ToonTalk™ -- An Animated Programming Environment for Children

Ken Kahn, Animated Programs
Kahn@CSLI.Stanford.edu

April 16, 1996 version

This paper was published in
The Journal of Visual Languages and Computing, Volume 7, number 2,
© Academic Press, June 1996.
This paper is a major revision and expansion of a paper by the same name published in the 1995 Proceedings of the National Educational Computing Conference

Key words: 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 [Pap77]. 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 ([Mal80] and [Pro91]).

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.

Children's play with Lego bricks can be usefully broken into three components: design, construction, and use. For some children and some projects, design is very hard. They still get immense enjoyment and a feeling of accomplishment from constructing a toy by following detailed instructions and then playing with the resulting toy. Children with more experience or ambitious goals make creative variants of existing designs and novel designs of their own. Similarly, with programming the hard part is *designing* programs. Building them usually consists of tedious activities like typing and fixing syntax errors. ToonTalk differs from other programming environments in that it makes the process of building a program fun and entertaining. Beginning ToonTalk users may be no more creative building programs than when they build a model airplane or follow detailed Lego instructions. But in all these cases, the process of building, as well as playing with the result, can be fun. One very effective way that children learn about design is by playing with good designs by others.

ToonTalk, in contrast to Logo or Basic, was designed so that a child at home could become a proficient programmer without human instruction. For instance, a recent survey indicated that in the United States the average amount of time high school students spend programming computers was less than 20% of the total time using computers. [Bec93] For most children this is too little to enable them to master the skill of programming.

The Logo community is also interested in giving children the ability to program for epistemological reasons [Pap93]. 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 [Yod94]. 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 both to be 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 those 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.

The challenge is to take a good theoretically oriented language design that is small and powerful and find a consistent set of *concretizations* that cover every language construct.

ToonTalk is based upon a concurrent constraint language [Sar93] similar to Janus [SKL90] and Linear Janus. Janus is similar enough to other concurrent programming languages that programs written in concurrent logic programming languages like Flat Guarded Horn Clauses, FCP, Parlog, Strand, and PCN [Sha89] can be straight forwardly constructed in ToonTalk. Conversely, ToonTalk programs, despite having been constructed in an interactive animated fashion, can be translated to textual equivalents in these languages.

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 [Sha89]. 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. [Res88] 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.

Why existing visual programming languages aren't enough.

Much of the research on visual programming languages shares the same goals as this work. The idea is that by using pictures to represent programs, programming will be easy to learn and programs easy to understand. Anecdotal evidence suggests that powerful visual languages are somewhat easier but that they are still difficult to learn and master. Other visual languages are limited in scope or expressive power but are easy to learn and use (e.g. KidSim [Smi94]).

One shortcoming of most visual programming systems is that they don't animate the execution of programs -- or if they do, the animation is limited to highlighting some picture elements. Most systems that do produce animations of program executions start with annotated textual programs (e.g., [Bro87]). Pictorial Janus [KS90b] was an attempt to remedy this by supporting complete visualizations both of the program source and its execution. This helped users understand their programs and programs of others. It also made it easier to learn the underlying computation model. But it was not enough. Learning took time and only relatively sophisticated users (e.g., graduate students) ever mastered it.

We believe the difficulty is this: in order for a static picture to represent the dynamic behavior of a program, it needs to rely upon a rich set of encodings. Control and data flow need to be encoded in abstract diagrams. Abstract diagrams are arguably easier for people to deal with than symbolic formalisms, but they are still very difficult. Why not take the next step from static visual programming languages and begin to use dynamic images, i.e., animation, to depict the dynamic processes of a program?

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. While the advantages of animated source code are many, constructing animation is generally difficult and time-consuming. Good animation authoring tools help but it is still much more difficult to animate an action than to describe it symbolically.

Luckily, there is one sort of computer animation that is trivial for a user to produce -- video game animation. 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.

Video games, like literature and film, derive much of their popularity from a "suspension of disbelief". The human participant enjoys going along with the underlying fantasy. A help-screen in a video game or a badly focused scene in a movie can interfere with this enjoyment. To maintain a non-stop, consistent fantasy is difficult. For example, nearly every program that relies upon a graphical user interface supports various accelerator keys. A straight-forward addition of accelerator keys into a system like ToonTalk would interfere with the suspension of disbelief. To solve this, we made all of the tools into cartoon characters. For example, there is a hand held vacuum in ToonTalk for removing things. Normally, one uses it by moving the programmer persona's hand over it, grasping it, moving to the item one wishes to remove and then clicking a button to start the vacuum. Pushing a single accelerator key to accomplish this can be much faster, but how can it work without interfering with the underlying fantasy? Since the vacuum is an animated character, the story the system portrays is that the accelerator key is just a way to whistle for the vacuum to come run to your hand.

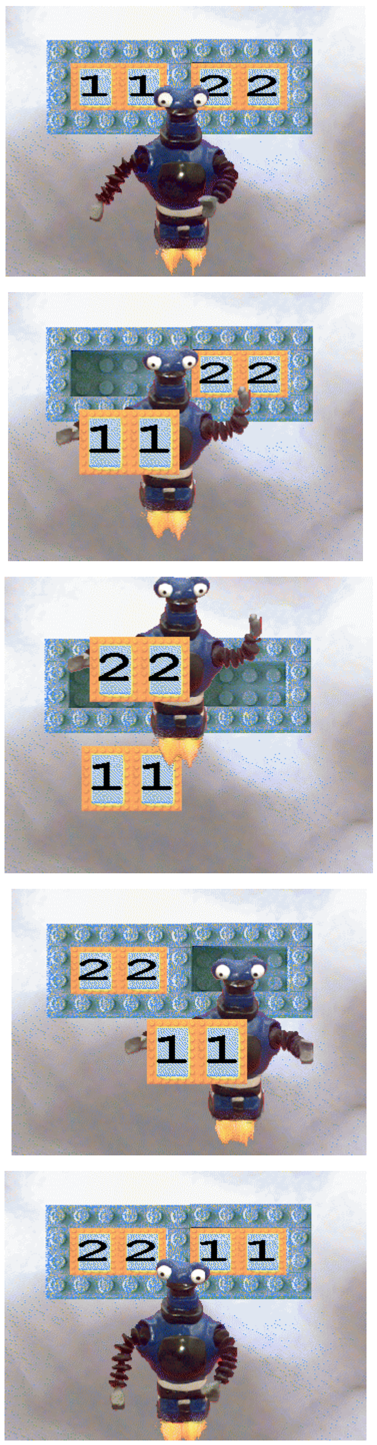


Figure 1 -- Snapshots from swapping two elements. (color original)

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, 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*).

There has been much research on *demonstrative* or programming by example systems. Perhaps the earliest was Pygmalion, built in the mid 1970s by David Canfield Smith as part of his doctoral thesis [Smi75]. He even spoke of programs as "movies". But there was no animated game-like world in which the programmer operated. Instead, icons and menus were selected and a sequence of such selections could be iconized.

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. After all, a Turing Machine is both a concrete and a universal computing formalism. (Alan Turing strove not just for mathematical computing formalisms but for their concrete analogs as well.) One can imagine a Turing Machine game that in theory supports the construction of arbitrary computations, but I can't imagine using it to build real programs.

