DOCUMENT RESUME

ED 392 435 IR 017 726

AUTHOR Kahn, Ken
TITLE ToonTalk(TM)--An Animated Programming Environment for Children.
PUB DATE 95
NOTE 9p.; In: "Emerging Technologies, Lifelong Learning, NECC '95"; see IR 017 705.
PUB TYPE Reports - Descriptive (141) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS Authoring Aids (Programming); Autoinstructional Aids; *Children; *Computer Games; *Computer Graphics; Computer Interfaces; Computer Simulation; *Computer System Design; Independent Study; *Programming; *Video Games
IDENTIFIERS *Computer Animation

ABSTRACT

This paper describes ToonTalk, a general-purpose concurrent programming system in which the source code is animated and the programming environment is a video game. The design objectives of ToonTalk were to create a self-teaching programming system for children that was also a very powerful and flexible programming tool. A keyboard can be used for various accelerators, but a ToonTalk user can get by with just a game pad, joystick, or mouse. Every abstract computational aspect is mapped into a concrete metaphor. The ToonTalk "world" resembles a 20th century city; an entire ToonTalk computation is a city. The programmer controls a "programmer persona" or robot in this video world to construct, run, debug and modify programs. In addition to a message-passing interface, ToonTalk provides a direct control of sprites (animated graphical elements); a sprite can be flipped over to reveal a notebook which contains remote controls for that sprite. Initial testing of ToonTalk use by children has revealed that it provides an entertaining way of onstructing programs. (Contains 10 references.) (AEF)

ToonTalk™ - An Animated Programming Environment for Children

by Ken Kahn

Paper presented at the NECC '95, the Annual National Educational Computing Conference (16th, Baltimore, MD, June 17-19, 1995.

paper

ToonTalk™ — An Animated Programming Environment for Children

Ken Kahn
Animated Programs
44 El Rey Road
Portola Valley, CA 94028
415-851-0890 voice; 415-851-4816 fax
Email: Kahn@CSLI.Stanford.edu

Keywords: computer programming, children, video games, concurrent programming

Abstract

Seymour Papert once described the design of the Logo programming language as taking the best ideas in computer science about programming language design and "child engineering" them (Papert 1977). Twenty-five years after Logo's birth, there has been tremendous progress in programming language research and in computer-human interfaces. Programming languages exist now that are very expressive and mathematically very elegant and yet are difficult to learn and master. We believe the time is now ripe to attempt to repeat the success of the designers of Logo by child engineering one of these modern languages.

When Logo was first built, a critical aspect was taking the computational constructs of the Lisp programming language and designing a child friendly syntax for them. Lisp's "CAR" was replaced by "FIRST", "DEFUN" by "TO", parentheses were eliminated, and so on. Today there are totally visual languages in which programs exist as pictures and not as text. We believe this is a step in the right direction, but even better than visual programs are animated programs. Animation is much better suited for dealing with the dynamics of computer programs than static icons or diagrams. While there has been substantial progress in graphical user interfaces in the last twenty-five years, we chose to look not primarily at the desktop metaphor for ideas but instead at video games. Video games are typically more direct, more concrete, and easier to learn than other software. And more fun too.

We have constructed a general-purpose concurrent programming system, ToonTalk, in which the source code is animated and the programming environment is a video game. Every abstract computational aspect is mapped into a concrete metaphor. For example, a computation is a city, an active object or agent is a house, birds carry messages between houses, a method or clause is a robot trained by the user and so on. The programmer controls a "programmer persona" in this video world to construct, run, debug and modify programs. We believe that ToonTalk is especially well suited for giving children the opportunity to build real programs in a manner that is easy to learn and fun to do.

Goal Number 1: *A Self-teaching* Programming System for Kids

Programming can be a fun and empowering activity, but it is accessible only to those who manage to surmount a large initial hurdle. This hurdle includes learning a formal programming language and computational concepts such as variables, procedures, and flow of control. If this hurdle could be minimized and overcoming what remains can be made fun, then children and curious adults would be able to creatively mold computers into whatever they want. Learning to use computers without learning to program, is like learning to read without learning how to write.

Our goal is to create a computer system which children can use to build a very wide variety of programs without being taught how to use it. Many believe that any system that is easy enough to learn to use without the help of a teacher will have to be very limited. ToonTalk, however, is a self-teaching system that is flexible and expressive. A wide range of programs can be constructed ranging from games like Pong, Hangman and PacMan to programs for controlling motors and sensors of Lego and other construction toys to conventional programming examples like factorial and parallel quick sort.

