Programming Environments for Novices

Mark Guzdial
College of Computing, Georgia Institute of Technology
`guzdial@cc.gatech.edu`

May 7, 2003

1 Specializing Environments for Novices

The task of specializing programming environments for novices begins with the recognition that programming is a hard skill to learn. The lack of student programming skill even after a year of undergraduate studies in computer science was noted and measured in the early 80's [32] and again in this decade [20]. We know that students have problems with looping constructs [31], conditionals [8], and assembling programs out of base components [33]—and there are probably other factors, and interactions between these factors, too.

What are the critical pieces? What pieces, if we "fixed" them (made them better for novice programmers), would make programming into a more manageable, learnable skill? If we developed a language that changed how conditionals work or loops, or make it easier to integrate components, would programming become easier? That's the issue that developers of educational programming environments are asking.

Each novice programming environment (or family of environments) is attempting to answer the question, "What makes programming hard?" Each answer to that question implies a family of environments that address the concern with a set of solutions. Each environment discussed in this chapter attempts to use several of these answers to make programming easier for novices.

Obviously, there are a great many answers to the question "What makes programming hard?" For each answer, there are a great many potential environments that act upon that answer, and then there are a great many other potential environments that deal with multiple answers to that question. That's not surprising, since it's almost certainly true that there is no one correct answer to the question that applies to all people.

Not all of these potential environments have been built and explored, however. The field of Computer Science Education Research is too new, and there are too few people doing work in this field. We are still in the stage of the field of identifying *potential* answers to key questions—indeed, even figuring out what the key questions are!

Nonetheless, there *are* many novice programming environments that have been built, and not all can be discussed in a short primer. Instead, this chapter will focus on three families that have been particularly influential in the development of modern environments and in the thinking of the CS Ed research community.

- The *Logo* family of programming environments, that began as an off-shoot of the AI-programming language *Lisp* and spawned a rich variety of novice programming environments.
- The *rule-based* family of programming environments, that drew from both Logo and *Smalltalk-72*, but even more directly, *Prolog*.
- The *traditional programming language* family of novice programming environments, which tried *not* to change the language, but instead provide new student-centered supports for existing programming languages.

The audience for these environments ranges from young school children for the Logo environments to undergraduate university students for some of the traditional programming language environments. In this chapter, the issue of student differences (e.g., age, background, motivation) is simply glossed over. Such a huge simplification is acceptable in this situation because the problem is so hard. No matter what the age of the students, programming is hard to learn. Whether students attempt to learn to program at a young age or at the age of young adults, the tasks and difficulties remain similar. The environments in the sections below are attempting to deal with those challenges at whatever the age of the student audience.

2 Logo and its Descendants: The Goal of Computational Literacy

Logo was developed in the mid-1960's by Wally Feuzeig and Danny Bobrow at BBN Labs, in consultation with Seymour Papert at nearby MIT. Logo was designed to be "Lisp without parentheses." Lisp was a popular programming language for artificial intelligence programming. Lisp was known for its flexibility and the ease with which data could become program, or vice-versa, making it very easy for programs to manipulate their own components. Lisp was especially good for creating and manipulating representations of knowledge. (See Figure 1 for the family tree of this section.)

The answer to the question of "What makes programming hard?" for the Logo developers was another question. When Logo was first being developed, people didn't *know* that programming was going to be so hard for so many. Programming was still a curiosity, an activity practiced only by the few who had access to the still-rare machines. The Logo developers asked instead "Why should students program?"

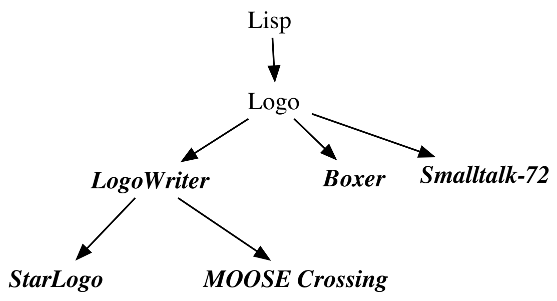


Figure 1: The Logo family of novice programming environments (italics indicate real environments tuned to novices)

The answer to that question was related to Piagetian thinking about learning. Seymour Papert worked for a time in Jean Piaget's laboratory. The goal for Logo was for students to think about their own thinking, by expressing themselves in their programs and then debugging the programs until they worked [23]. By debugging their programs, the argument went, the students were debugging their own thinking. Logo proponents argued that Logo then led students to thinking about their own thinking and gain higher-order thinking skills (such as skills at debugging and planning).

Students' early activities with Logo involved mathematical and word games. Logo was especially strong at playing games with language, e.g., creating a Pig Latin generator or a Haiku generator. Later, a robot "turtle" was added to the Logo environment, so that students could control the robot with commands like `forward 1` to move the robot forward a little bit, `right 1` to turn right one degree, and `pendown` so that the robot's pen (literally, a pen attached to the robot) would draw on the surface below the robot as it moved. With the turtle, Logo could be used for graphics, and in reverse, graphics could be used to understand mathematics. When Logo was moved onto graphical user interfaces (the first uses of Logo were on paper-scrolling teletype terminals), the turtle went with it, but as a graphical object that left graphical pen trails on the screen. With turtle graphics (using the turtle to draw), mathematics, and language support in Logo, student programs in Logo *could* range over a broad set of knowledge areas.

However, Logo soon became intimately linked with turtle graphics. For many, Logo was *only* turtle graphics, and any program that offered turtle graphics was consequently some form of Logo. Logo proponents and researchers pointed out that turtle graphics was significant in its complexity and scope. A book by Abelson and diSessa pointed out how much of mathematics could be addressed through the geometry available through turtle graphics, often in interesting and even novel ways [1]. Nonetheless, the close relationship between Logo and the turtle led some to believe that Logo was limited to just explorations of geometry.

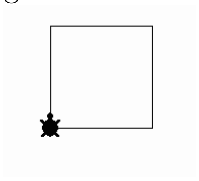
Logo use grew through the early 1980's. Many different forms of Logo were developed, and many are still available today [1]. Because of its linkage with turtle graphics, Logo was fairly popular in science and math classes, but not much farther. Further, questions started to be raised about whether students were really learning to program [18], and what benefits (if *any*) were to be gained from learning to program [24, 14]. In general, studies of the relationship between higher-order thinking skills and programming never showed any significant correlation [22].

2.1 Examples of Logo

The basic commands of Logo could be combined and iterated using recursion or simple looping constructs, like `repeat`. Executing:

```
repeat 4 [fd 100 rt 90]
```

generates:



The syntax of Logo is simple and sparse, like Lisp. It has few special characters. Most of the syntax derives directly from the provided procedures: If you know the procedure, you can figure out how the statements parse. The argument for this kind of syntax was to make it simple for the students to learn the rules and understand the programs.

There are no end line markers (e.g., semi-colons) in Logo. Instead, each procedure (like `fd`) knows how many inputs it needs. The parsing and evaluation are tied tightly in Logo. Code can be contained in lists which are delimited with square brackets (`[ ]`). Thus `repeat` is a function that takes two inputs: A number of iterations to execute, and a list of code to execute (evaluate) that many times.

We can define procedures to "teaching the turtle" to do something (the language used to explain to children what programming was about). Here is the procedure used for defining the word `square` to mean the procedure of drawing a square.

```
to square
  repeat 4 [fd 100 rt 90]
end
```

We can now generate the square with `square`. By parameterizing the `square` procedure, we can draw squares of different sizes.

