CHAPTER

One

Novice Programming Comes of Age

DAVID CANFIELD SMITH
Stagecast Software, Inc.

ALLEN CYPHER
Stagecast Software, Inc.

LARRY TESLER
Stagecast Software, Inc.

S
R
L

Abstract

Since the late 1960s, programming language designers have been trying to develop approaches to programming computers that would succeed with novices. None has gained widespread acceptance. We have taken a different approach. We eliminate traditional programming languages in favor of a combination of two other technologies: programming by demonstration (PBD) and visual before-after rules. Our approach is now available as a product named Stagecast Creator, which was introduced in March 1999. It is one of the first commercial uses of PBD. Stagecast Creator enables even children to create their own interactive stories, games, and simulations. Here, we describe our approach, offer independent evidence that it works for novices, and discuss why it works when other approaches haven't and, more important, can't.

1.1 Introduction

The computer is the most powerful tool ever devised for processing information, promising to make people's lives richer (in several senses). But much of this potential is unrealized. Today, the only way most people are able to interact with computers is through programs (applications) written by other people. This limited interaction represents a myopic and procrustean view of computers, like Alice looking at the garden in Wonderland through a keyhole. Until nonprogrammers can program computers themselves, they'll be able to exploit only a fraction of a computer's power.

The limits of conventional interaction have long motivated researchers in end-user programming. An end user in this context uses a computer but has never taken a programming class—a definition describing the vast majority of computer users. We use the term *novice programmer* to describe end users who want to program computers. Is *novice programmer* an oxymoron? Is it a reasonable goal? Certainly there are "novice document writers," "novice spreadsheet modelers," and even "novice Internet surfers." But in more than thirty years of trying, no one has come up with an approach that enables novices to program computers. Elliot Soloway, director of the Highly Interactive Computing Project at the University of Michigan, estimates that even for novices who do take a programming class, less than 1 percent continue to program when the class ends. We'll explore the reasons

S
R
L

for this, but first we explore our own new approach to programming that seems to work for novices.

Stagecast Creator, a novice programming system for constructing simulations from Stagecast Software, Inc., founded by the authors and others in 1997, is the culmination of a seven-year research and development effort, the first five at Apple Computer (Cypher and Smith 1995; Smith, Cypher, and Spohrer 1994; Smith and Cypher 1995; Smith, Cypher, and Schmucker 1996). The project, initially called KidSim, was later renamed Cocoa, and finally became Creator. The goal was to make computers more useful in education. For a variety of reasons, the co-inventors of Creator—the authors Smith and Cypher—focused on simulations, a powerful teaching tool for making abstract ideas concrete and more understandable. Simulations encourage experimentation, helping children develop sequential, causal reasoning—in other words, the scientific method. The goal of the Creator project evolved into empowering end users—teachers and students—to construct and modify simulations through programming.

Our initial approach was much like that of other language developers: to invent a programming language that would be acceptable to end users. We tried a variety of syntaxes; all failed dismally. That experience, together with the history of programming languages during the past thirty years—from Basic to Pascal, from Logo to Smalltalk, from Hypertalk to Lingo—convinced us we could never come up with a language that would work for novices.

Our first insight was that language itself is the problem and that any textual computer language represents an inherent barrier to user understanding. Learning a new language is difficult for most people. Consider the years of study required to learn a foreign language, and such languages are natural languages. A programming language is an artificial language that deals with the arcane world of algorithms and data structures. We concluded that no conventional programming language would ever be widely accepted by end users.

1.2 Programming without a Textual Programming Language

How can a computer be programmed without a textual programming language? Our solution combined two existing techniques: PBD and visual before-after rules. In PBD, users demonstrate algorithms to the computer by operating the computer's interface just as they would if they weren't

S
R
L

programming. The computer records the user's actions and can then reexecute them later on different inputs. PBD's most important characteristic is that everyone can do it. PBD is not much different from or more difficult than using the computer normally. This characteristic led us to consider PBD as an alternative approach to syntactic languages.

A problem with PBD has always been how to represent a recorded program to users. It's no good allowing users to create a program easily and then require them to learn a difficult syntactic language to view and modify it, as with most PBD systems. In Creator, we first sought to show the recorded program by representing each step in some form, either graphically or textually. Some of the representations were, in our opinion, elegant, but all tested terribly. Children would almost visibly shrink from their complexity. We eventually concluded that no one wanted to see all the steps; they were just too complicated.