There is precedent for powerful, yet self-teaching, systems outside of computer programming. Children, for example, learn on their own how to build complex Lego constructions. They master video games that require exploration and problem solving in complex fictional worlds. Analysis of video games and Lego systems has provided many of the ideas that make ToonTalk easy to learn (Malone 1980; Provenzo 1991).

Some of the design principles derived from good construction toys and video games include:

1. Make the initial experience simple and gradually increase complexity.
2. Encourage exploration and curiosity.
3. Provide and maintain appealing fantasies.
4. Continually challenge without frustrating.
5. Frequent use of animation and film techniques and principles (video games only).

In constructing the ToonTalk programming system, we strove to follow these principles. We also borrowed heavily from the *technology* of video games. For example, we copy the way that video games frequently put the player into the game world by providing a persona or avatar in that world that the player controls and identifies with. Children react to events as if they were, for example, one of the Mario brothers when they play the Mario Brothers games. In ToonTalk, the programmer is an animated character building, testing and debugging programs.

The Logo community is also interested in giving children the ability to program for epistemological reasons (Papert 1993). They argue that programming is a rich soil for learning fundamental thinking and problem-solving skills. Children learn about representation, problem decomposition, abstraction, debugging and so on. This can happen while learning programming and it is very important, but unfortunately it is not the typical result of learning Logo (Yoder 1994). While we hope that this kind of learning will be more frequent with ToonTalk, we will be satisfied if the outcome is simply to empower children to creatively master computers.

Goal number 2: A *Powerful* Programming System for Kids

ToonTalk was built to be both very easy to learn and to be a very powerful and flexible programming tool. These are usually considered conflicting goals that require compromises. A language like C++ is very flexible and powerful but it is also very complex and difficult to learn. HyperTalk, in contrast, has been learned and used by many non-programmers but it has many inherent limitations. Kids can pretend to help in the garden with toy shovels and rakes, but if they really want to do gardening they should have real tools that have been adapted to their special requirements.

Theory based programming language design has produced many languages that are small yet powerful. Most functional and logic programming languages are examples as are some object-oriented programming languages. The problem is that languages like Scheme, ML, Prolog, Flat Guarded Horn Clauses (FGHC) and SmallTalk80, while small and elegant, are difficult for even computer science students to learn. It would seem absurd to expect second graders to master a programming language that professional programmers find very difficult.

But maybe the difficulty is not in the concepts *per se* but their lack of accessible metaphors. Consider as an example the concept of communication channels found in many concurrent systems. Associated with these channels are read and write privileges. Some support the notion that an attempt to read from an empty channel will suspend until something is written. These concepts are usually taught in an advanced computer science course. But these difficult abstract concepts can be replaced by exactly equivalent everyday concrete analogs. In ToonTalk birds and nests are the communication channels. The ability to write on a channel is a bird who behaves like a carrier pigeon. Read access to that channel is the nest of that bird. Birds can be copied and each copy has the same nest (i.e., each bird copy is a write capability on the same channel). The behavior of birds implements the operational semantics of channels. When a bird is given a message it flies to its nest, leaves the message on the nest, and flies back to where it was. If there already are things on its nest it puts the new item under these items (this implements a first-in first-out semantics for the queued messages). If another bird is busy putting something there it waits for its turn (this provides arbitration between multiple writers on the same channel). A bird finds its nest even if it has been moved (so a read capability will continue to work even if it has been transferred). These rules of behavior for birds are not very hard for a seven-year old to understand.

ToonTalk is based upon a concurrent constraint language (Saraswat 1993) similar to Janus (Saraswat, Kahn and Levy 1990). We chose concurrent constraint programming as the underlying foundation of ToonTalk for many reasons. One reason is that over ten years of use at many research centers has demonstrated that there is no risk that the language will be inadequate for building a wide variety of large programs (Shapiro 1989). The languages are small yet very powerful. This has lead to a much simpler design for ToonTalk than had it been based upon conventional languages.

Another reason for the choice is that these languages are inherently concurrent. Many find it surprising that a concurrent language would be better for children than a sequential language. They see sequential programming as hard enough without having to consider multiple interacting sequential programs. However, sequential languages extended to be concurrent are very complex but languages designed from scratch to be concurrent can be very elegant.

