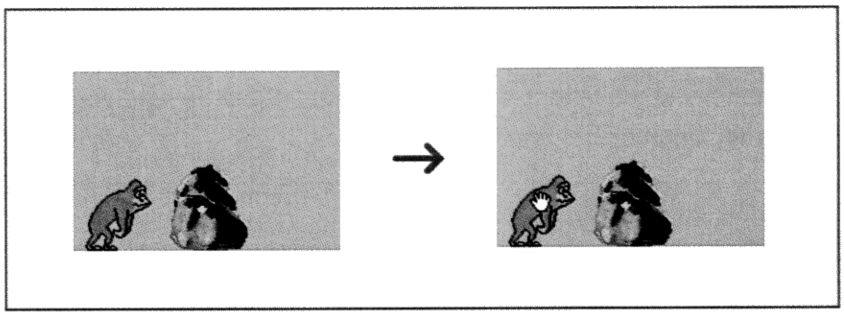


Figure 7. Defining a rule by demonstration

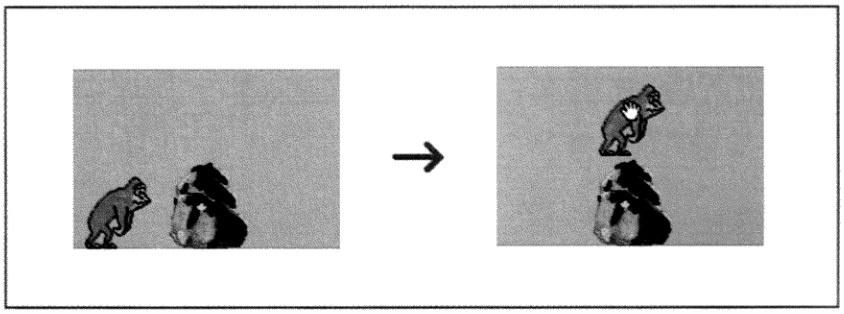


Figure 8. Dragging the monkey above the rock

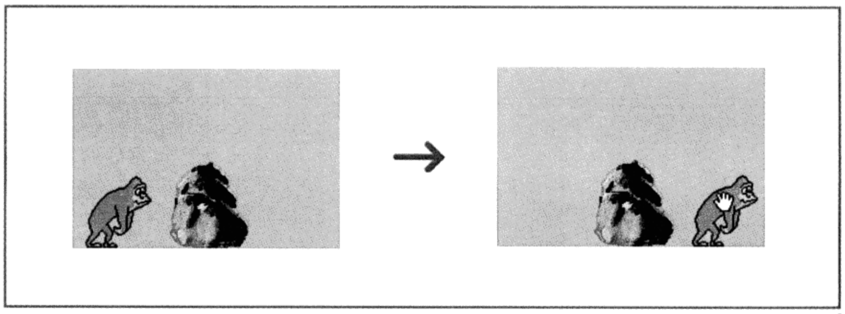


Figure 9. Dragging the monkey to the right of the rock

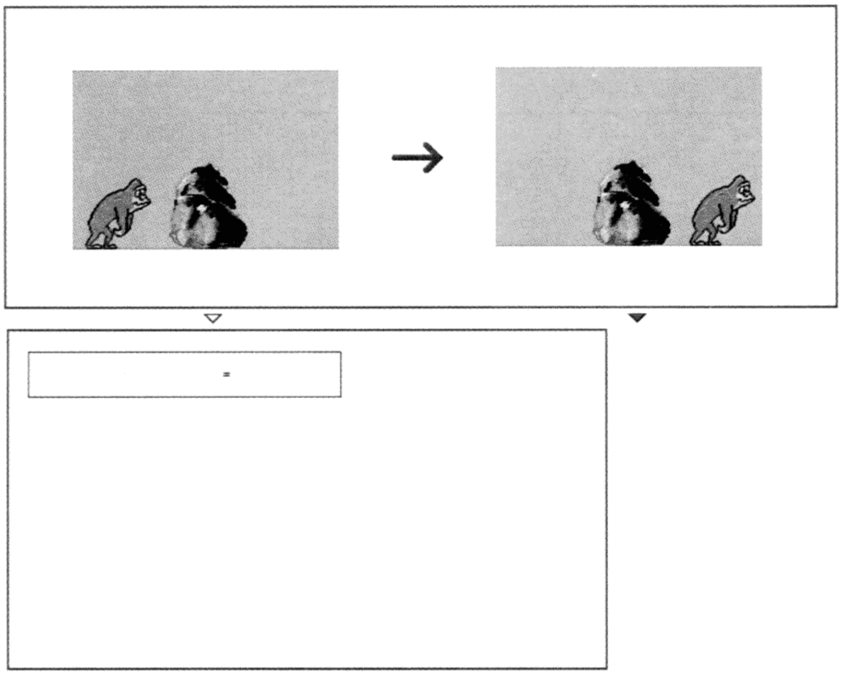


Figure 10. Defining a property test

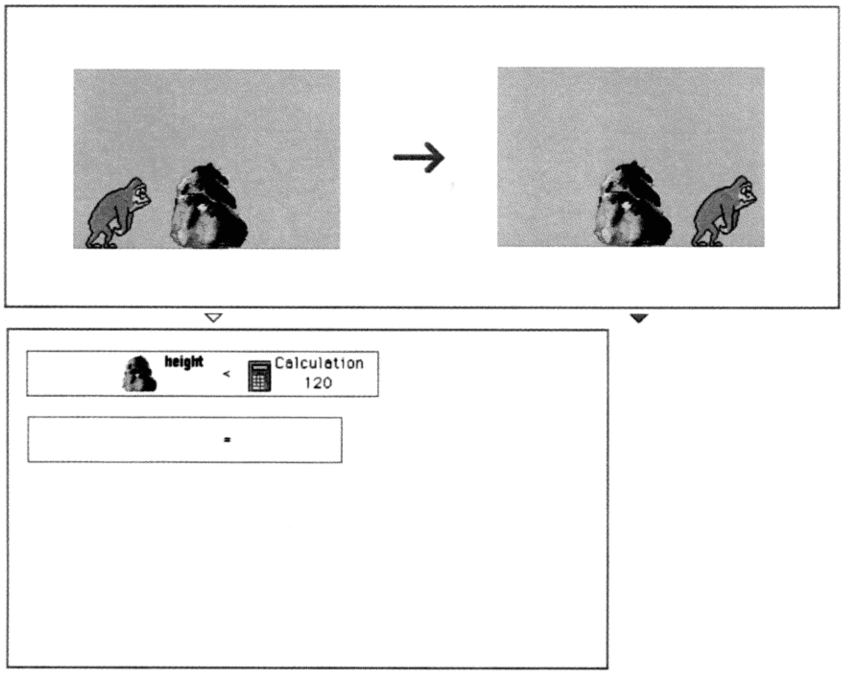


Figure 12. Checking if the rock is less than twice the monkey's height

. . . then . . . else," semicolons, or other language syntax. Yet the effect when executed is that the monkey jumps over the rock. The child has programmed the monkey. This is the essence of programming in KidSim: programming by direct manipulation editing.

Suppose now that the child wants to restrict the monkey to climbing over rocks that are up to twice its height (monkeys being good climbers) but no higher. Suppose the monkey's height is 60, and the height of the current rock is 70. Here's how to do it.

4. The child clicks on the triangle below the left side of the rule (Figure 10). This displays a box in which property tests may be defined. KidSim always provides an empty test.