Our second insight was not to represent each step in a program; instead, Creator displays only the beginning and ending states. Creator does in fact have a syntax—the lists of tests and actions in a rule—but people can create programs for a long time without even being aware of this syntax. This feature is dramatically different from conventional languages, in which users must know whether a routine is called "move" or "go," the order of the parameters and where all the various quotation marks, semicolons, and parentheses belong.

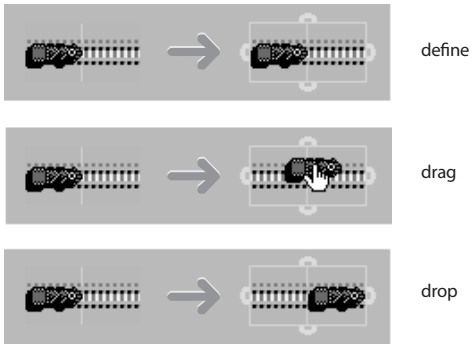
As an example of the Creator approach, suppose we want the engine of a train simulation to move to the right. We move the engine by defining a visual before-after rule for the engine. Rules are the Creator equivalent of subroutines in other languages. Each rule represents an arbitrary number of primitive operations, or statements in other languages. Visually, Creator shows a picture of a small portion of the simulation on the left, then an arrow, and then a picture of what we want the simulation to look like after the rule executes. Figure 1.1 shows the interactive, visual process of creating a rule by demonstration.

First, we define the initial rule. Notice that the left and right sides start out the same; all rules begin as identity transformations. Users define the behavior of the rule by demonstrating changes to the right side. Here, we grab the engine with the mouse and drag it to the right. When we drop the engine, it snaps to the grid square it is over. That's all there is to it. Nowhere did we type *begin-end, if-then-else,* semicolons, parentheses, or any other language syntax. The rule we just created may be read as follows:

If the engine is on a piece of straight track and there is straight track to its right, then move the engine to the right.

S
R
L

FIGURE **1.1**



Defining a rule by demonsration.

Notice that programming is kept in domain terms, such as *engines* and *track,* rather than in computer terms, such as *arrays* and *vectors*. Also, instead of dealing with objects indirectly through coordinates, users program them by manipulating them directly; that is PBD (see Table 1.1).

Since a rule in Creator may not show all the steps involved, just their beginning and ending states, it is not a representation for the steps, suggesting instead the *effect* of the rule. The rule acts as a memory jogger for users. This turned out to be the key technique in Creator for helping users understand recorded programs, even those written by others.

A similar commercial software development system called AgentSheets, developed by Alexander Repenning (1993) at the University of Colorado in Boulder, also uses visual before-after rules (see Repenning and Perrone's chapter, "Programming by Analogous Examples," which also gives an example using a train). A delightful system, it is the closest to Creator of any software system we know of.

1.3 Theoretical Foundations

Why does Creator's approach to programming apparently work where syntactic languages don't? We hinted at the answer earlier. An essential ingredient is certainly the PBD technique, which eliminates the need for any syntactic language during program construction. The technique of using visual before-after rules finishes the job, eliminating the need for any syntactic

TABLE **1.1**

Examples of the kinds of operations that can be recorded by demonstration.

Operation	*What the user does*	*What the computer records*
Move	Drag an object with the mouse	Move <object> to <location>
Create	Drag an object from the Character Drawer into the rule	Create <object> at <location>
Delete	Select an object by clicking on it and press the Delete ke	Delete <object>
Set Variable	Double-click on an object to display its variables, select a variable's value, and type a new value	Put <value> into <object>'s <variable>

language for program representation. But why would these two techniques be acceptable to the typical novice programmer? The answer is interesting, illustrating why traditional approaches haven't and, more important, *can't* work.