Programs typically model the world and the world is concurrent. Sequential programming languages provide a world in which only one thing can happen at a time. This is a very strange world. Children when first exposed to programming, especially object-oriented programming, expect it to be concurrent. Most of the programs that children want to write are naturally concurrent. A Pong game, for example, consists of at least a paddle, a ball and a score keeper. Good object-oriented design indicates that each should be handled by an independent computational component. Kids (and most non-programmers when introduced to an object-oriented programming system) expect that each object is running all the time. How weird to have to switch attention between controlling the paddle, ball and score keeper instead of just letting each one do its thing. But anarchy results unless components can communicate and synchronize. Making conventional languages concurrent with communication and synchronization facilities leads to a complex mess that gives concurrent programming its reputation for being very hard. A new concurrent programming language with a good semantically motivated design can avoid this mess.

Resnick attempted to introduce concurrency to children by extending the Logo programming language to support multiple concurrent threads (Resnick 1988). The language was too complex, but the research confirmed the appropriateness of concurrent programming languages for children. Resnick also built a parallel form of Logo called Star Logo which had the simplifying but severe limitation that only multiple instances of the same program can run in parallel. More recently LCSI's MicroWorlds Logo introduced a very simple form of parallelism to Logo. While sometimes useful it is very limited in its

ability to describe communication and synchronization between these parallel activities.

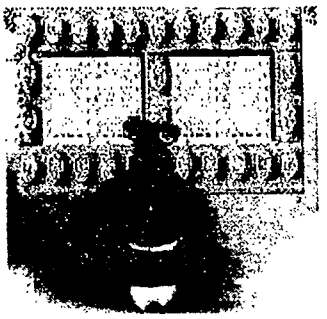
Animated Source Code

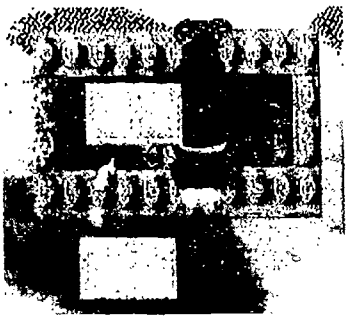
The fundamental idea behind ToonTalk is that source code is animated. (ToonTalk is so named because one is "talking" in (car)toons.) This does not mean that we take a visual programming language and replace some static icons by animated icons. It means that animation is the means of communicating to both humans and computers the entire meaning of a program. Given the dynamic nature of computation animation is especially well-suited for this.

Even small children have no troubles producing a range of sophisticated animations when playing games like Mario Brothers. While the range is, of course, very limited relative to a general animation authoring tool, video game style animation is fine for the purposes of communicating programs to computers. If, for example, a program fragment needs to swap the values of two locations, what can be more natural and easy than grasping the contents of one, setting it down, grasping the contents of the other, placing it at the first location and then moving the original item to the second location? (See figure 1.) This is something a very young child can understand and do while only a programmer can write the following equivalent code:

```
temp := x;
x := y;
y := temp;
```

Once the step is taken to use video game technology for the construction of source code, it is easy to see other uses of video game technology for browsing, editing, executing and debugging programs. Other ideas from video games can be borrowed. Some video games have animated characters whose purpose is to provide help to users. These characters can play the role of on-line help and tutorial systems.





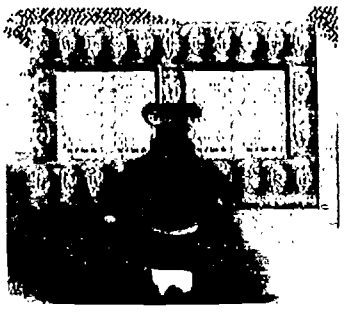


Figure 1 — Snapshots from swapping two elements. (color original)

ToonTalk — The Language and Metaphor

Video games, especially adventure and role-playing ones, place the user in an artificial universe. The laws of such universes are designed to meet constraints of game play, learnability, and entertainment. While playing these games the user learns whether gravity exists, if doors need keys to open, if one's health can be restored by obtaining and consuming herbs, and so on. What if the laws of the game universe were designed to be capable of general purpose computing, in addition to meeting the constraints of good gaming?