5. The child drags the height property of the rock into the left side of the test.

6. Since the right side of the test is to contain a calculation, the child displays the KidSim calculator (Figure 11). The child drags the monkey's height property into the calculator

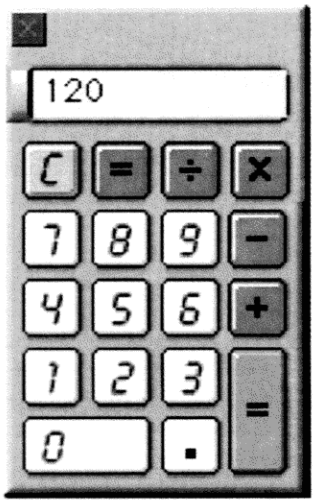


Figure 11. KidSim calculator

display, pushes the multiplication button and the 2 button, then pushes the = button; 120 appears in the display. The child drags this value into the right side of the property test. The rule is shown in Figure 12. Since 70 (the height of the rock) is less than 120 (twice the monkey's height), this rock passes the test. However, other rocks the monkey encounters in its travels may be higher than 120, so this rule would not match, and the monkey could not climb over those rocks.

7. Finally the child closes the rule editor window. A miniature image (Figure 13) of the rule is placed at the top of the monkey's list of rules. This image visually suggests its behavior.