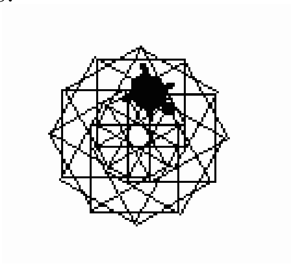
[1] `http://el.media.mit.edu/logo-foundation/products/software.html`

```
to square :size
  repeat 4 [fd size rt 90]
end
```

We can now generate the square with `square 100`.

The syntax for specifying parameters in Logo is drawn from the general syntax for variables. Unlike most programming languages, Logo remained close to its Lisp roots and distinguished between the *value* of the variable and the *name* of the variable. Logo proponents argued that such distinctions improved students' understanding of what the programs were doing. We can see the use of parameterization by exploring the square procedure the way that children were expected to play with squares.

If we move the turtle slightly and then repeat the `square` procedure, we can get interesting designs. `repeat 100 [square 100 fd 10 rt 30]` generates a figure like this:



Language play is just as natural in Logo. In the below example, the procedure `start` defines three lists. Each time that `makeOne` is executed, a random (`pick`) element from the list is selected and composed into a sentence (`se`) which is then shown (`show`). The result are sentences like "Mommy runs quickly" and "Daddy jumps high". We see here the syntax for defining a variable (`make`) requires the name of the variable (e.g., `"names`) versus the syntax for accessing the variable (`:names`).

```
to start
  make "names [Matthew Katie Jenny Mommy Daddy]
  make "verbs [runs eats jumps poops walks sits]
  make "adverbs [high quickly perfectly slowly peacefully]
end

to makeOne
  show (se pick :names pick :verbs pick :adverbs)
end
```

2.2 Programming in support of a task

The next step in the evolution of Logo was to consider "What tasks did students want to use programming for?" Or, to build upon the core question of Logo, "What domains did students want to learn about *through* programming?" The first versions of Logo basically offered a programming environment that was a variation of a command line: A graphical area was visible.

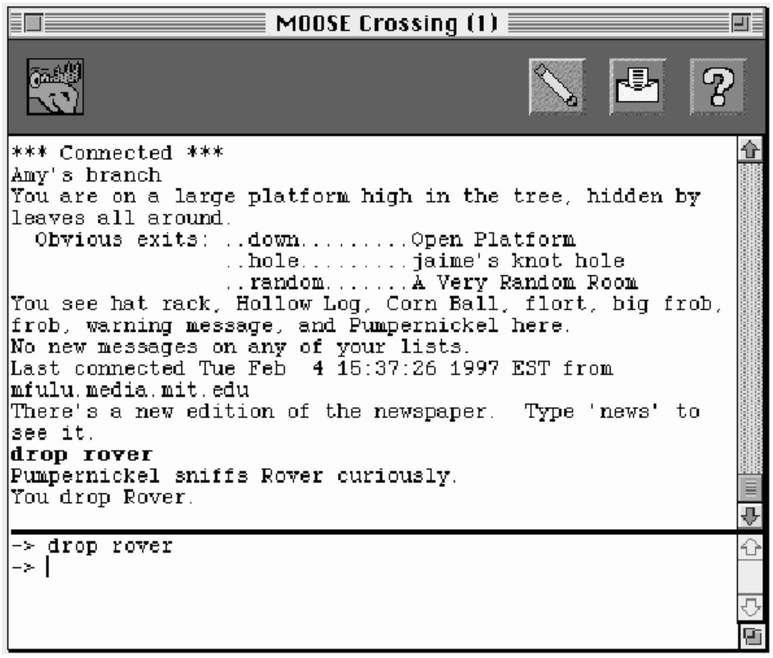
The first kind of Logo that really changed the students' programming environment *LogoWriter*. LogoWriter integrated a word-processor, capable of both graphics and text, with a Logo interpreter. From a language perspective, the LogoWriter turtle could now act as a cursor changing letters beneath it, or stamping graphics onto the page. From an environment perspective, LogoWriter felt as much like an applications program as a programming language. Now, students could do language manipulation where they *saw* the language manipulation (as the cursor moved and the words changed), and create programs that constructed mixed text-and-graphics the way that they might in other applications software. LogoWriter took seriously providing task support so that the range of potential domains to explore with programming was as broad as possible.

LogoWriter was used by Idit Harel in her thesis studies where she had fourth-graders (10 years old) building software to teach fractions to third-graders [13]. Her fourth-graders created software that wasn't too sophisticated, but did mix text and graphics utilizing the special features of LogoWriter. In the end, the fourth-graders learned significantly more about fractions than a comparison group of fourth-graders taking a traditional math curriculum.

StarLogo followed along the path of extending the language and the environment to focus on a particular *kind* of task. StarLogo supported exploration of distributed environments [27]. Mitchel Resnick provided students with not one turtle, but literally thousands of turtles—all running essentially the same, small program. He also introduced the notion of a *patch*, a spot on the screen that could hold state (such as a color) and could run programs, but could not move as a turtle could. By using patches to represent food for ants or wood chips for termites, StarLogo could be used to explore how ants (turtles) gather food or how termites (turtles) create piles, all without coordination but through the power of simple, distributed programs.

2.3 MOOSE Crossing: Practical Logo for Communities

MOOSE Crossing (by Amy Bruckman) again tuned Logo to a particular domain and task, but a social one rather than a scientific or academic task. MOOSE Crossing is a shared, textual, virtual reality. Students sign on to MOOSE with specialized client software and explore a world created by peer students (all under 12 years old)—and extend the world themselves. Students might create specialized rooms where everything said in the room is turned into Pig Latin, or specialized objects like pet dragons that follow their owners around. Students move around, control their world, and interact through Logo-like commands. These commands can then be strung together in procedures such that the dragon "wags its tail" (i.e., displays the words "dragon wags its tail" to all those in the same room of the virtual space) when the dragon is "pet" (i.e., some user in the same room types "pet the dragon.") The turtle



```
*** Connected ***
Amy's branch
You are on a large platform high in the tree, hidden by
leaves all around.
  Obvious exits: ..down.........Open Platform
                 ..hole.........jaime's knot hole
                 ..random.......A Very Random Room
You see hat rack, Hollow Log, Corn Ball, flort, big frob,
frob, warning message, and Pumpernickel here.
No new messages on any of your lists.
Last connected Tue Feb  4 15:37:26 1997 EST from
mfulu.media.mit.edu
There's a new edition of the newspaper.  Type 'news' to
see it.
drop rover
Pumpernickel sniffs Rover curiously.
You drop Rover.
```

```
-> drop rover
->
```

Figure 2: The MOOSE client running on a Macintosh

is replaced with text describing the student-created world (Figure 2).

Bruckman made several changes in the language for MOOSE, based on the experience of years of work in Logo and her concern about making the language accessible to novice programmers. For example, she decided to remove the distinction between the name and the value of the variable. Variables were simply known by name, and whether the value or name was being accessed was determined by context, as in more traditional programming languages. Bruckman used the design principle of "Prefer intuitive simplicity over formal elegance" in make this tradeoff [6]. She made similar tradeoffs elsewhere in her language. Where `say ''Hi there!  How is your project coming?''` was acceptable in MOOSE, so was `say Hi there!  How is your project coming?` in recognition that students had difficulty with syntax like matching quotes.