The ToonTalk world resembles a twentieth century city. There are helicopters, trucks, houses, streets, bike pumps, toolboxes, hand-held vacuums, 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 the 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 among houses is accomplished by homing pigeon-like birds. 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 will 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 matching 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 using scales tipped one way or another, the conditionals can include less than, equal to, or greater than tests. (Example uses of these concretizations can be found in later sections.)

Synchronization is accomplished in ToonTalk by the fact that if a robot is given a cubby and its thought bubble includes items that aren't yet in the cubby, it will wait until those items appear. In particular, an empty hole may be filled or a bird may cover a nest in a hole by an item. If the robot expects to be given a particular kind of cubby (e.g., a box holding the number zero) and another item is in the cubby (e.g., the number one) then it will give the cubby to the robot behind it in line.

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 possible actions 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 (which drives off to build a new house),
- performing simple primitive operations such as addition or multiplication by building a stack of numbers (which are combined by a small mouse with a big hammer),
- copying an item by using a magician's wand,
- terminating an agent by setting off a bomb,
- changing a tuple by taking items out of compartments of a box and dropping in new ones.

These correspond to the permissible actions of a concurrent logic programming agent or an actor [KS90a, Agh87]. The last one may appear to a computer scientist to introduce mutable data structures into the language, which are known to introduce much complexity to parallel programs. Since boxes, however, are copied and not shared, this is not the case. An apparently destructive operation on a private copy is semantically equivalent to constructing the resulting state from scratch. But the destructive operation is often more convenient. In situations where there would be a performance penalty from unnecessary copying of boxes, a house can be built to hold a single copy and sharing can be accomplished by copying birds which deliver requests to a nest in that house.

When the user controls the robot to perform these actions, she is acting upon concrete values. This has much in common with keyboard macro programming and programming by example [Smi75]. 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 disrupting the ToonTalk metaphor. The initial notebook contains sample programs and access to facilities like animation and sounds.

One can understand ToonTalk completely in its own terms. E.g., a bird, when given something, flies to her nest, leaves the item there and returns. This is how kids typically understand it. Computer scientists, however, might like to understand the relationship between computation and these ToonTalk objects and activities. Here's the mapping between computational abstractions and ToonTalk's computational concretizations:

Computational	**ToonTalk**
computation	city
agent (or actor or process or object)	house
methods (or clauses or program fragments)	robots (with thought bubbles)
method preconditions	contents of thought bubble
method actions	actions taught to robot inside thought bubble
tuples (or arrays or vectors or messages)	cubbies
comparison tests	scales
agent spawning	loaded trucks
agent termination	bombs
constants	number pads, text pads, pictures
channel transmit capabilities	birds
channel receive capabilities	nests
program storage	notebooks

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 confusing 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. In addition, 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.

Currently the robots and boxes are just placed on the back side. Soon users will be able to build an entire city on the back side of any sprite. This design supports complex and dynamic behaviors for sprites. The metaphor of cities on the back of objects is, admittedly weird and perhaps inconsistent with a twentieth century city theme, but it does match the semantics well, should be easy to learn and use, and can be fun (the idea of cities inside of cities inside of cities inside of ...).

In addition to turtle graphics, many Logo implementations include libraries for controlling motors and sensors in a Lego, Fisher Technic or Capsula construction set [Pap93]. 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. The interface is very similar to the remote controls for sprites. A remote control for, say, a motor can be obtained and used like any number. If its value is set to, say, 5 the motor will be turned on and its power level set to 5. If it is given to a robot trained to repeatedly add 1 to a number then the motor will speed up. Similarly, there are ToonTalk sensors which are numbers that are continually updated to show the status of its connected Lego sensor. This is also the way ToonTalk deals with other input devices like the computer's mouse.

While our experience with controlling Lego devices using ToonTalk 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. Also the transition from direct manipulation to program manipulation is very easy in ToonTalk.

ToonTalk Tools

ToonTalk needed to provide the user with the ability to copy, remove, restore, and change the size of items. The first design used three magic wands that were visually distinguished by color and abstract animation for the different functions. Rather than one wand for each of the five functions, opposite functions were collapsed into one tool. The wand for removing objects could also restore previously removed objects. Another wand could both grow and shrink objects.

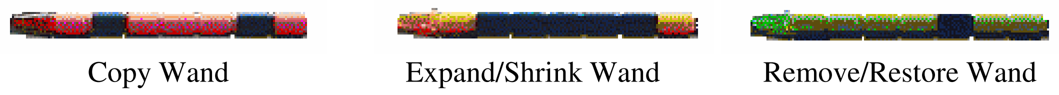
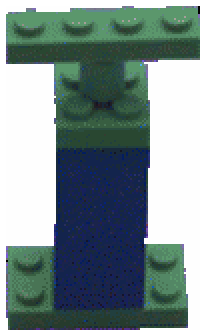


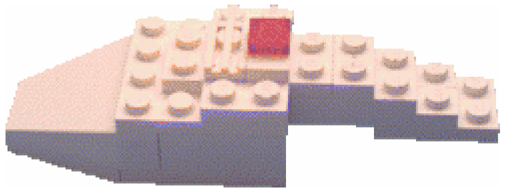
Figure 2 -- First version: 3 Magic Wands (color original)

Preliminary informal user studies showed that this scheme was non-obvious and users got the wands confused despite the colors and animation. In order to make the functionality of each tool easier for users to guess and to make the tools easier to distinguish from each other, I replaced the size changing wand with a bike pump. I replaced the removing and restoring wand with a hand-held vacuum.

I could have replaced the copy wand with any number of metaphors like a copier or an instant camera. However, a copier would not fit well since the other tools are used by holding them in your hand while a copier is stationary and you take things to it. Furthermore, a copier or camera metaphor might have implied that the result wasn't a functioning copy but just a picture of the original item. The disadvantage of using a magic wand to copy things is that it is hard to guess its function from its appearance and it doesn't fit the theme of bike pumps and dust busters. This was alleviated somewhat by making the wand look like the kind of magic wand that twentieth century magical performers typically use.

New Copy Wand





Bike Pump

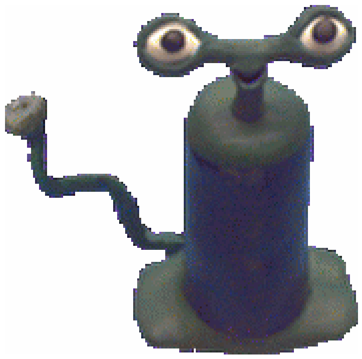
(replaces Grow/Shrink Wand)

Hand-held Vacuum

(replaces Remove/Restore Wand)

Figure 3 -- Current version: Magic wand, bike pump and hand-held vacuum (color original)

This new design also facilitated a tool/character duality. The bike pump and vacuum in ToonTalk aren't just inert objects that one uses, they can transform into animated characters with the ability to act on their own. (See Figures 3 and 4.) A benefit of animated characters is that they can get out of the user's way when not needed and they can come to the user when needed. It also increase the fun, appeal and aesthetics of these tools. The tools morph to their inert state when not being used and therefore don't distract the user. This duality of object and character has also worked well for the toolbox and notebooks in ToonTalk. The duality is a fun fantasy that has been exploited very successfully in movies like Disney's *Beauty and the Beast*.