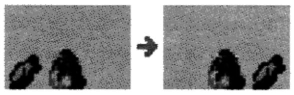


Figure 13. A miniature image

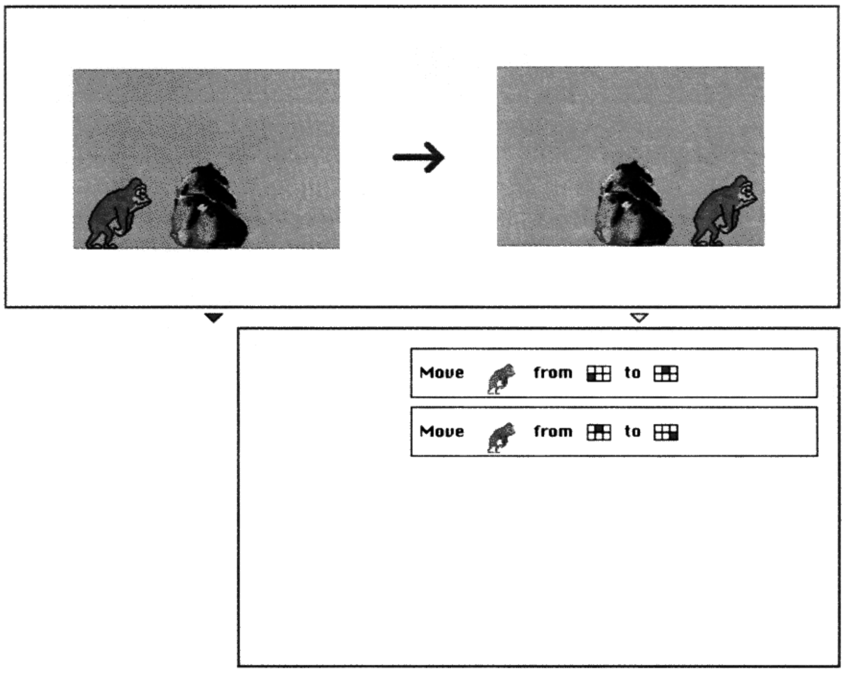


Figure 14. Pictorial display of recorded actions

KidSim can display (upon request) the program that was built as the child edited the right side of the rule (Figure 14).

Now we can define what it means to execute a graphical rewrite rule: when a graphical rewrite rule matches, KidSim executes the program that was recorded by demonstration for it.

The iconic-symbolic representation for the program is an attempt to be close to what children are thinking when they edit—the "minimum translation distance" principle. However, we feel children will rarely want to see this representation, and we make no particular claims for it. It will usually be enough for kids to look at the miniature images of rules to understand what they do. We have found that children can look at dozens of graphical rewrite rules and (a) tell them apart and (b) explain what they do, even rules written by other children (see Figure 15). Here is where combining graphical rewrite rules and programming by demonstration result in a system that is stronger than either. Graphical rewrite rules solve the PBD representation problem, and programming by demonstration solves the rule-semantics problem.

A problem with rule-based systems is that rule order is crucial and often hard to get right. The problem grows with the number of rules. This problem is somewhat mitigated in KidSim because its rules are quite high-level. We have found that we can accomplish interesting tasks in relatively few rules. For example, an optimized strategy for playing the game "MasterMind" requires only about 15 rules in KidSim. Rules can be grouped also into subroutines, thereby forming larger conceptual chunks. Nevertheless, this problem could become serious when the number of rules gets large. We may adopt a strategy like Lisp70's in which rules are automatically factored into a discrimination tree by specificity, which removes the need for children to manually order the rules.

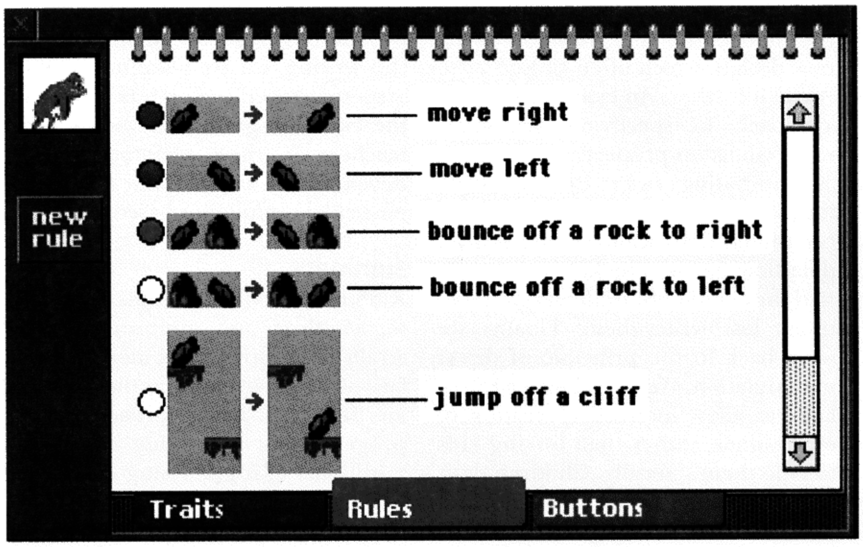


Figure 15. The rules for a monkey. (The annotations are not part of the actual display.)