Bruckman's design principle is an important recognition of what novices understand and what's needed to understand abstractions. The simple observation that students are not the same as experts is common in the cognitive science literature, especially with respect to novice and expert software designers and programmers [16] [2]. A programming language or environment feature that may make sense for experts may be confounding for novices. Current learning theory, based on the work by Jean Piaget, suggests that students need concrete experiences before they can understand abstractions on the experiences [34]. The abstraction of separating names and values may make more sense to experts who have significant experiences than students facing their first programming environment.

Bruckman found that students did learn programming in this environment, supported and motivated by the social context [4]. Students in MOOSE Crossing were able to communicate with one another, share their creations, and even teach each other to program with no external (e.g., adult) support–a surprising event that she documented in a detailed case study. Her studies showed that MOOSE defeated some gender stereotypes by showing that girls were just as successful at programming as boys, if the context is motivating [5]. This was a strong finding in favor of Papert's initial premise that it was what one did with programming that really mattered most.

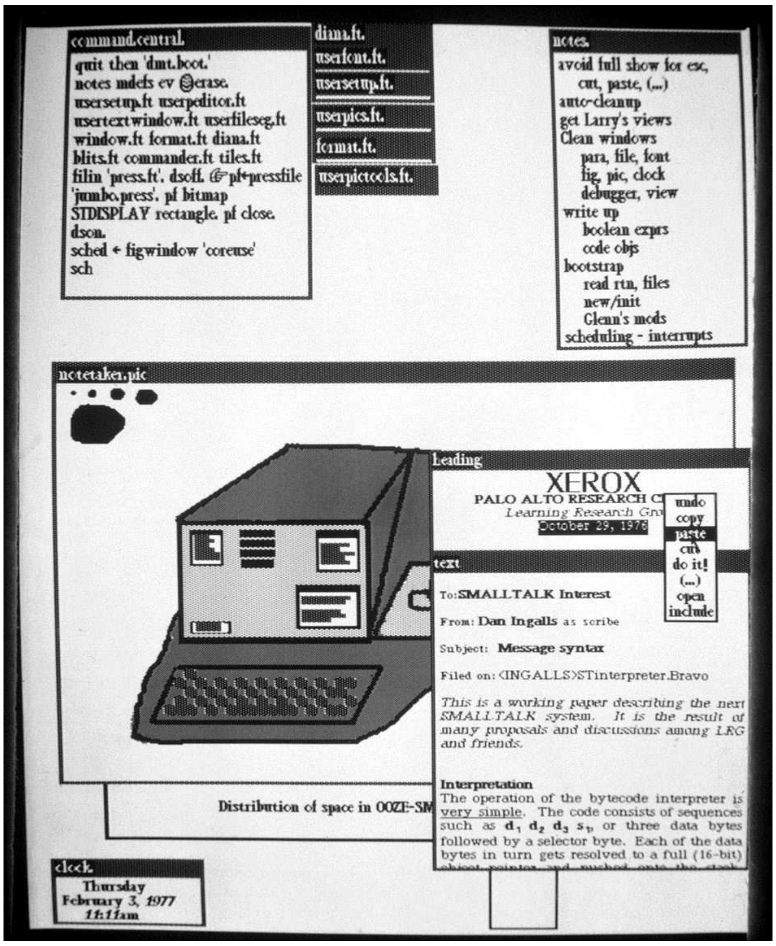
2.4 Boxer and Smalltalk: Extending Beyond Logo

Andrea diSessa also extended Logo, but in a different direction. Rather than tune it to a specific task, he tried to think about what computation would look like if it were a real *literacy*—as ubiquitous as text reading and writing is today [7]. Boxer was based on a principle of *naive realism*: Every object in the system has an on-screen graphical representation that can be inspected, modified, and extended. For example, variables are not just names in Boxer. Creating a variable creates a named *box* on the screen which corresponds to that variable. Setting the variable's value changes (visibly) the contents of the corresponding box. Changing the contents of the box (with direct manipulation and typing) changes the value of the variable. Lists exist in Boxer, but so do

diSessa answers the question about "What's hard about programming?" with the answer, "The interface and its relation to the language" Too much is abstract and hidden in traditional programming languages. Boxer both makes the system easy to understand (because of naive realism) and easy to apply to domains because, like LogoWriter, it plays upon similarity to applications software.

However diSessa is also answer the question with the answer, "The culture." Programming will also be challenging, but no more challenging than learning to read and write. If programming skill was something that one started at an early age, and it was something that *everyone* did, it would be easier for students to pick up. The interesting question is what such computational literacy means for a society. Does science and mathematics become easier to learn because everyone has the computational skills to develop models and visualizations to explore and better understand complex concepts?

Smalltalk-72, by Alan Kay, Dan Ingalls, Adele Goldberg, and other members of the Xerox PARC Learning Research Group, extended the model of Logo in several different ways. Smalltalk was developed along the path to creating the *Dynabook*, a computer whose purpose is to support learning through creation and exploration of the rich range of media that a computer enables [17]. Kay agreed with Papert that computers should be used by students for knowledge expression and learning through debugging of those expressions. However, he felt that the computational power provided by Logo was too weak, so he invented *object-oriented programming* as a way of enabling much more complex artifacts to be created in exploration of more complex domains. The command-line metaphors of Logo were too weak for the drawing, painting, and typeset-quality text that Kay felt was critical in order to enable rich media creation, so he and his group literally invented the desktop user interface as we know it today (Figure 3). Within this metaphor, Smalltalk provided a wide variety of programmer tools within the environment, including code browsers, object inspectors, and powerful debugging tools.



```
quit then 'dmt.boot.'
notes mdefs ev @erase.
usersetup.ft userpeditor.ft
usertextwindow.ft userfileseg.ft
window.ft format.ft diana.ft
blits.ft commander.ft tiles.ft
filin 'press.ft'. dsoff. ☞pf←pressfile
'jumbo.press'. pf bitmap
STDISPLAY rectangle. pf close.
dson.
sched ← figwindow 'coreuse'
sch
```

avoid full show for esc,
  cut, paste, (...)
auto-cleanup
get Larry's views
Clean windows
  para, file, font
  fig, pic, clock
  debugger, view
write up
  boolean exprs
  code objs
bootstrap
  read rtn, files
  new/init
  Glenn's mods
scheduling - interrupts

XEROX
PALO ALTO RESEARCH C
Learning Research Gro
October 29, 1976

To: **SMALLTALK Interest**

From: **Dan Ingalls** as scribe

Subject: **Message syntax**

Filed on: <INGALLS>STinterpreter.Bravo

This is a working paper describing the next SMALLTALK system. It is the result of many proposals and discussions among LRG and friends.

Interpretation
The operation of the bytecode interpreter is very simple. The code consists of sequences such as $\mathbf{d}_1$ $\mathbf{d}_2$ $\mathbf{d}_3$ $\mathbf{s}_1$, or three data bytes followed by a selector byte. Each of the data bytes in turn gets resolved to a full (16-bit)

Figure 3: Smalltalk-72, the first system with overlapping windows, icons, menus, and a mouse pointing device—all designed for the novice programmer.

Smalltalk's syntax is similar to Logo's, in that there are a small number of grammatical rules and very few reserved syntactic structures. Smalltalk developers felt that the few number of rules would make it simpler to learn. In Smalltalk, every statement is of the form `<receiverObject> <message>`.

- A statement like `4 printString` means "Send the integer 4 the message `printString` (which returns the text representation of the object)."