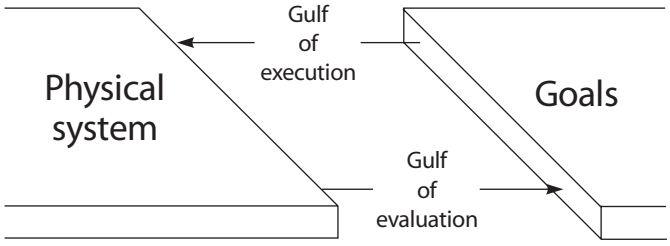
The main problem novice programmers have when programming computers is the gap between the representations the brain uses when thinking about a problem and the representations a computer will accept. "For novices, this gap is as wide as the Grand Canyon," as Don Norman documented in his 1986 book *User Centered System Design* (see Figure 1.2). He argued that there are only two ways to bridge the gap: move the user closer to the system or move the system closer to the user. Programming classes try to do the former. Students are asked to learn to think like a computer. This radical refocusing of the mind's eye is difficult for most people. Even if they learn to do it, they don't like where they end up. They don't want to think like a computer; they want to use computers to accomplish tasks they consider meaningful.

In Creator, we've tried to do the opposite of what programming classes do—we want to bring the system closer to the user. We did this by making the representations used when programming the computer more like the representations used in the human brain. To do this, we needed a theory of the brain's representations that would be helpful to us. We found two: one developed by Aaron Sloman, the other by Jerome Bruner.

1.3.1 Sloman's Approach

In 1971, Aaron Sloman divided representations into two general types: analogical and "Fregean," after Gottlob Frege, the inventor of predicate

FIGURE **1.2**



The "Grand Canyon" gap between human and computer.

calculus. In an analogical representation, Sloman wrote, "the structure of the representation gives information about the structure of what is represented" (Sloman 1971, 273). A map is an example; from a map, one can tell the relationships between streets, the distance between two points, the locations of landmarks, and which way to turn when you come to an intersection.

By contrast, Sloman (1971) wrote, "In a Fregean system there is basically only one type of 'expressive' relation between parts of a configuration, namely the relation between 'function-signs' and 'argument-signs.' . . . The structure of such a configuration need not correspond to the structure of what it represents or denotes" (273). We can, for example, represent some of the information in a map through predicate calculus statements, such as

```
g: "Gravesend"
u: "UnionVille"
m: "Manhattan Beach"
s: "Sheepshead Bay"
East(g, u)
EastSouthEast(s, g)
South(m, s)
```

> The generality of Fregean systems may account for the extraordinary richness of human thought. . . . It may also account for our ability to think and reason about complex states of affairs involving many different kinds of objects and relations at once. The price of this generality is the need to invent complex heuristic procedures for dealing efficiently with specific problem-domains. It seems, therefore, that for a frequently encountered problem domain, it may be advantageous to use a more specialized mode of representation richer in problem-solving power. (Sloman 1971, 274)

Most programming languages use Fregean representations, aiming to be general and powerful. Creator emphasizes ease of use over generality and power, and so it has adopted analogical representations. Although Creator is "Turing-equivalent," meaning it can compute anything, it addresses only the specialized problem domain of visual simulations. It doesn't try to do everything well but is very good at what it does. A better way to describe it than Turing-equivalent may be "PacMan-equivalent." Creator is powerful enough to let kids program the game PacMan. That's all we're trying to do.

Creator uses analogical representations in its rules. For example, a rule for moving a train engine, as shown in Figure 1.1, can do the same thing as Fregean HyperTalk code, which can include dozens of arcane commands, as in the following list which goes on for another seventy lines. It is obvious which is easier to understand.

```
on runTrain
  global AutoSwitch,BtnIconName,PrevBtnIconName
  global Dir,PrevDir,LastLoc,PrevLocs,LookAhead,
         TheNextMove
  global LastMoveTime,SoundOff,MoveWait,Staging,
         TheStage,TheEngine
  global TheMoves,Choices,Counter,EngineIcon,XLoc
  —This routine is long.
  —Most of the code is inline for acceptable speed
  lock screen
  setupTrain
  unlock screen
     repeat
     —check user action often
        if the mouseClick then checkOnThings the clickLoc
     —get iconName of current position
     put iconName(icon of cd btn LookAhead) into
          BtnIconName
     if the number of items in BtnIconName > 1 then
        put "True" into Staging
        if TheStage = 0 then put BtnIconName into
          PrevBtnIconName
        if BtnIconName contains "roadXing" then put
          LookAhead into XLoc
        if BtnIconName contains "Rotatetrain" then put 1
          into TheStage                                    S
     end if                                                R
                                                           L
```

```
        if the mouseClick then checkOnThings the clickLoc
        put LastLoc & return before PrevLocs
        put LookAhead into LastLoc
        put Dir & return before PrevDir
        if the mouseClick then checkOnThings the clickLoc
        add 1 to Counter
    . . .
```