It seems that no one has ever tried to do this. (And when we figured out how to, we applied for a patent on the invention.) Rocky's Boots and Robot Odyssey were two games from The Learning Company in the early 1980s that excited many computer scientists. In these games, one can build arbitrary logic circuits and use them to program robots. This is all done in the context of a video game. The user persona in the game can explore a city with robot helpers. Frequently in order to proceed the user must build (in an interactive animated fashion) a logic circuit for the robots to solve the current problem. ToonTalk is pushing the ideas behind Robot Odyssey to an extreme, capable of supporting arbitrary user computations (not just the Boolean computations of Robot Odyssey).

Computer scientists strive to find good abstractions for computation. Here, in addition, we are striving to find good "concretizations" of those abstractions. The challenges are twofold: to provide high-level powerful constructs for expressing programs and to provide concrete, intuitive, easy-to-learn, systematic game analogs to every construct provided.

The ToonTalk world resembles a twentieth century city. There are helicopters, trucks, houses, streets, bike pumps, toolboxes, dust busters, boxes, and robots. Wildlife is limited to birds and their nests. This is just one of many consistent themes that could underlie a programming system like ToonTalk. A space theme with shuttle craft, teleporters and like would work as well. So would a medieval magical theme or an Alice in Wonderland theme.

An entire ToonTalk computation is a city. Most of the action in ToonTalk takes places in houses. Communication between houses (and to built-in capabilities) is accomplished by birds (kind of like homing pigeons). Birds accept things, fly to their nest, leave them there, and fly back Typically houses contain robots that have been trained to accomplish some small task. Robots have thought bubbles that contain pictures of what the local state should be like before they perform their task. Local state is held in cubby holes (i.e. boxes). Cubbies also are used for messages and compound data (i.e., tuples). If a robot is given a cubby containing everything that is in its thought bubble, it will proceed and repeat the actions it was taught. Abstraction arises because the picture in the thought bubble can leave things out and it will still match. A robot corresponds roughly to a method in an object-oriented language or a conditional. A line of robots provides something like an "if then else" capability. Animated scales can be placed in a compartment of a box. The scale will tip down on the side whose neighboring compartment is greater than (or if text, alphabetically after) the compartment on the other side. By placing scales tipped one way or another the conditionals can include less than, equal or greater than tests.

The behavior of a robot is exactly what it was trained to do by the programmer. This training corresponds in traditional terms to defining the body of a method or clause. The actions possible are:

- sending a message by giving a box or pad to a bird,
- spawning a new agent by dropping a box and a team of robots into a truck,
- performing simple primitive operations such as addition or multiplication,
- copying an item by using a magician's wand,
- terminating an agent by setting off a bomb,
- changing the contents of the compartments of a box.

When the user controls the robot to perform the actions she is acting upon concrete values. This has much in common with keyboard macro programming and programming by example (Smith 1975). The hard problem for programming by example systems is how to abstract the example to introduce variables for generality. ToonTalk does no induction or learning. Instead the user explicitly abstracts a program fragment by removing detail from the thought bubble. The preconditions are thus relaxed. The actions in the body are general since they have been recorded with respect to which compartment of the box was acted upon, not what items happened to occupy the box.

If a user never turned off their computer nor wanted to share a program with another then this world with houses, robots, etc., would be adequate. To provide permanent storage we have introduced notebooks into ToonTalk. A programmer can use notebooks to store anything they've built. Notebooks can contain notebooks to provide a hierarchical storage system. Notebooks provide an interface to the essential functionality of the file system without leaving the ToonTalk metaphor. The initial notebook contains sample programs and access to facilities like animation and sounds.

Beyond the Programming Language

The Logo programming language isn't just a child-engineered version of Lisp, but also includes turtle graphics. While Logo is sometimes used to perform numerical or textual computations, its primary appeal has been in its turtle graphics package. While turtle graphics is still appealing and many modern Logo implementations have extended the idea to have multiple turtles with different appearances, it does not have the same appeal as game programming has with children. While game programming is possible using turtle graphics, it is difficult.

ToonTalk does not currently support turtle graphics — instead, effort was made to provide support for game programming. The lowest level of support is a message-passing interface to a *sprite* library. Sprites are animated graphical elements that are composed to make a game. Mario is a sprite, as is a mushroom he might eat, and so on. A sprite's appearance is selected from a set of animation loops. A sprite's size and location changes can be animated as well. A mechanism is provided for detecting and acting upon collisions between sprites. A ToonTalk message-passing interface to such functionality means that one can obtain a bird for a sprite and give that bird messages that mean things like "move up 10 units", or "change size to 20", "set speed to 30" or "are you colliding with anyone?". This interface is very general and powerful but it turned out to be too awkward and clumsy for doing simple things.