- Even control structures have this basic form. `(a=b) ifTrue: [Smalltalk beep]` means "Test if a is equal to b. Whatever boolean object is returned, send that object the message `ifTrue:`. If the boolean object is True, the *block* `Smalltalk beep` will be executed."

- Standard control structures like `while` and `for` follow the same consistent pattern. `1 to: 10 do: [:index | sum := sum + index]` adds the numbers 1 to 10 into the `sum`. `to:do:` is a message to the integer 1, which takes the arguments 10 and the code block.

While Smalltalk-72 was tested successfully with novice programmers, the later versions emphasized object-oriented programming for expert programmers and the desktop user interface for applications software–and de-emphasized the computational literacy ideas that Kay shared with diSessa. The notion of Smalltalk for novice programmers all but disappeared for perhaps 15 years.

The United Kingdom Open University adopted a form of Smalltalk for its introductory computing course. The Smalltalk used in the OU course is basically VisualWorks (a direct ancestor of Smalltalk-72), but with several enhancements.

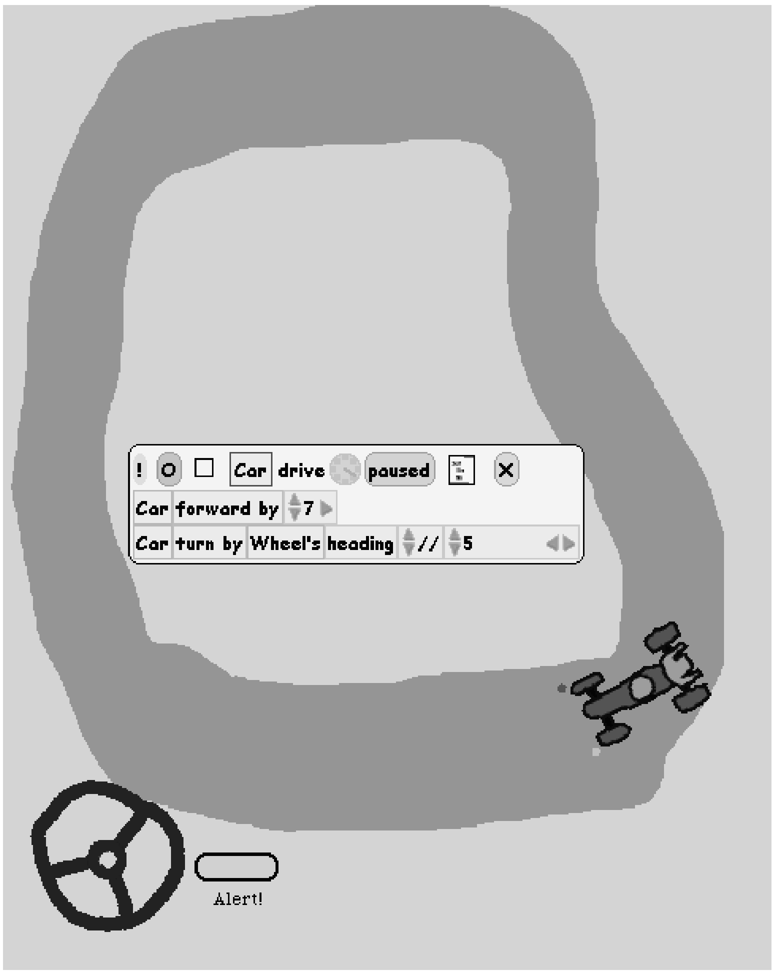
- The programming environment was carefully constructed so that few windows would be on the screen at once and only the portions of the environment relevant to the students' current project would be accessible.

- The domain of projects was mostly graphical with much of the programming projects centered in a three-dimensional graphical world where students could control objects and construct simulations.

One of the latest versions of Smalltalk, Squeak [15, 10], is being used again with students, especially younger children. *Squeak* is Smalltalk from the late 1970's minimally updated to run on modern machines, but then augmented with a wide range of new features, especially in support for multimedia. An alternative interface for using Squeak has been implemented *e-toys* that allows for a drag-and-drop tiling-based programming environment. Students literally drag variables, values, and methods from place-to-place to define procedures, mostly to control graphical objects (Figure 4)–and mostly with more complex syntax than in traditional Smalltalk. Like in Bruckman's MOOSE Crossing, the Squeak e-toys interface favors concreteness and ease of use to powerful abstractions. The e-toys interface has been used with success with 10–12 year old students [12].

3 Rule-Based Programming: Changing the Language and the Interface

Another set of answers to the question "What makes programming hard?" includes "The interface" (as diSessa said) but also "The kind of programming" (as Kay said).



```
Car drive paused
Car forward by 7
Car turn by Wheel's heading // 5
```

Figure 4: An example of a car-driving project in Squeak using the tiling-based

One direction of research in novice programming environments has developed non-textual programming environments oriented around *rule-based programming* rather than traditional imperative or procedural programming. Students using rule-based programming describe *states of the world* as opposed to telling the computer how to operate upon the world.

Prolog was a popular rule-based programming language, even with novices, even when it simply had a command-line kind of interface. In Prolog, one states facts about the world, e.g., "The factorial of n is 1 if $n = 1$, and otherwise, it's the factorial of $n - 1$." That isn't explicitly telling the computer *how* to get a factorial: It states a definition of factorial, which happens to be complete enough to be executable. That's how Prolog works. Prolog avoids some of the complexities of loops and conditionals with which research shows students have difficulty.

For example, Prolog can be taught *facts* by simply entering them, e.g.,

```
| parent(tom, bob).

| parent(bob, jim).

| parent(tom, liz).
```

The database can then be queried, using specific statements or parameterized *patterns*, as in:

```
| ?- parent(tom, bob).

yes

| ?- parent(tom, john).

no

| ?- parent(tom, X).

X = bob ?
```

Note that `X` in the above example is not a variable in the sense of most programming languages. `X` is not a name associated with some data. Instead, `X` is an *unbound* variable whose binding is found by search in the Prolog logic database.

We can state logical relations on top of these facts, such as the grandchildren of Tom being those Y's who are children of X where Tom is X's parent.

```
| ?- parent(tom, X), parent(X, Y).

X = bob
```

Prolog thus emphasizes the *logic* of programming, without confusing the matter with graphics or even procedures. Programmers in Prolog specify how things are *related* without specifying *what* is to be done–leaving those details to Prolog. For Prolog proponents, Prolog boils programming down to the basic activity of stating logical relations.

3.1 Extending Prolog into graphical domains

Some versions of Prolog could be used to generate graphics or other media, but the core of Prolog's descendants are entirely graphical. They have boiled the language down to a *matching rules* representation. The Prolog While some versions of Prolog could be used for graphical and other tasks not strictly textual, other languages

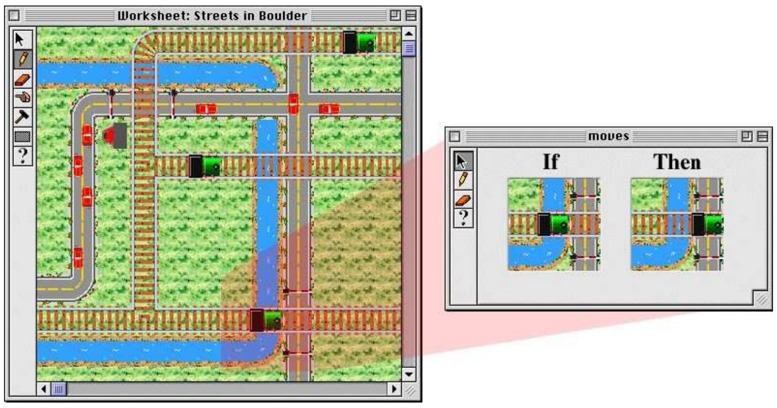


Figure 5: An AgentSheets simulation of a train

carried Prolog into a purely graphical language. These latter languages emphasize applications in a concrete, graphical domain appropriate to creating graphical simulations or videogames. It's a domain motivating for many students, and it lends itself toward providing immediate feedback to students' explorations. Just as we saw in the progression of Logo environments, and in the movement from Smalltalk to the Squeak e-toys interface, the progression of environments after Prolog is toward concreteness and graphical domains.