1.3.2 Bruner's Approach

In 1966, the educational psychologist Jerome Bruner (1966) asserted that any domain of knowledge can be represented in three ways:

- "By a set of actions appropriate for achieving a certain result ('enactive' representation). We know many things for which we have no imagery and no words, and they are very hard to teach to anybody by the use of either words or diagrams and pictures." For example, you can't learn to swim by reading a book.
- "By a set of summary images or graphics that stand for a concept without defining it fully ('iconic' representation)." For example, children learn what a horse is by seeing pictures of horses or actual living horses.
- "By a set of symbolic or logical propositions drawn from a symbolic system that is governed by rules or laws for forming and transforming propositions ('symbolic' representation)."

The first two ways are analogical representations; the third is Fregean. Jean Piaget, the noted Swiss psychologist best known for his work in the developmental stages of children, believed that children grow out of their early enactive and iconic mentalities and that becoming an adult means learning to think symbolically. By contrast, Bruner recommends encouraging children to retain and use all three mentalities—enactive, iconic, and symbolic—when solving problems. All three are valuable in creative thinking.

Creator seeks to involve all three mentalities in programming. The enactive mentality is involved in PBD when users manipulate images directly; drag-and-drop functions are enactive. The iconic mentality is involved in visual before-after rules and the domain of visual simulations. Finally, the symbolic mentality is involved in Creator's use of variables,

which can help model deeper semantics in simulations. For example, predator-prey-type simulations can be modeled through variables.

1.4 Empirical Evidence

We've also gathered evidence that the Creator approach to programming works with novices. This evidence has taken three forms: informal observation, formal user studies, and anecdotal user reports.

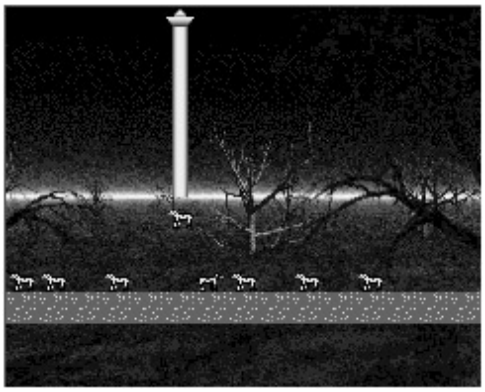
Teachers and parents who worked with prerelease versions of Creator used it for years. We and our associates conducted hundreds of hours of direct tests on children and adults for the past five years, most on children ages six to twelve in school settings.

We implemented three computer prototypes of Creator, each smaller and faster and closer to product quality than the previous one, testing each on progressively larger audiences of novice users, and the final prototype—Cocoa—to an audience of hundreds of novice users. We distributed Cocoa through the Internet, just as we have with Creator, but our most important source of information was longitudinal studies in several elementary school classrooms in California. Teachers integrated the prototypes into year-long curriculums designed to improve their students' problem-solving skills. They contrived problems that required programming for their solutions; one had her class program ocean science simulations. Our most gratifying success was when the students in one class asked to extend the school year so they could continue to work on their simulations. For the next six weeks of vacation, a third of the class continued to come to school once a week to program. The surprising thing is not that two-thirds of the children decided not to participate but that any of them wanted to keep going to school during summer vacation. These kids did not find programming an onerous task.

Independent researchers at several universities in the United States and England conducted formal user studies of the Creator prototypes KidSim and Cocoa (Bruner 1966; Gilmore et al. 1995; Rader, Brand, and Lewis 1997; Sharples 1996). While each identified areas for improvement, all answered affirmatively what we consider the two most important questions: Can kids program with this approach? Do they enjoy it?

The studies found that within fifteen minutes, most novice-user children were able to create running simulations with moving interacting objects. The studies found no gender bias: girls and boys enjoy Creator

FIGURE **1.3**



Alien Abduction.

equally. The studies suggest that the technology is usable by novices and is flexible enough for implementing a variety of ideas.