In addition to the message passing interface ToonTalk provides a direct control of sprites. A sprite can be flipped over. Initially on the back side is just a notebook which contains remote controls for that sprite. For example, its width control can be obtained from the notebook. As the width of the sprite is changed the number in the control changes as well. Also the user can change the value of the number and the sprite's width automatically changes accordingly. There are currently remote controls for position, speed, size, collision detection, and appearance selection.

For example, one can train a robot to repeatedly increment a number and put that robot to work on the speed of the sprite and the sprite will accelerate. One can place this robot and remote control on the back side of a picture and then flip the sprite back over. If this sprite is copied or saved and later retrieved from a notebook, this acceleration behavior will still be on the flip side of the sprite and active. This enables one to build a nice library of useful behaviors for sprites like bouncing off of walls or tracking the mouse. Sprites can be composed simply by placing one on top of another. (The hand vacuum is necessary for separating them later.) Behaviors can be copied and combined directly.

In addition to turtle graphics, many Logo implementations include libraries for controlling motors and sensors in a Lego, Fisher Technic or Capsula construction set (Papert 1993). This enables children to build toys with behaviors. Children have made things ranging from robots, to cars that follow lines drawn on the floor, to household machines like toy washing machines that stop when the door is opened, to traffic lights that change color and respond to a pedestrian button. As with LegoLogo, ToonTalk provides an interface for turning on and off motors and lights and reading sensors. Currently, only a message-passing interface exists but another interface based upon remote controls is planned. While our experience with this is limited, it does seem that the underlying concurrency of ToonTalk enables much more modular control than Logo does. A ToonTalk house can be built with robots for controlling a traffic light, another for a car which stops at red lights, and so on. In contrast, in Logo a sequential program must alternate its attention between different elements.

Building a Pong-like Game in ToonTalk

In an attempt to provide a more detailed understanding of ToonTalk, the full-length version of this paper describes how a child builds a Pong-like game from scratch. ToonTalk is started and the user finds herself (or himself) flying in a helicopter over a city. Houses can be seen below. She lands in front of a house and enters the front door. Following her everywhere is a toolbox character. Inside, there is a friendly Martian ready to offer tours or coaching, but she ignores him since she's used the system a bit and feels confident she can build a simple Pong-like game on her own. (See Figure 2.) She sits down on the floor and her toolbox scurries in front of her and opens. Four characters emerge. A notebook flies out. A hand-held vacuum with legs runs out. A bike pump hops away. And a magic wand floats out. Remaining in the toolbox are eight kinds of things that the programmer will use to construct her game: number pads, text pads, cubby holes, bird nests, scales, robots, construction trucks and bombs. (See Figure 3.)