Stagecast Creator[2] and AgentSheets[3] both explicitly support rule-based programming *and* a different kind of interface for programming. The developers of Stagecast (formerly *KidSim* and *Cocoa*) explicitly aimed to use all that was learned about direct-manipulation interfaces to make the task of programming easier [28]. In both of these tools, the user defines rules that describe how the state of the world should change if particular conditions are met.

For example, consider the AgentSheets simulation of a train (in Figure 5). The rule appears on the right of Figure 5 (from [26]). The rule states that if the train is on the track on the left of the track, the train should move forward onto the right of the track.

Both AgentSheets and Stagecast Creator are most often used for building simulations or video games (Figure 6). The graphical nature of rules lend themselves to the kinds of motion and manipulations that many videogames provide. Both AgentSheets and Stagecast Creator support non-graphical rules, as well. For example, a more complex *if* condition can lead to a set of *then* actions, including sounds and setting variable values.

Both are used extensively in educational settings. Stagecast Creator has been used by kids to build a variety of videogames, including several for Internet competitions. AgentSheets has been used in some quite remarkable simulations for exploring social

[2]`http://www.stagecast.com`
[3]`http://www.agentsheets.com`

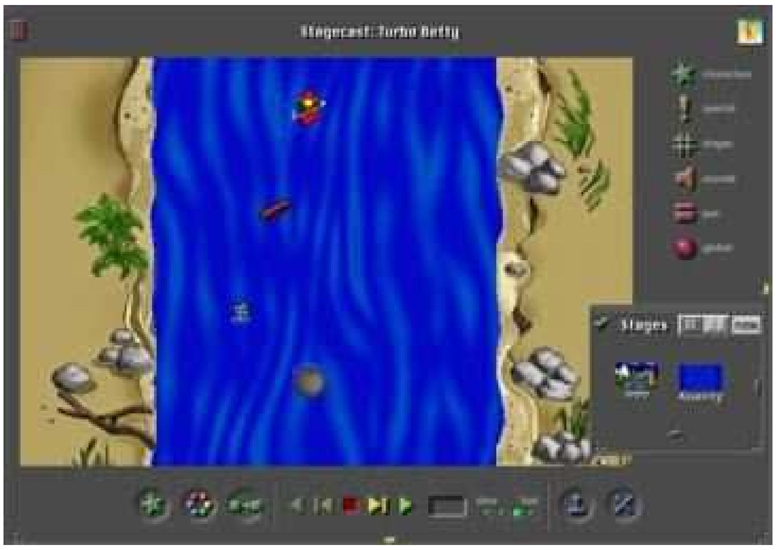


Figure 6: StageCast Simulation

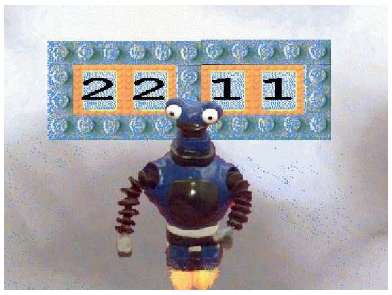


Figure 7: ToonTalk characters manipulate Lego-like data

studies, e.g., simulations of how peaceful protests can become riots, in a StarLogo-like fashion [25].

ToonTalk[4] by Ken Kahn is explicitly influenced by the work of Seymour Papert, but it follows the rule-based and non-text model of Stagecast Creator and AgentSheets. ToonTalk takes the model of programming-as-videogame much further than these other two environments. In ToonTalk, a student's program explicitly manipulates *characters* who, in turn, manipulate data and structures of data which appear as Lego bricks (Figure 7). The rendering of ToonTalk is exceptionally high-quality: The look-and-feel is as nice as a high-end videogame (Figure 8).

ToonTalk gives the same answers to "What makes programming hard?" as Stagecast Creator and AgentSheets, but it provides some additional ones.

- ToonTalk is concerned with making it obvious *who* is doing what a program

[4]`http://www.toontalk.com`

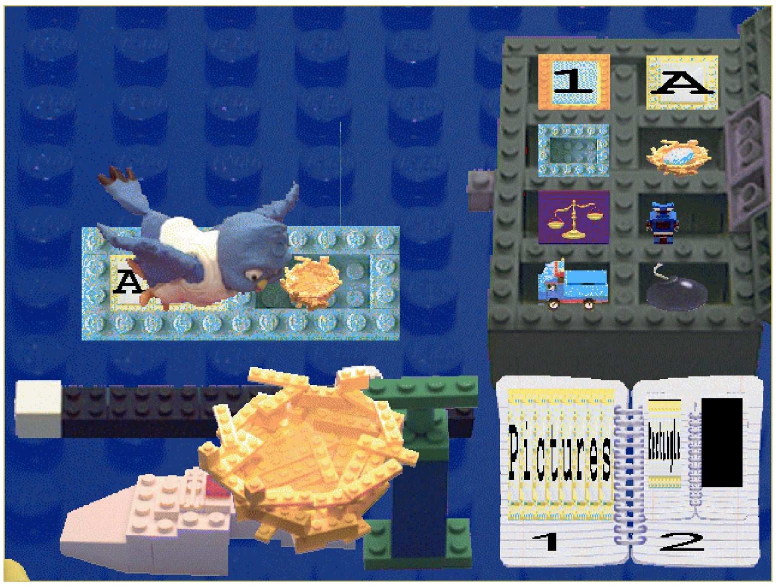


Figure 8: ToonTalk running has multiple characters and assembled data elements, all rendered in beautiful quality

commands. Agency is made visible through its characters.

- Like Boxer, ToonTalk includes naive realism in that everything is visible. What's more, ToonTalk provides the metaphor of Lego to make clear how virtual objects are, literally, *assembled.*

- ToonTalk takes great pains to make sure that its execution has the same realism as high-end videogames. For example, ToonTalk (like StarLogo) provides a high degree of concurrency—things happen at once, just as they do in the real world. Kahn believes that this makes it easier for students to understand and develop in ToonTalk.

4 Putting a New Face on an Old Language: Programming for Future Programmers

Still another answer to the question of "What makes programming hard?" is to say, "It's not the programming language at all." There's an argument that using idiosyncratic or *ad hoc* programming languages decreases student motivation, since the programming skills developed can't be used elsewhere. Perhaps the answer is "It's the programming environment—it needs to support learning the skills of expert programmers." This answer is probably most relevant to those students studying computer science as a potential profession, since they are clearly motivated to learn existing programming languages. Some researchers have argued that it's also relevant to students studying programming to learn problem-solving skills [30]. Environments that act upon this answer emphasize teaching design skills and *scaffolding* (providing additional support that students need but experts don't [3]) students to use traditional languages. In particular, the more complex syntax of traditional programming languages was viewed as a stumbling block for novice programmers, so much of the scaffolding was aimed at relieving syntax complexity.