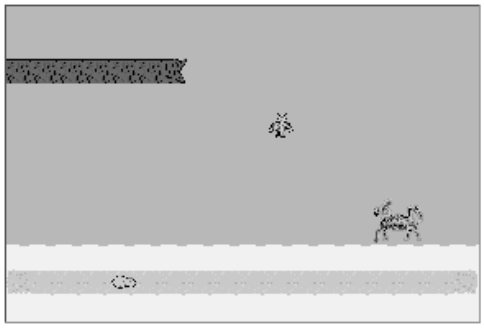
One of our early concerns was whether Creator would have enduring interest for children. We've now heard from some users and their parents and teachers that it does. For example, in Cedar Rapids, Iowa, Steve Strong, who teaches computer programming to students ages fourteen to seventeen, lets each one choose the language he or she would like to learn, including C, Java, and Creator. Since adding Creator to the curriculum, he reports that as many girls as boys now take his course; students who use Creator have well-developed projects to show at the end of the class, whereas those using traditional languages typically have only a small part of their project implemented. Moreover, students learning Creator first and other languages later are better programmers than those who go directly to a traditional language.

Figure 1.3 shows a hilarious game created by a twelve-year-old boy in which a spaceship beams up cows. A user controls the ship's direction with the arrow keys and the beam with the space bar. The goal is to beam up all the cows. Because of the high-level nature of the Creator rules, this game required only thirteen rules to implement.

Figure 1.4 shows a model created by an eleven-year-old girl of the way owls hunt mice in winter, when mice dig tunnels under the snow. But the owl had better watch out, because a wolf wants to eat it. The player controls

S
R
L

FIGURE **1.4**



Olivia's Owl.

the owl with the keyboard's arrow keys. The trick is to drop the owl on the mouse just as it passes under its claws. The goal is to catch five mice without getting eaten yourself. This game (actually two games in one) required fifty-seven rules.

1.5 Conclusion

Early evidence suggests the approach to programming being pioneered by Creator is more acceptable to novice programmers than traditional approaches. Creator uses PBD, which is inherently enactive and iconic, for program construction. It also uses an analogical representation—visual before-after rules—for its programs. The programming domain is limited to visual simulations, helping Creator bring the system closer to the user. In summary, Creator shifts the language design emphasis from computer science to human factors.

References

Bruner, J. 1966. *Toward a theory of instruction.* Cambridge, Mass.; Harvard University Press.

S
R
L

Brand, C., and C. Rader. 1996. How does a visual simulation program support students creating science models? In *Proceedings of IEEE Symposium on Visual Languages* (Boulder, Colo., Sept. 3–6). Los Alamitos, Calif.: IEEE Computer Society Press.

Cypher, A., and D. Smith. 1995. KidSim: End user programming of simulations. In *Proceedings of CHI'95* (Denver, Colo., May 7–11). New York: ACM Press.

Gilmore, D., K. Pheasey, J. Underwood, and G. Underwood. 1995. Learning graphical programming: An evaluation of KidSim. In *Proceedings of Interact'95* (Lillehammer, Norway, June 25–30). London: Chapman & Hall.

Norman, D. 1986. Cognitive engineering. In *User centered system design: New perspectives on human-computer interaction,* ed. D. Norman and S. Draper. Hillsdale, N.J.: Erlbaum.

Rader, C., C. Brand, and C. Lewis. 1997. Degrees of comprehension: Children's mental models of a visual programming environment. In *Proceedings of CHI'97* (Atlanta, Ga.). New York: ACM Press.

Repenning, A. 1993. AgentSheets: A tool for building domain-oriented dynamic visual environments. Ph.D. diss. University of Colorado, Boulder; see www.agentsheets.com.

Sharples, M. 1996. How far does KidSim meet its designers' objectives of allowing children of all ages to construct and modify symbolic simulation? Report of the School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, England.

Sloman, A. 1971. Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (London)*. San Francisco: Morgan Kaufmann.

Smith, D., and A. Cypher. 1995. KidSim: Child-constructible simulations. In *Proceedings of Imagina '95* (Monte Carlo, Feb. 1–3). Institut National de l'Audiovisuel.

Smith, D., A. Cypher, and K. Schmucker. 1996. Making programming easier for children. In *The design of children's technology,* ed. A. Druin. San Francisco: Morgan Kaufmann. See also *Interactions of ACM 3,* no. 5 (September–October 1996): 58–67.

Smith, D., A. Cypher, and J. Spohrer. 1994. KidSim: Programming agents without a programming language. *Common. ACM* 37, no 7 (July): 54–67.

S
R
L

S
R
L