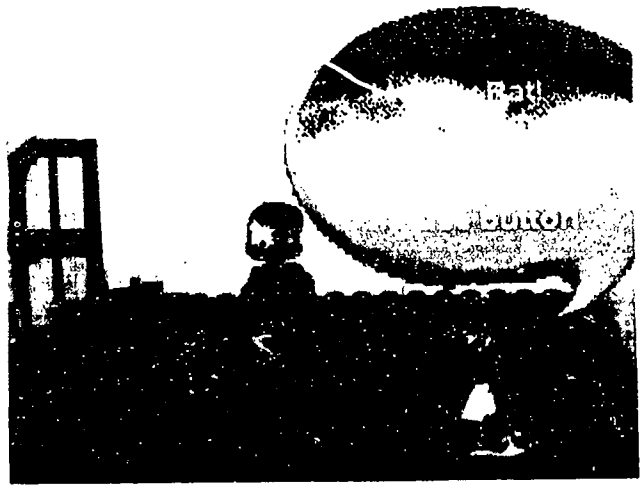


Figure 2 — Programmer and toolbox inside

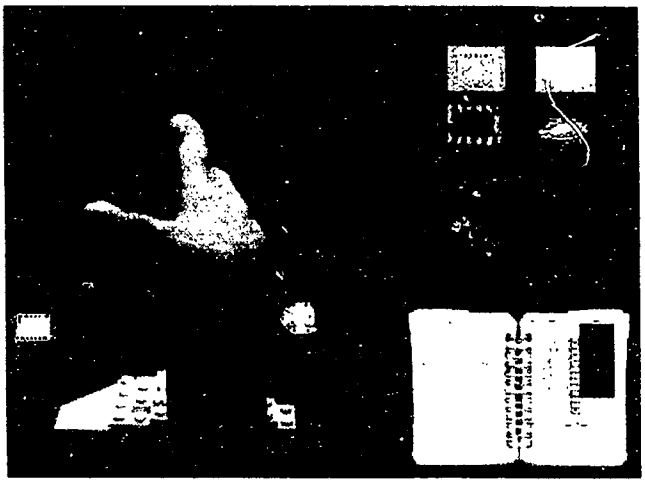


Figure 3 — Floor view after sitting

Briefly, she begins by building a paddle that follows the vertical movements of the mouse. She trains a robot to copy the mouse's vertical speed over to the paddle's vertical speed. She puts the robot on the back side of the paddle and gives it a cubby containing a mouse sensor and speed control. She then trains a robot to respond to collisions of the ball. She trains another robot to deal with the ball missing the paddle and moving off the left side of the screen. She adds sound effects and plays with her game.

Summary and Future Plans

We have presented what we believe is the first system to support an animated source code and to use video game technology to support general purpose programming. As of February 1995, ToonTalk has operational versions of all of the constructs described above. The Martian character is operational as a coach and help system but not yet as a tutor.

Testing of ToonTalk in a fourth-grade class and in some homes began in January. Several children have played with ToonTalk for an hour or so. One encouraging observation from this casual use by children is that it seems to be succeeding in providing an entertaining way of constructing programs. Children like to play with the birds, hand-held vacuum, bike pump and magic wand, watch houses being built and destroyed, fly around in the helicopter and so on even if they are not constructing a useful program. Today it is too early to evaluate how well children can use ToonTalk to build programs, but it every encouraging that they find that just playing around with the equivalent of the program editor lots of fun.

ToonTalk runs on 386 or better PC running Microsoft Windows 3.1 or better. A Mac port should not be too hard.

ToonTalk variants can easily be imagined. A virtual reality version of ToonTalk would enable one to build programs from inside VR. Since ToonTalk is built upon a concurrent foundation, a modem-based or networked version would be a natural extension. Multi-user games could be built and programmers could work together in the same world. I often think about what a professional version of ToonTalk would be like. A compiler could augment the system's interpreter and would make more ambitious programs much faster and smaller.

ToonTalk is a real programming environment and yet it does not rely upon a keyboard. A keyboard is handy for various accelerators but one can get by with just a game pad, joystick or mouse. This opens up the possibility of a port to a game machine. We are excited by the possibility that the millions of kids around the world who have a game machine at home might be able to use it to make their own games and do real programming.

Acknowledgments

I am very grateful for the help, advice and support I have received from many people during this project. In particular David Kahn, Mary Dalrymple and Markus Fromherz deserve special thanks for all their help. I am very grateful to Greg Savoia for the wonderful artwork and animation he contributed to ToonTalk.

References

Thomas Malone. *What Makes Things Fun to Learn? — A Study of Intrinsically Motivating Computer Games*. Stanford University Psychology Department doctoral thesis, 1980.

Seymour Papert. presentation at the August 1977 Logo Users Meeting, MIT.

Seymour Papert. *Children's Machines*. Basic Books, 1993.

Eugene Provenzo. *Video Kids — Making Sense of Nintendo*, Harvard University Press, Cambridge, Ma. 1991.

Mitchell Resnick. Multilogo: A study of children and concurrent programming. Technical report, MIT Media Lab, 1988.

Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus—A step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. MIT Press, October 1990.

Vijay A. Saraswat. *Concurrent constraint programming languages*. Doctoral Dissertation Award and Logic Programming Series. MIT Press, 1993.

Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 1989.

David Smith, *Pygmalion: A Creative Programming Environment*, Stanford University Computer Science Technical Report No. STAN-CS-75-499, June 1975.

Sharon Yoder, Discouraged? .. Don't Dispair! [sic], *Logo Exchange*, Vol 12, No. 4, Summer 1994, Journal of the ISTE Special Interest Group for Logo-Using Educators.

Internet access to the full-size version of this paper is available via anonymous ftp from csli.stanford.edu. The ToonTalk entry in pub/Preprints/INDEX contains more information.