Of course, graphical rewrite rules really do constitute a programming language. The language has a syntax: left side → right side, and it has an ordering of "statements": top to bottom. Nevertheless, we feel justified in calling KidSim "languageless programming" because of the complete absence of a traditional linguistic syntax such as if-then-else, and because the left and right sides of rules are images of the game board, not abstract representations of it. Furthermore, KidSim follows all of the UI principles listed here, making it feel more like direct manipulation editing than programming.

Kid Tests of KidSim

Over the past two years we have formed a close association with fifth-grade classrooms in two elementary schools. As part of other projects, Apple Computer had endowed both schools with numerous Macintosh computers. Each of the classrooms has about 15 computers, one for every two children. While this ratio is not representative of schools in general, it did provide a good laboratory for experimenting with ways to improve education through technology.

This association with the schools has been essential in the development of KidSim. If you want to design a program for children, then children must participate in the design. Feedback from the students has caused us to change the design of KidSim several times. Each time we went back to the children for their reaction to the new design, which often caused us to revise it further. An example was our approach to specifying arithmetic expressions on property values, such as computing twice the monkey's height. We invented several clever (we thought) notations, most having a data flow flavor. The kids repeatedly said they could not understand them, much less write them. Finally, we went back to the principle of direct manipulation. We introduced a calculator to allow interactive creation of expressions, rather than forcing kids to type them statically. Children drag property values to the calculator and push buttons to operate on them, much as they would with a physical calculator. The calculator metaphor obeys almost all of the good UI principles mentioned earlier—concrete, interactive, direct manipulation, seeing and pointing, familiar conceptual model, and modeless. We found that all the children could use it. (A calculator "tape" is available for displaying the steps should a child want to see them.)

Having a working prototype is also essential in getting feedback from students. They are able to respond more easily when they can try out a design rather than having to imagine how it might work. But even before we got a prototype working, we tested the ability of children to write graphical rewrite rules via "Post-it Notes programming," in which they wrote rules on note pads and then acted out their "programs." Among the 30 fifth graders (10-year-olds) we tested, both boys and girls, none had any trouble writing rules. Furthermore, they responded enthusiastically to the concept. When we gave them new problems, they raced back to their desks, scribbled out a new rule or two, then raced back to us and demanded "Test us now." There was no writer's block as is often observed with programming languages, in which kids do not know how to proceed. This experience has been repeated with the computerized rules.

These efforts do not constitute a formal test of KidSim. Nevertheless, the results are so positive that we are encouraged to think the KidSim approach has promise. At the time of this writing, we are planning to initiate a structured test on 60 children in the two fifth-grade classrooms. The teachers in these classrooms have developed a curriculum around a particular simulation based on [6].

Summary

KidSim is a tool kit that makes it easy for children and nonprogramming adults to construct and modify simulations by programming their behavior. It takes a new approach toward programming by getting rid of the traditional programming language syntax. Drawing on the lessons learned from personal computer user interfaces, KidSim combines two powerful ideas—graphical rewrite rules and programming by demonstration. The result appears to solve the end-user programming problem for some types of simulations.

Ultimately we want to extend KidSim to adult programming tasks (AdultSim?). At the moment we do not know how to do this, and we suspect that the effort required will be nontrivial. However, we do feel that we can characterize the result: all successful end-user programming systems for adults (or kids) will follow the principles we have described.

Acknowledgments
We wish to express our gratitude to the many people who have helped design and test KidSim: to the two marvelous fifth-grade teachers Betty Jo Allen-Conn in the Los Angeles Open Magnet School and Phyllis Lewcock in the Cupertino Stevens Creek Elementary School, as well as to Julaine Salem, a technical support teacher in the Cupertino Union School District who has assisted both; to all their kids who have told us what they like and do not like, especially Brad Harrah, Liz Wertz, Emily Dean, Shane Varaiya, Joey Steger, Colin Yamaoka, Marcus Contro, Elnas Farnam, Alison Lucero, and Andrew Noto; to Alan Kay's Vivarium project for endowing the Open School with computers and innovative ideas and to Apple's Classroom of Tomorrow (ACOT) project for doing the same with the Stevens Creek School; to Apple Computer for its generous support and continuing interest in education; to Ruben Kleiman and the entire SK8 team for making a great language—SK8—in which the KidSim prototype is written; to Alan Peterson, Ramon Felciano, Rodrigo Madanes, and Dave Vronay for programming assistance; to Mark Loughridge for his great graphics, animation, and title ideas. Finally, a special thank you to A.K. Dewdney for the *Planiverse*, the microworld which launched the KidSim effort. ◼