Most of this work was done when Pascal was the dominant programming language in schools (Figure 9). These environments had in common support for *structured editing* and design support. Structured editing refers to how the text of the program is manipulated. Rather than simply typing the textual program, structured editors support specification of elements (e.g., from menus) and the completion of *placeholders* that fill in the details of the program.

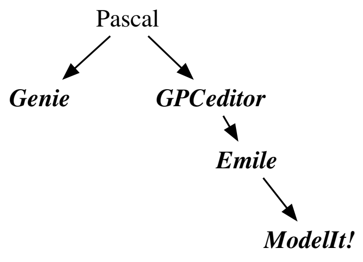
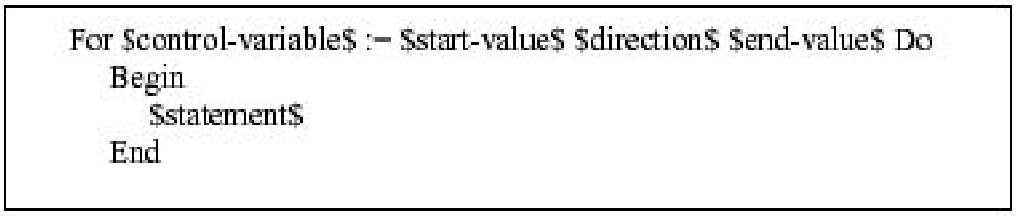


Figure 9: Family of novice programming environments based on supporting traditional programming languages

Probably the largest effort to create structured Pascal editors for students was the Genie effort at Carnegie-Mellon University [21]. Over some ten years, several different Genie editors were created. All of them provided structured editing support. For example, a user might choose a `for` loop to be inserted into her code. The loop would be inserted with placeholders identifying where additional pieces needed to be specified (Figure 10), which could be completed by selecting placeholders and making menu selections. Genie also provided visualizations in its debuggers so that diSessa's principle of naive realism was used to facilitate debugging (Figure 11).

The Genie developers also realized that part of students' problem with programming was in figuring out how to start and how to move forward to completion. Students lacked design skills [30]. Genie provided design views of programs that explicitly encouraged to see their programs as sets of components that they were assembling (Figure 12).

The GPCeditor was another Pascal-based structured editor, like Genie, but it started from the design view [11]. Rather then choose language elements, students using the GPCeditor specified their *goals*, and then selected *plans* from a *Plan Library*, which were instantiated as *code* (thus, Goal-Plan-Code editor). Figure 13 shows the



```
For $control-variable$ :– $start-value$ $direction$ $end-value$ Do
    Begin
        $statement$
    End
```

Figure 10: A Genie `for` loop with placeholders to-be-completed

students goal-plan decomposition in the upper left, their Plan Library in upper right, the actual code (with the selected code corresponding to the highlight goal-plan) in lower left, and the hierarchy of goal-plans in the lower right (overlapped with the execution window of a running program).

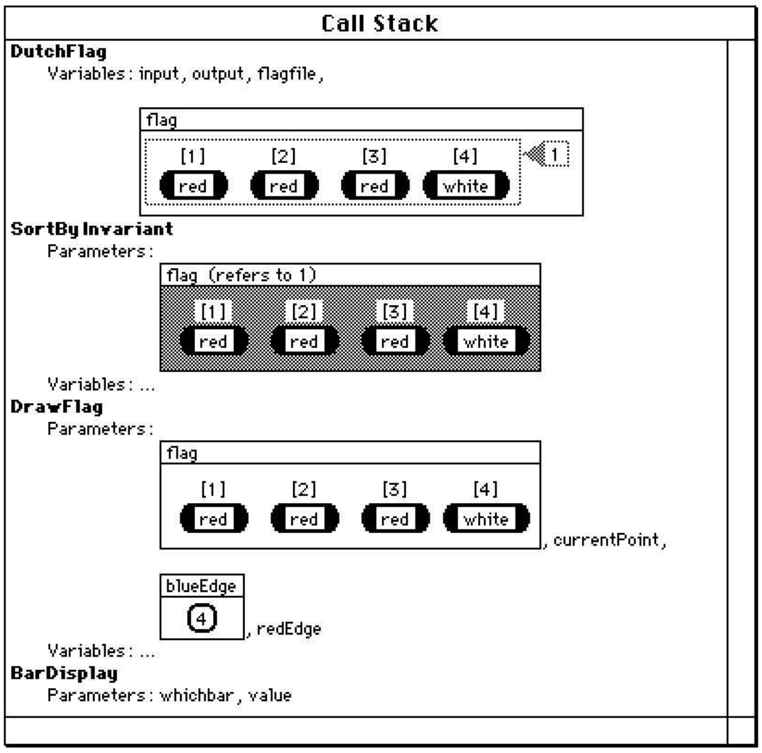
The GPCeditor was notable for its evaluation effort. It was used in secondary schools for several years, and findings suggest that students did develop design skills that transferred from GPCeditor to more traditional Pascal programming environments [11]. However, it's not clear that GPCeditor made the task any *easier*.

Successors to GPCeditor followed the same progression as Logo, in that they provided support for specific kinds of tasks that might themselves be motivating to students. *Emile* supported students building hypermedia (based on HyperCard) programs that simulated physical systems [9], using a goal-plan-code structure to ease the editing of the (fairly) traditional programming language. The study of Emile showed that students did learn a lot about physics through construction of their simulations, but less about programming. A still later successor, ModelIt [29], supported simulation of systems, but moved away from programming to a direct-manipulation system that supported specification of relationships between simulation factors as manipulation of graphical and textual statements–a similar progression as from Prolog to its successors.

5 Summary: Trends and Future Directions

In the over forty years of programming environments for novices that we skimmed over here, it is clear that the research community has only started to address the question of "What's hard about programming?" We can identify several trends in the research.

First, there is a clear trend toward a more traditional language syntax. The powerful abstractions of the earliest programming languages for novices have been de-emphasized in favor of programming languages that are easier to read, perhaps with support to make them easier to write. The syntax of Bruckman's MOOSE Crossing and of Squeak's e-toys are more complex than the Logo and Smalltalk ancestors–the parser has to work harder to figure out what's going on and the number of rules for



```
Call Stack
DutchFlag
    Variables: input, output, flagfile,
            flag
              [1]    [2]    [3]    [4]
              red    red    red    white     1
SortByInvariant
    Parameters:
            flag (refers to 1)
              [1]    [2]    [3]    [4]
              red    red    red    white
    Variables: ...
DrawFlag
    Parameters:
            flag
              [1]    [2]    [3]    [4]
              red    red    red    white   , currentPoint,
            blueEdge
              4    , redEdge
    Variables: ...
BarDisplay
    Parameters: whichbar, value
```

Figure 11: Genie's editor with visualizations of data elements

interpreting what's going on grows. Why should a more complex language be easier for students to understand? It's not obvious, but we can make some conjectures.

- Novice programmers may not actually think about the process of interpretation or compilation. To think about *how* the language is understood by the computer is a level of abstraction that is beyond the novice.

- Natural language *is* complex and ambiguous. As the programming languages reflect natural complexity, they may become easier for the novice to understand, even though the mathematical and logical clarity decreases.

- Finally, it's not clear how much computer science influences novice programmers, especially when the students are in secondary or post-secondary education. In one of our recent introductory courses at Georgia Tech, 83% of the student respondents to a survey claimed that they had no previous programming experience. But when interviewed, most professed to using Logo or Basic in elementary school, or looking through books or magazines about programming at some point. Computer science is part of modern culture, and some of the predominant programming cultures have some influence on students.