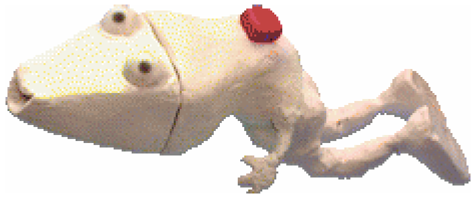


Figure 4 -- Bike pump and vacuum as animated characters

ToonTalk Communication

The initial design for communication between robots in ToonTalk mirrored the postal system with mail boxes, postal carriers and return addresses. But the semantic match was poor since the kind of communication in ToonTalk allows for the *communication* of the ability to send or receive messages. Forwarding addresses only partially accomplish this. I evaluated many other designs including rafts on rivers, faxes, email, and magical forms before deciding on carrier pigeon-like birds and nests.

Not only is the semantic match perfect but the birds and nests design meets the other criteria of ease of learning and use, fun and appealing, and consistency with the overall theme of a city. Even small children find it easy to understand that if you give a bird something, she'll take it to her nest and fly back. You can even have a bird deliver a box with another bird or nest in it.

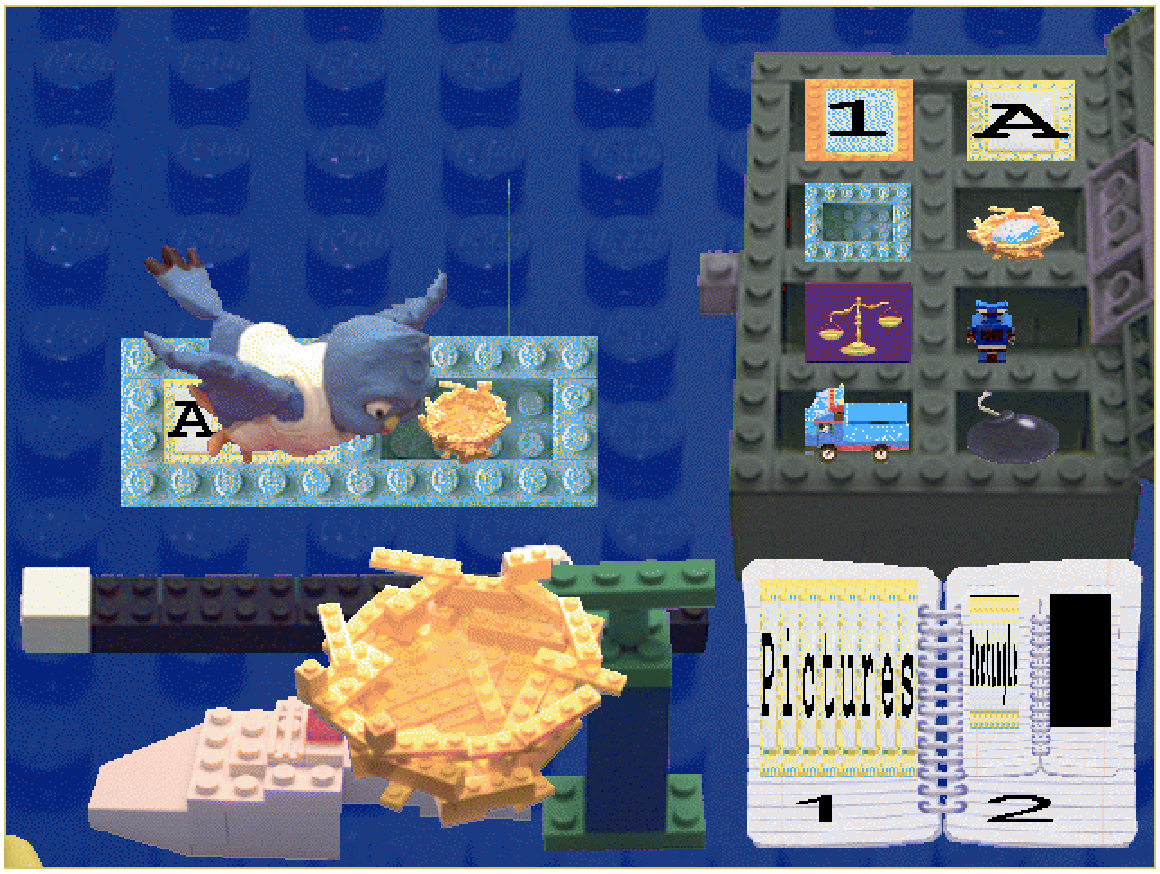


Figure 5 -- A bird delivering a box with a nest to its nest

ToonTalk Arithmetic

One might think that the metaphor for doing arithmetic in ToonTalk is obvious -- give the user a virtual calculator. Indeed this was the plan for quite a while. While a calculator is familiar and thereby should be easy to learn and use, it doesn't support well what is common in ToonTalk: computing with pre-existing numbers. Consider a user who needs to add two numbers. The user might drop a number on the calculator to input that number, press the "+"

button, and drop the other number, press the '=' button, and then finally copy off the resulting number. Altogether, this is an awkward process that isn't much fun.

We considered alternatives such as adding by stacking and multiplying by using the copying wand to make "n" copies. While pedagogically appealing, it was hard to cover the full range of operations (e.g., division) and also to work with non-integer values.

ToonTalk arithmetic is now accomplished by a mouse with a very big hammer. If you place a number on another number, the mouse runs out and smashes them together, leaving behind their sum. If instead you put "x5" on top, it'll multiply the number underneath by five. All of the standard arithmetic operations are available in this manner. Working with children has shown that this is easy to learn and use and they find it fun and appealing. Another benefit of this scheme is that it generalizes to text (putting text on text to perform concatenation) and pictures (putting pictures on pictures to form composite groups).