References
1. Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, Reading, Mass., 1992.
2. Baecker, R.M. and Buxton, W.A.S. *Readings in Human-Computer Interaction*. Morgan Kaufmann, Los Altos, Calif., 1987.
3. Cypher, A., Ed. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass., 1993.
4. Cypher, A. Eager: Programming repetitive tasks by demonstration. In *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass., 1993, pp. 205–217.
5. Davis, R. and King, J. An overview of production systems. Rep. STAN-CS-75-524, Computer Science Dept., Stanford Univ., Stanford, Calif., 1975.
6. Dewdney, A.K. *The Planiverse, Computer Contact with a Two-Dimensional World*. Poseidon Press, New York, 1984.

7. diSessa, A.A., and Abelson, H. Boxer: A reconstructible computational medium. In *Studying the Novice Programmer,* E. Soloway and J. Spohrer, Eds. Lawrence Erlbaum, Hillsdale, N.J., 1989 pp. 467–481.

8. Fenton, J. and Beck, K. Playground: An object-oriented simulation system with agent rules for children of all ages. In *Proceedings of OOPSLA '89.* ACM, New York, 1989, pp. 123–137.

9. Furnas, G. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of CHI '91.* ACM, New York, 1991, pp. 71–78.

10. Halbert, D. Programming by example. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, Berkeley, Calif., 1984.

11. Heckel, P. *The Elements of Friendly Software Design.* Sybex, San Francisco, Calif., 1982.

12. Laurel, B., Ed. *The Art of Human-Computer Interface Design.* Addison-Wesley, Reading, Mass., 1990.

13. Maher, M.L., Sriram, D., and Fenves, S.J. Tools and techniques for knowledge-based expert systems for engineering design. *Adv. Eng. Softw. 6*,4 (Oct. 1984), 178–188.

14. Newell, A. and Simon, H.A. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, N.J., 1972.

15. Repenning. A Agentsheets: A tool for building domain-oriented dynamic, visual environments. Ph.D. dissertation, Dept. of Computer Science, Univ. of Colorado at Boulder, Boulder, Colo., 1993.

16. Rychener, M.D. Production systems as a programming language for artificial intelligence. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.

17. Scardamalia, M. and Bereiter, C. Higher levels of agency for children in knowledge building: A challenge for the design of new knowledge media. *J. Learn. Sci. 1*,1 (1991), 37–68.

18. Sloman, A. Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence. In *Proceedings of the Second International Joint Conference on Artificial Intelligence.* 1971, pp. 270–278.

19. Smith, D.C. *Pygmalion, A Computer Program to Model and Stimulate Creative Thought.* Birkhäuser Verlag, Basel, Switzerland, 1977.

20. Smith, D.C., Irby, C., Kimball, R., Verplank, W., and Harslem, E. Designing the Star user interface. *Byte* 7,4 (Apr. 1982), 242–282.

21. Soloway, E., and Spohrer, J. *Studying the Novice Programmer.* Lawrence Erlbaum, Hillsdale, N.J., 1989.

22. Tesler, L., Enea, H., and Smith, D.C. The Lisp70 pattern matching system. In *Proceedings of the Third International Joint Conference on Artificial Intelligence.* 1973, pp. 671–676.

About the Authors:
DAVID CANFIELD SMITH is a senior scientist in Apple Computer's Advanced Technology Group. His research interests include human-computer interaction, educational software, programming language design, programming environments, end-user programming, and getting rid of the "priests" of computing.

ALLEN CYPHER is a research scientist in Apple Computer's Advanced Technology Group. His main areas of interest are in end-user programming and programming by demonstration.

JAMES C. SPOHRER is a program manager in Apple Computer's Advanced Technology Group. His research interests include authoring tools, end-user programming, educational software, and intelligent multimedia titles.

Authors' Present Address: Apple Computer, One Infinite Loop, MS: 301-3D, Cupertino, CA 95014, dsmith@apple.com, cypher@apple.com, spohrer@taurus.apple.com