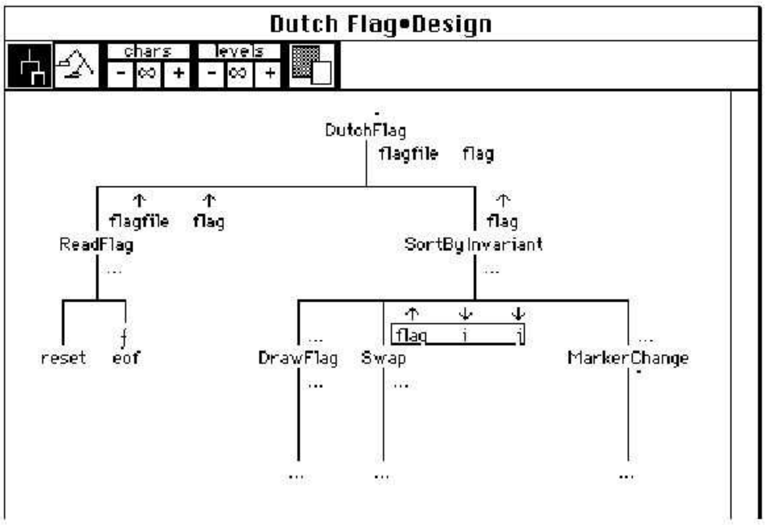


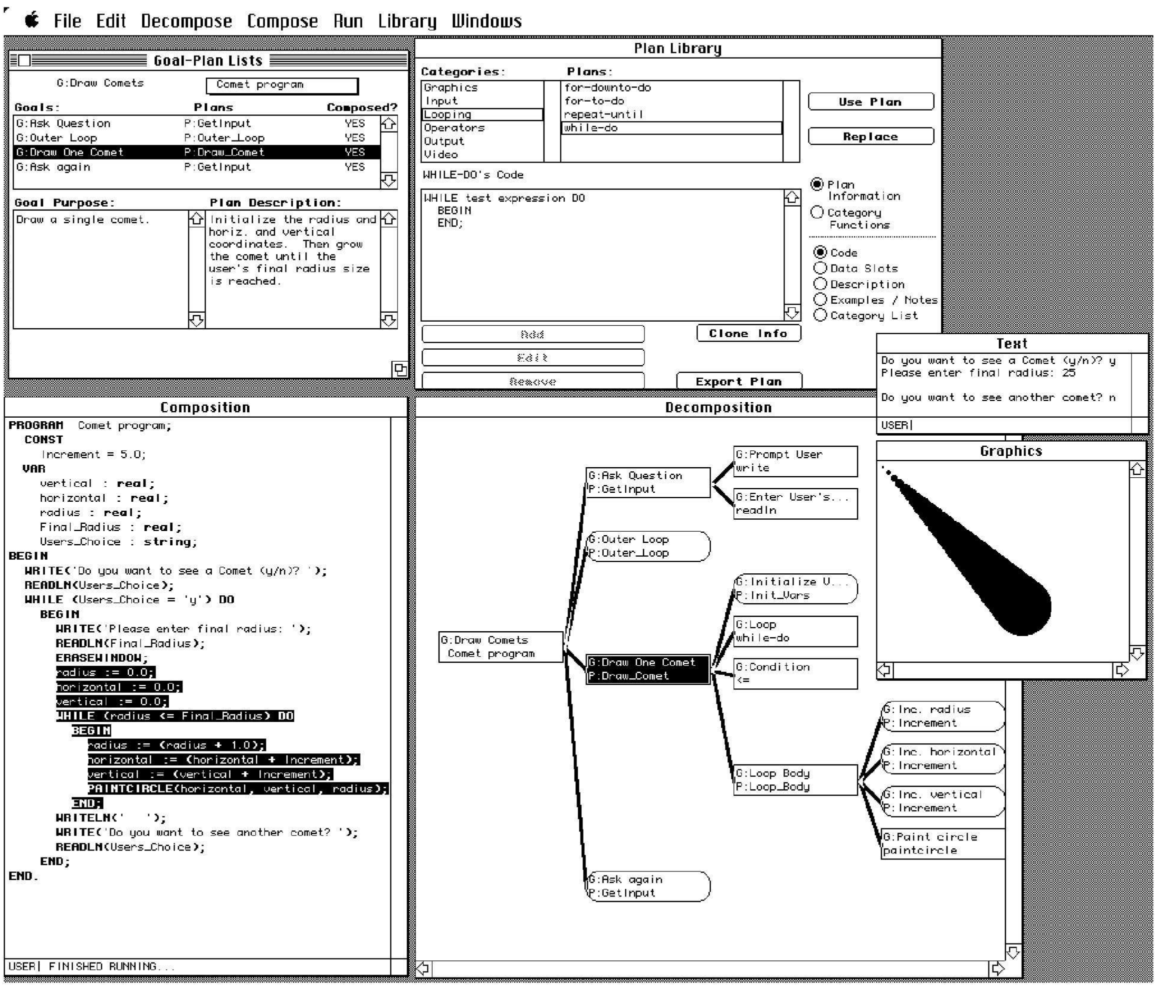
Figure 12: Genie's design view

Second, there is another trend toward relevance and applicability. Students want to work on computational artifacts that have meaning for them, e.g., that are interesting and relevant. Research studying how students relate to computer science and why they don't stay with it has shown that a *lack* of meaning and relevance are key issues for students' distaste for computer science (e.g., [19]). The turtle graphics of Logo and logic databases of Prolog have faded in favor of building video games, text and graphical virtual realities, and simulations.

Third, related to the trend toward relevance and applicability is a trend toward environments and tasks that give students immediate feedback on their work. Students want to "pet Rover" and see their car turn and move their robot. The issue is more than just being motivating to pet the dog and see the graphical reaction. If the students' programs work, they see immediate and understandable responses: Rover wags his tail, the car turns, and the robot moves. Contrast this with a program to sort a list of names or compute the n-th digit of π and then printing the result. The student has to analyze the result and decide if it really is sorted or really does reflect the correct digit of π. When students are dealing with the complexity of learning programming, they don't want or need to deal with subtle shades of correctness–they want it to be right or wrong, so that they can correct it and move on.

There are a great many issues that have scarcely been addressed in novice programming environments, and many more that arise due to new technologies. Moore's Law constantly changes the technological scenery, thus changing what's possible for novice programming and what novices can do with their new programming skills.

- For example, we know that today's children use processors all the time, but not on the desktop—in their handheld videogames, cellphones, and Palm Pilots. Is



G:Draw Comets   Comet program

Goals:	Plans	Composed?
G:Ask Question	P:GetInput	YES
G:Outer Loop	P:Outer_Loop	YES
G:Draw One Comet	P:Draw_Comet	YES
G:Ask again	P:GetInput	YES

Goal Purpose: Draw a single comet.

Plan Description: Initialize the radius and horiz. and vertical coordinates. Then grow the comet until the user's final radius size is reached.