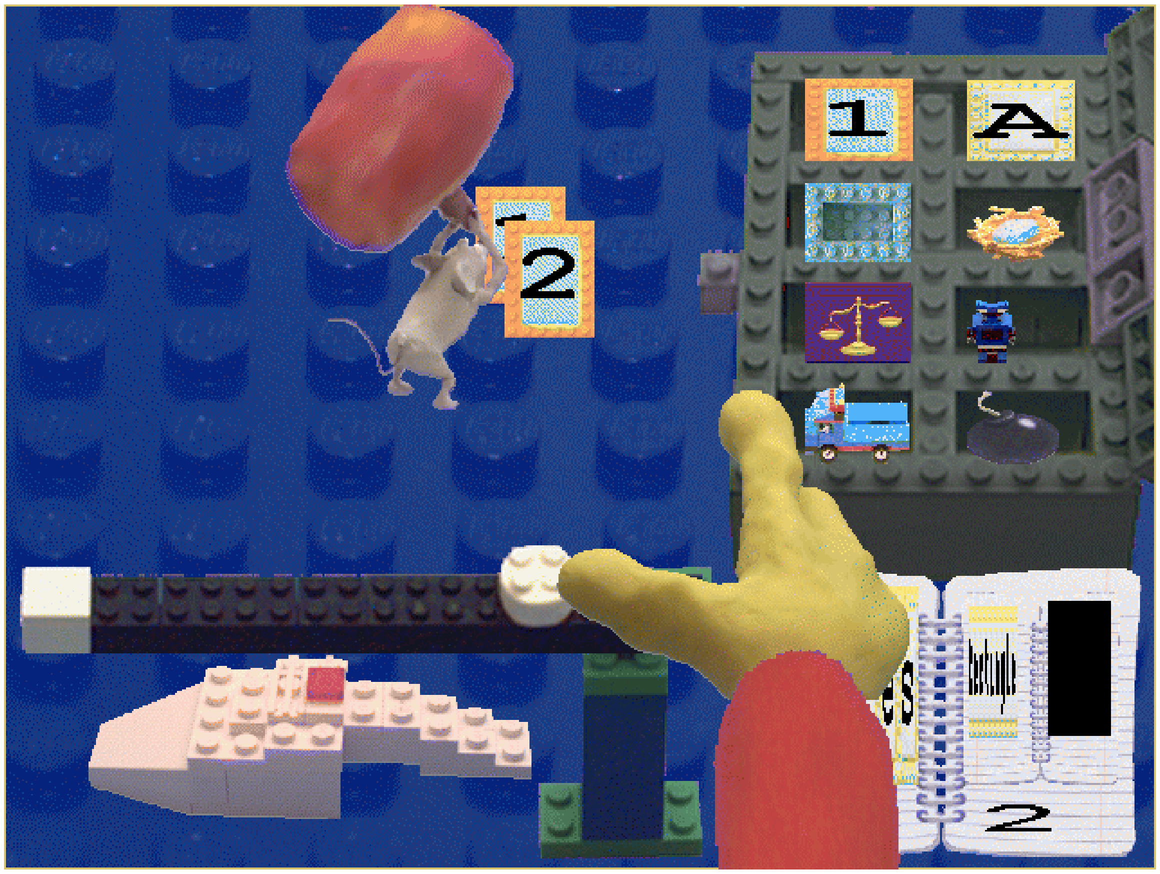


Figure 6 -- A mouse performing 1+2

Building a Ping Pong game in ToonTalk

In an attempt to provide a more detailed understanding of ToonTalk, consider a child who wants to build a Pong-like game from scratch. (What follows is an attempt to describe in language an interactive animated experience. Much of the apparent complexity disappears when this material is presented in a demo. The current beta version of ToonTalk contains a demo featuring an imaginary dialog between two children as they attempt to build a Ping Pong game. Unlike the following scenario, the process of exploration, debugging and experimentation in building the Ping Pong game is portrayed. Interested readers can contact the author regarding obtaining a copy of the beta test version of ToonTalk.)

ToonTalk is started and the user finds herself (or himself) flying in a helicopter over a city. Houses can be seen below.

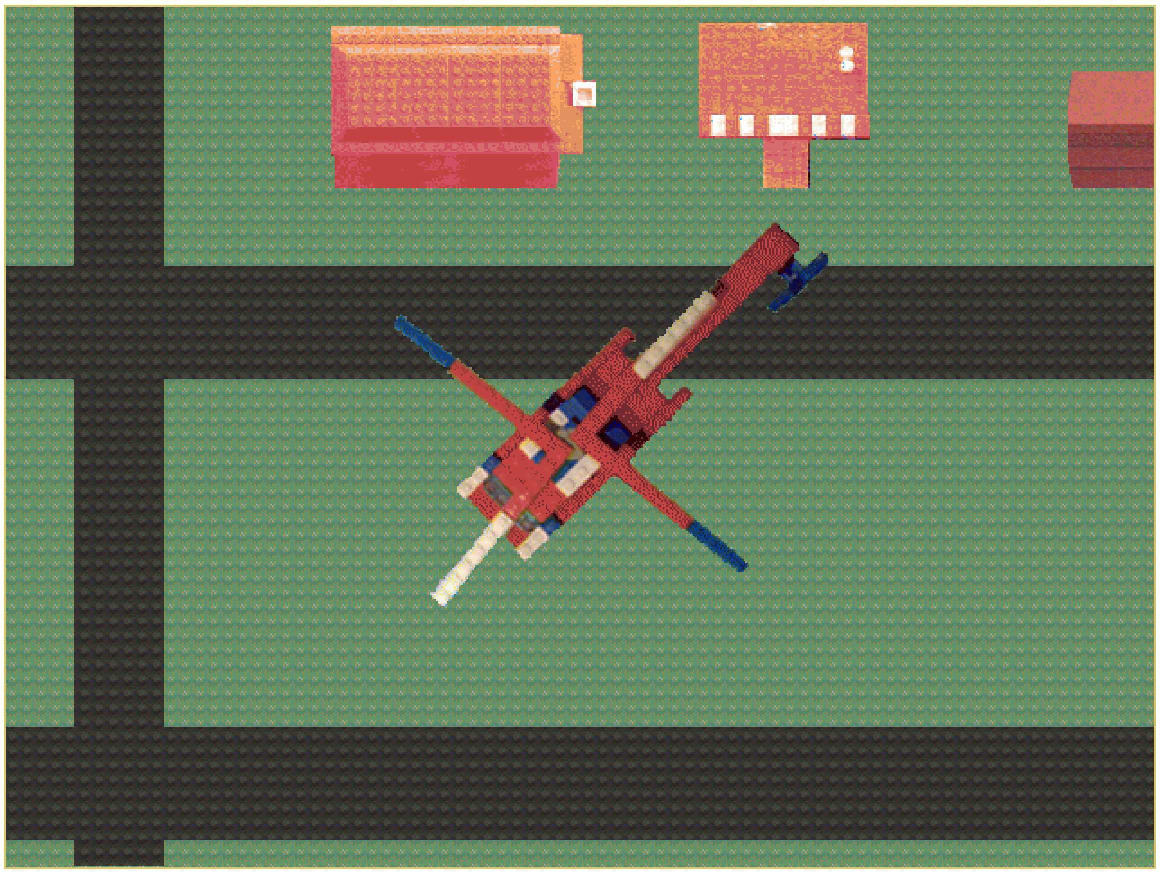


Figure 7 -- Programmer flying over houses

She lands in front of a house and enters the front door. Following her everywhere is a toolbox character.