Categories:	Plans:
Graphics	for-downto-do
Input	for-to-do
Looping	repeat-until
Operators	while-do
Output	
Video	

WHILE-DO's Code

```
WHILE test expression DO
  BEGIN
  END;
```

(● Plan Information) ○ Category Functions

(● Code) ○ Data Slots ○ Description ○ Examples / Notes ○ Category List

```
Do you want to see a Comet (y/n)? y
Please enter final radius: 25

Do you want to see another comet? n
USER|
```

```
PROGRAM  Comet program;
  CONST
    Increment = 5.0;
  VAR
    vertical : real;
    horizontal : real;
    radius : real;
    Final_Radius : real;
    Users_Choice : string;
BEGIN
  WRITE('Do you want to see a Comet (y/n)? ');
  READLN(Users_Choice);
  WHILE (Users_Choice = 'y') DO
    BEGIN
      WRITE('Please enter final radius: ');
      READLN(Final_Radius);
      ERASEWINDOW;
      radius := 0.0;
      horizontal := 0.0;
      vertical := 0.0;
      WHILE (radius <= Final_Radius) DO
        BEGIN
          radius := (radius + 1.0);
          horizontal := (horizontal + Increment);
          vertical := (vertical + Increment);
          PAINTCIRCLE(horizontal, vertical, radius);
        END;
      WRITELN('   ');
      WRITE('Do you want to see another comet? ');
      READLN(Users_Choice);
    END;
END.
USER| FINISHED RUNNING...
```

Figure 13: GPCeditor: Goal-Plan-Code editor

programming too locked to the desktop? What does programming mean on these smaller devices that might be more motivating (relevant and applicable) for students to manipulate?

- In some new work that we're undertaking at Georgia Tech, we have students learning to program by manipulating media, such as changing pixel color values to generate grayscale or by moving sound samples to modify sounds. Students start our writing relatively short programs that loop over the pixels or samples–but the loops loop *many* times. Four seconds of sound at low-quality is some 88,000 samples. If something goes wrong in the students' programs, printing variable values doesn't make sense. Nobody wants to wade through 88,000 lines of output. Traditional debuggers and breakpoints aren't useful, either–most debuggers are too confusing for novices, and nobody wants to hit the button *Continue from Breakpoint* 88,000 times. How do we help students to understand their programs in this kind of domain?

- The interaction between the programming environment and language has just started to be addressed. Computer speed is such that the parser can be running all the time, while editing, so that immediate feedback is offered during program entry. How do we use this computational power to scaffold the novice programmer?

Finally, it may be that "What makes programming hard?" is not the most fruitful question to ask. Perhaps we don't know yet what programming really is or what it could be (as diSessa suggests in his book [7]). Perhaps we don't know yet what students would really want to learn programming for.

Designing programming environments for novices is a fascinating field that we have only just started to explore. There is a great many more questions and answers to explore, and some wonderful environments yet to build and try in that exploration. The progress in the field is toward making programming more interesting, more relevant, and more powerful for students. The research opportunities could hardly be broader, and promise enormous potential impact.

References

[1] Harold Abelson and Andrea A. diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press, Cambridge, MA, 1986.

[2] Beth Adelson and Elliot Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360, 1985.

[3] Phyllis C. Blumenfeld, Elliot Soloway, Ronald W. Marx, Joseph S. Krajcik, Mark Guzdial, and Annemari Palincsar. Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, 26(3 & 4):369–398, 1991.

[4] Amy Bruckman. Situated support for learning: Storm's weekend with rachael. *Journal of the Learning Sciences*, 9(3):329–372, 2000.

[5] Amy Bruckman, Carlos Jensen, and Austina DeBonte. Gender and programming achievement in a cscl environment. In Gerry Stahl, editor, *Proceedings of the 2002 Computer Supported Collaborative Learning conference*, pages In–Press. University of Colorado at Boulder, Boulder, CO, 2001.

[6] Amy S. Bruckman. *MOOSE Crossing: Construction, Community and Learning in a Networked Virtual World for Kids.* PhD thesis, MIT, 1997.

[7] Andrea diSessa. *Changing Minds.* MIT Press, Cambridge, MA, 2001.

[8] T. R. G. Green. Conditional program statements and comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50:93–109, 1977.

[9] Mark Guzdial. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments*, 4(1):1–44, 1995.

[10] Mark Guzdial. *Squeak: Object-oriented design with Multimedia Applications.* Prentice-Hall, Englewood, NJ, 2001.

[11] Mark Guzdial, Michael Konneman, Christopher Walton, Luke Hohmann, and Elliot Soloway. Layering scaffolding and cad on an integrated workbench: An effective design approach for project-based learning support. *Interactive Learning Environments*, 6(1/2):143–179, 1998.

[12] Mark Guzdial and Kim Rose. Squeak, open personal computing for multimedia. 2001.

[13] Idit Harel and Seymour Papert. Software design as a learning environment. *Interactive Learning Environments*, 1(1):1–32, 1990.

[14] C. Hoyles and R. Noss. *Learning Logo and Mathematics.* MIT Press, Cambridge, MA, 1992.

[15] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA'97 Conference Proceedings*, pages 318–326. ACM, Atlanta, GA, 1997.

[16] R. Jefferies, A. A. Turner, P. G. Polson, and M. E. Atwood. *The processes involved in designing software.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[17] Alan Kay and Adele Goldberg. Personal dynamic media. *IEEE Computer*, pages 31–41, 1977.

[18] D.M. Kurland, C.A. Clement, R. Mawby, and R.D. Pea. Mapping the cognitive demands of learning to program. In R.D. Pea and K. Sheingold, editors, *Mirrors of Minds*, pages 103–127. Ablex, Norwood, NJ, 1986.

[19] Jane Margolis and Allan Fisher. *Unlocking the Clubhouse: Women in Computing.* MIT Press, Cambridge, MA, 2002.

[20] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*, 33(4):125–140, 2001.

[21] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. Evolution of novice programming environments: The structure editors of carnegie-mellon university. *Interactive Learning Environments*, 4(2):140–158, 1994.

[22] David B. Palumbo. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[23] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, New York, NY, 1980.

[24] R.D. Pea and D.M. Kurland. On the cognitive effects of learning computer programming. In R.D. Pea and K. Sheingold, editors, *Mirrors of Minds*, pages 147–177. Ablex Publishing, Norwood, NJ, 1986.

[25] Alex Repenning, A. Ioannidou, and J. Phillips. Collaborative use and design of interactive simulations. *Proceeings of Computer Supported Collaborative Learning Conference at Stanford (CSCL'99)*, 1999.

[26] Alex Repenning, A. Ioannidou, and J. Zola. Agentsheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), 2000.

[27] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* MIT Press, Cambridge, MA, 1997.

[28] David Canfield Smith, Allen Cypher, and Jim Spohrer. Kidsim: Programming agents without a programming language. *Communications of the ACM*, 37(7):55–67, 1994.

[29] E. Soloway, S.L. Jackson, J. Klein, C. Quintana, J. Reed, J. Spitulnik, S.J. Stratford, S. Studer, J. Eng, and N. Scala. Learning theory in practice: Case studies of learner-centered design. In M.J. Trauber, editor, *CHI96 Conference Proceedings*, pages 189–196. Vancouver, British Columbia, Canada, 1996.

[30] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[31] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.

[32] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and J. Greenspan. What do novices know about programming? In Andre Badre and Ben Schneiderman, editors, *Directions in Human-Computer Interaction*, pages 87–122. Ablex Publishing, Norwood, NJ, 1982.

[33] James C. Spohrer and Elliot Soloway. Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume March. 1985. Tucson, AZ.

[34] Sherry Turkle and Seymour Papert. *Epistemological pluralism and the revaluation of the concrete*, pages 161–192. Ablex, Norwood, NJ, 1991.