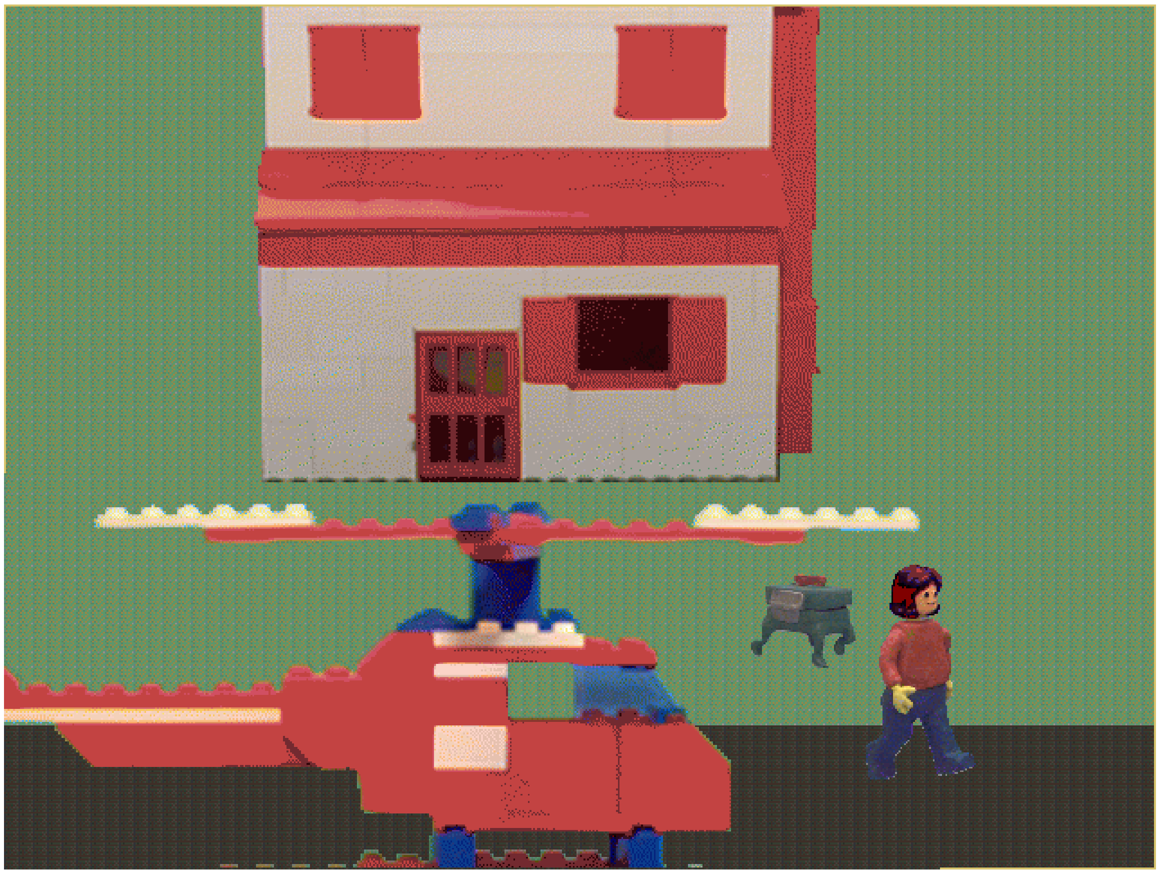


Figure 8 -- Programmer and toolbox outside

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 Ping Pong game on her own.

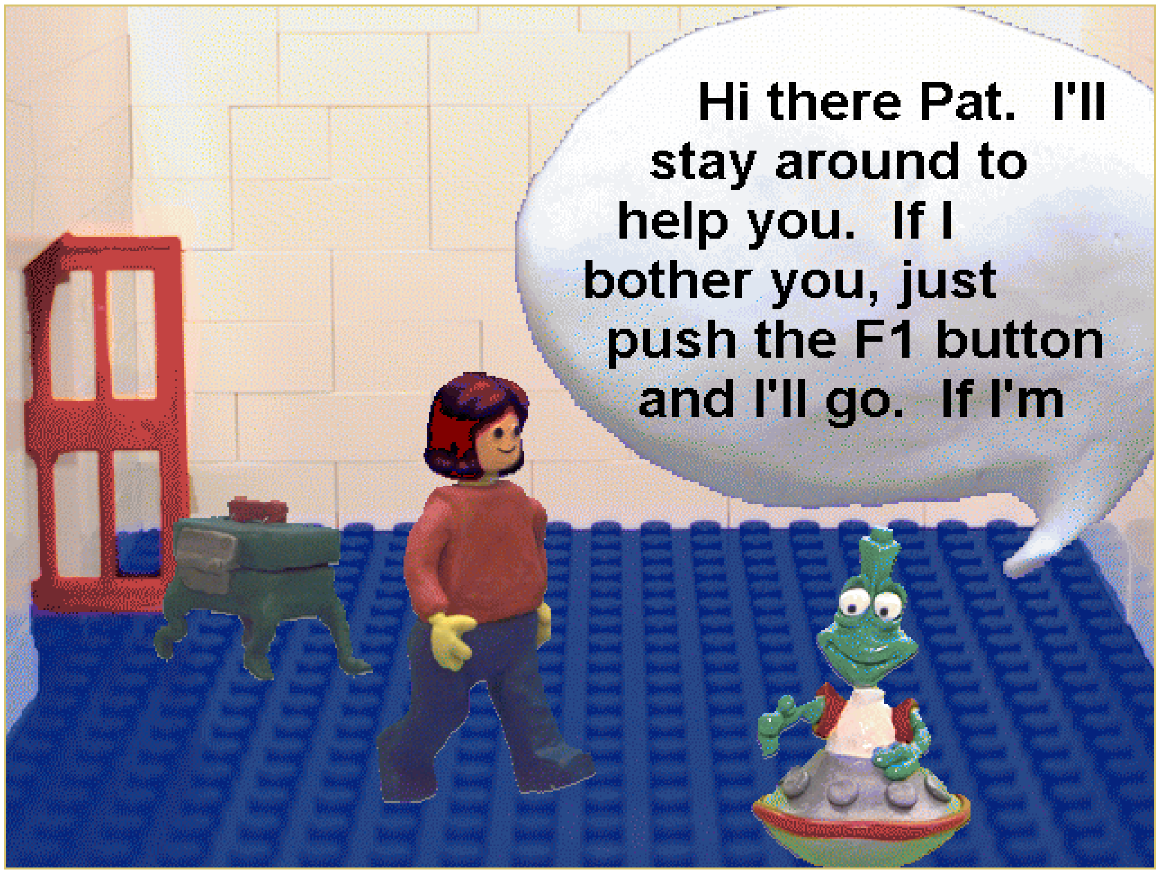

Hi there Pat. I'll stay around to help you. If I bother you, just push the F1 button and I'll go. If I'm

Figure 9 -- Programmer and toolbox inside house

She sits down on the floor and her toolbox scurries in front of her and opens. A notebook flies out. A hand-held vacuum with legs runs out. A bike pump hops out. 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.

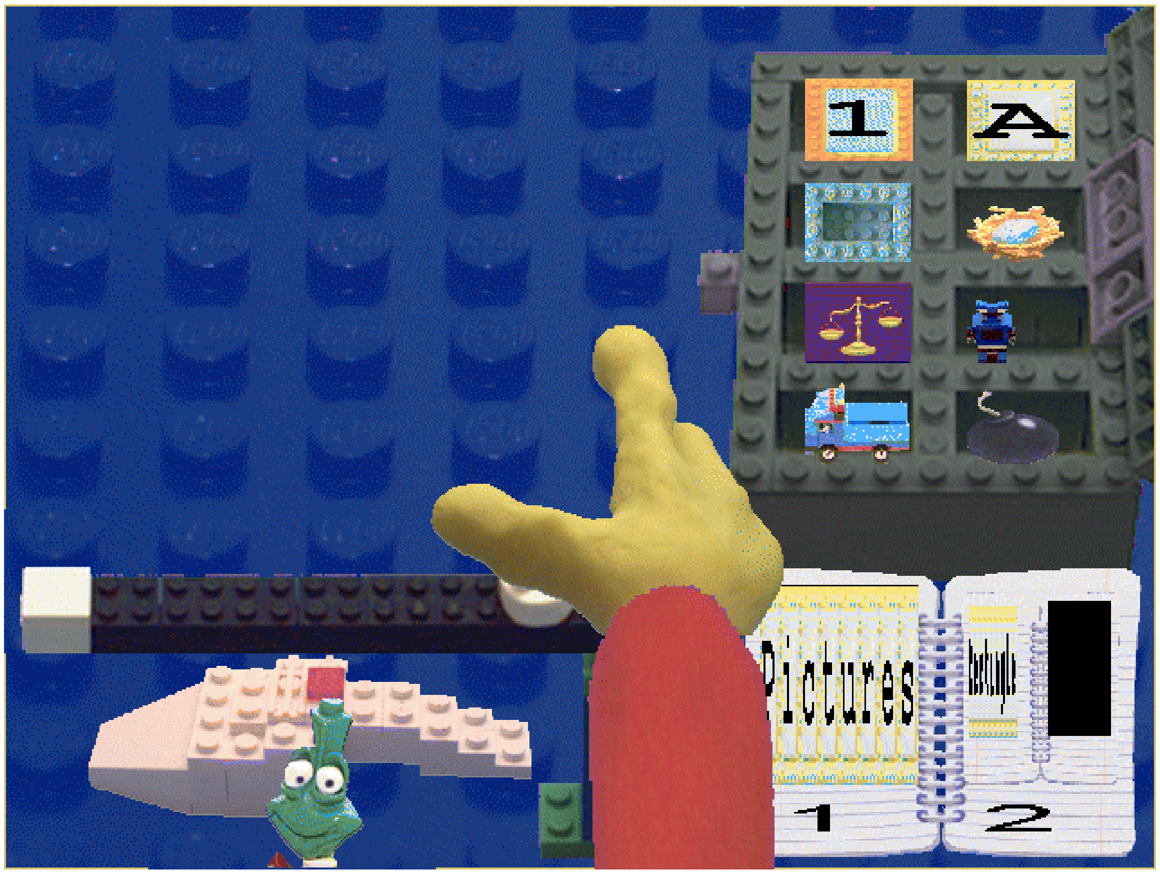


Figure 10 -- View of the floor after sitting

To build the paddle program she begins by flipping through the notebook on the screen until she finds a picture that she likes for the paddle. She reaches for it and gets a copy instead. Her plan is to put together a control panel for the paddle showing the paddle's vertical speed and the computer mouse's vertical speed. She plans to put a robot on the backside of the paddle that will just repeatedly copy the mouse's vertical speed over to the paddle. She begins by flipping the paddle over. On the back is a notebook full of remote controls for the paddle. She flips through until she finds a remote control for the vertical speed. She reaches for it and puts the copy she gets on the floor. She goes back to the original notebook and flips through it finding a notebook of sensors. She finds one for the computer mouse and take out the one describing the mouse's vertical speed . She drops it down next the paddle's vertical speed and the two boxes join to form one box with two compartments. (See Figure 11.) She then takes out a robot from the toolbox and gives it the cubby she just built. Since the robot hasn't been trained it doesn't know what to do with it. A copy of the cubby appears in its thought bubble and she sees an animation of the robot entering into its thought bubble.

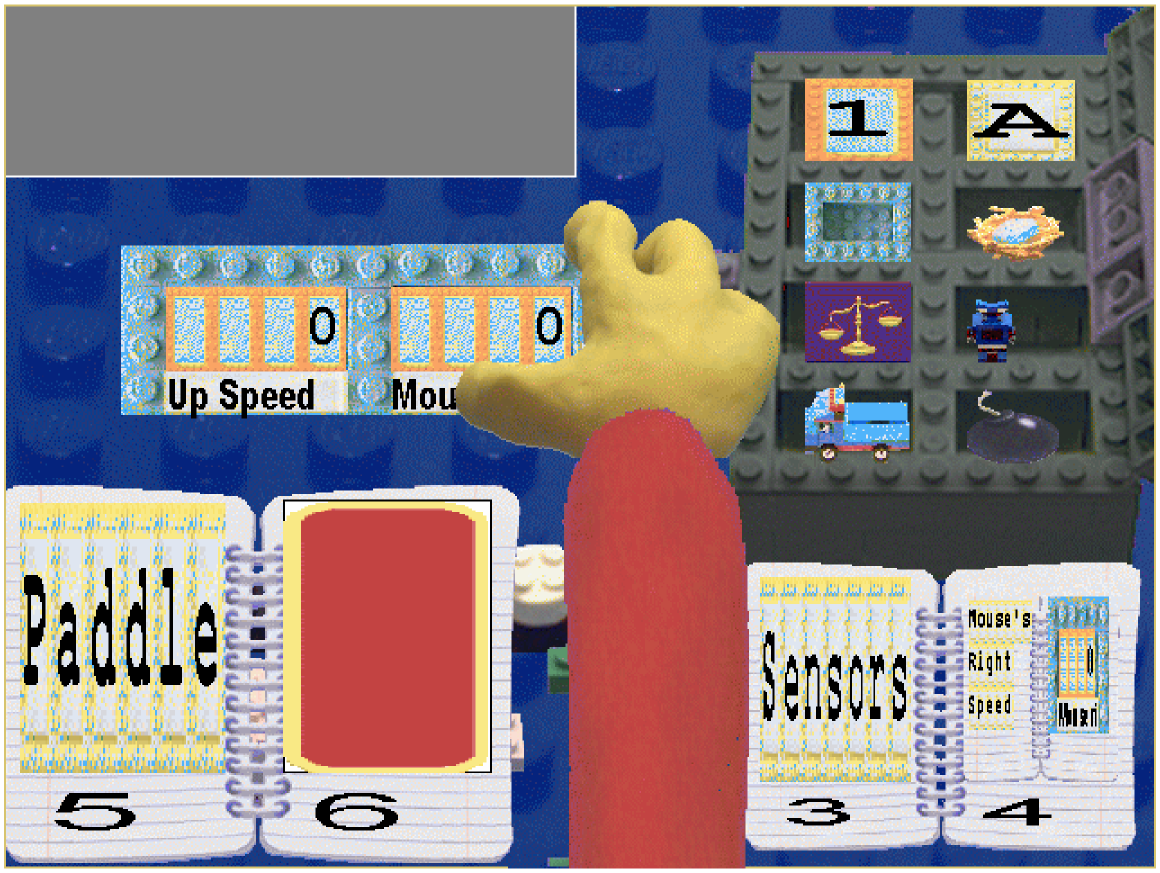


Figure 11 -- Just after building a cubby for controlling a paddle

She is now controlling the robot in order to train it within the thought bubble. Since all she wants is that the paddle should follow the vertical movements of the computer's mouse, she has the robot pick up the copy wand, copies the mouse's speed, types "=" and drops the result on the compartment containing the paddle's vertical speed. (See Figure 12.)

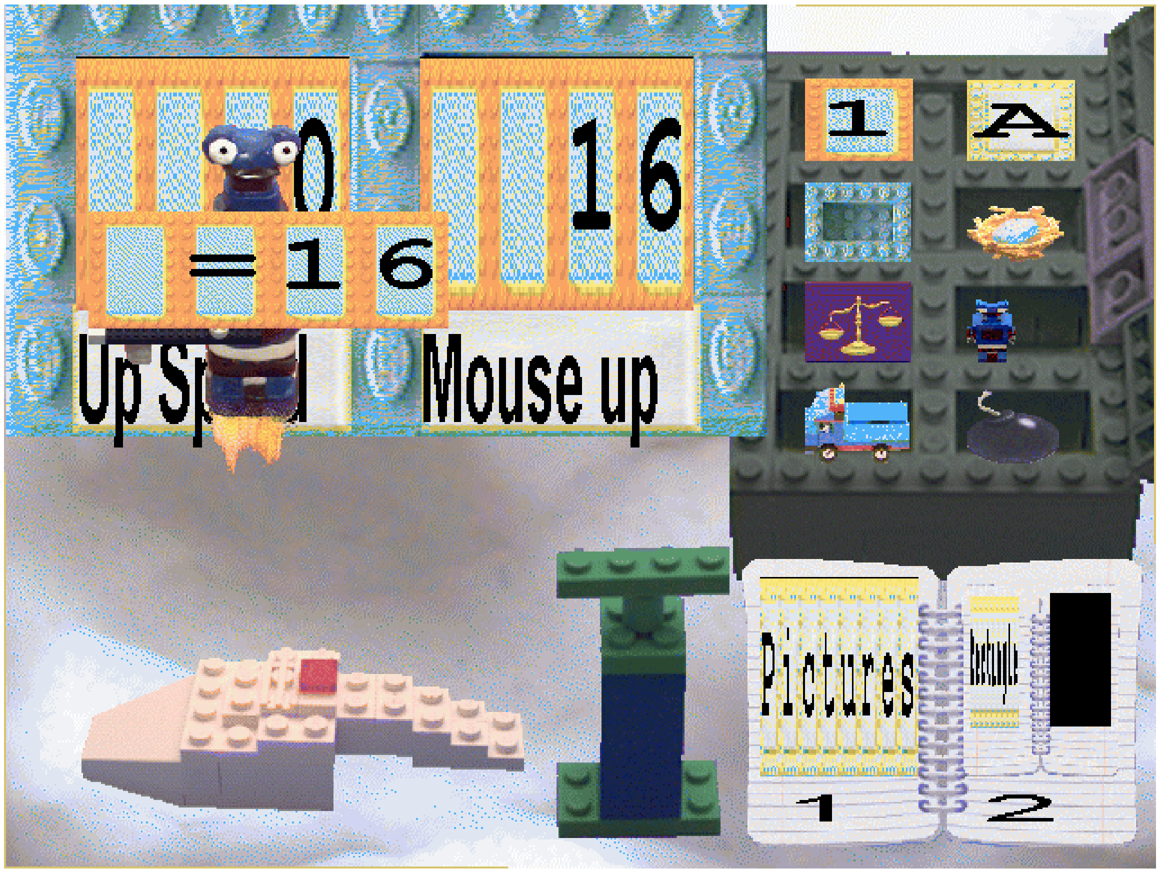


Figure 12 -- Training the robot so the paddle follows vertical mouse movements

The user pushes a button to indicate that she is done training the robot and the screen once again shows what's on the floor and that she is controlling the arm on the screen. In general, after training a robot, one needs to use the hand held vacuum to abstract the particular values in the thought bubble to blanks which indicate that any value is acceptable. When numbers which are remote controls are placed in a thought bubble, ToonTalk is able to automate this since the particular values cannot matter since they change dynamically. (They are like gauges instead of constants.)

She then drops the robot on the back side of the paddle. Again, she gives the robot the cubby. She flips the picture back over. The robot is now running on the backside of the paddle. On every frame it copies the mouse's vertical speed to the paddle. She tries it and the paddle now moves up and down as she moves the mouse and it ignores any horizontal movement. Happy with the result, she drops the paddle in the notebook for safe keeping for later. (It is now on the hard disk and will be there even if the computer crashes.)

The user now turns her attention to the ball. She wants the ball to bounce off of everything it collides with. She flips over a picture of the ball and gets its notebook. She focuses first on horizontal movement. She puts together a cubby containing a collision detector and a remote control for its horizontal speed. She gives this to a fresh robot and trains it to multiply the speed by -1.

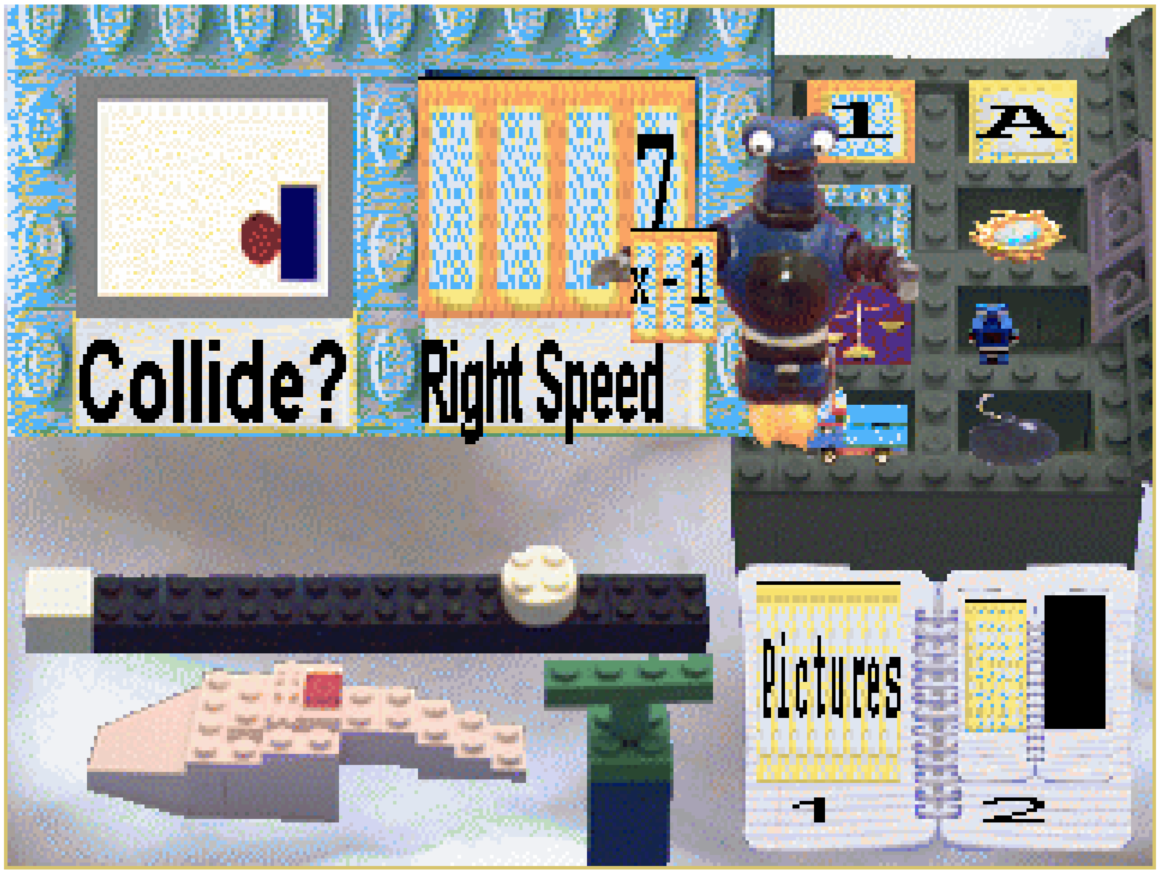


Figure 13 -- Training robot for ball bouncing

The user puts the robot and its cubby on the flip side and tries it out and the ball bounces fine when moving east or west. She copies the robot and gives it a similar cubby but with remote controls for vertical motion. She tries it out and is happy with how it is bouncing. She trains another robot to make a sound whenever there is a collision.

She takes out a background rectangle for the game and places on it three rectangles to act as the top, bottom, and right walls for the ball to bounce off. She adds the paddle and gives the ball a toss (by holding down a modifier key when releasing the ball). She plays with her new game and it works fine except that if she misses the ball with her paddle the ball continues traveling to the left without stopping. Using scales, she trains a robot to act whenever the horizontal position becomes less than zero. The robot makes a different sound and restarts the ball on the right side of the rectangle.

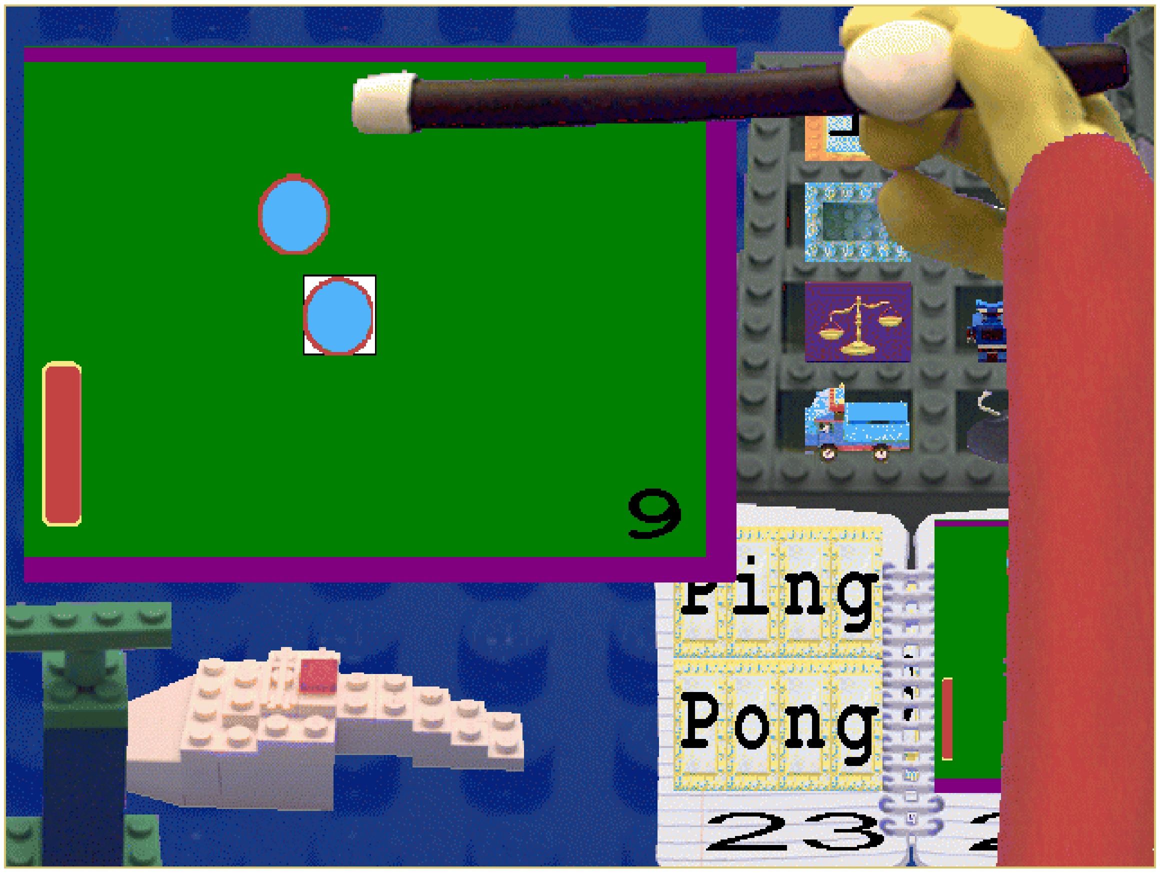


Figure 14 -- Adding another ball to the paddle game

She plays with her game a while, then drops the whole thing in her notebook to save it. She then takes it out and puts another ball in it to make it harder (see Figure 14). It is too hard so she drops another paddle in as well. She starts to think about how to add a score keeper and starts thinking how she can turn her Pong-like game into a Breakout-like game by getting some exploding bricks that her classmate built yesterday and adding a bunch of them to 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 November 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 1995. Over 75 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 is very encouraging that they find that just playing around with the equivalent of the program editor is lots of fun.

Limited testing has shown that while children easily master the objects and tools in ToonTalk (i.e. the "primitives" of the programming language) they need guidance to continue. The system is currently being enhanced with examples, demos, instruction sheets, and an interactive tutorial to enable them to continue on their own.

The first version is built for a PC running Microsoft Windows. 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. Big thanks go to Ruth Colton and her 4th grade class at Encinal School in Menlo Park, California. Many thanks to Jichang Tan, Hattori, and other beta testers for reporting many bugs for me to fix. I am very grateful to Greg Savoia for the wonderful artwork and animation he contributed to ToonTalk.

References.

[Agh87] G. Agha, *Actors: A Model for Concurrent Computation in Distributed Systems.* The MIT Press, 1987.

[Bec93] Henry Becker, Teaching *with* and *about* computers in Secondary Schools, *Communications of the ACM*, May 1993, Vol. 36, No. 3

[Bro87] Marc H. Brown. *Algorithm Animation.* The MIT Press, 1987.

[KS90a] Kenneth Kahn and Vijay Saraswat. Actors as a special case of concurrent constraint programming. In *Proceedings of the Joint Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*. ACM Press, October 1990.

[KS90b] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop*, October 1990.

[Mal80] Thomas Malone. *What Makes Things Fun to Learn? -- A Study of Intrinsically Motivating Computer Games,* Stanford University Psychology Department doctoral thesis, 1980.

[Pap77] Seymour Papert. presentation at the August 1977 Logo Users Meeting, MIT.

[Pap93] Seymour Papert. *Children's Machines*. Basic Books, 1993.

[Pro91] Eugene Provenzo, *Video Kids -- Making Sense of Nintendo*, Harvard University Press, Cambridge, Ma. 1991.

[Res88] Mitchell Resnick. Multilogo: A study of children and concurrent programming. Technical report, MIT Media Lab, 1988.

[SKL90] Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus--A step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. The MIT Press, October 1990.

[Sar93] Vijay A. Saraswat. *Concurrent constraint programming languages*. Doctoral Dissertation Award and Logic Programming Series. The MIT Press, 1993.

[Sha89] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 1989.

[Smi75] David Smith, *Pygmalion: A Creative Programming Environment*, Stanford University Computer Science Technical Report No. STAN-CS-75-499, June 1975.

[Smi94] David Smith, Allen Cypher and Jim Spohrer, KidSim: Programming Agents without a Programming Language, *Communications of the ACM*, Vol. 37, No. 7, July 1994.

[Yod94] 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.