Source Code Should be Animated

Ken Kahn, Animated Programs

This is the July 24, 2001 draft of an article written for Dr. Dobb's Journal.

Introduction

Visual Programming is the idea that source code would be better if text were replaced or augmented with pictures, diagrams, or icons. A visual program consists of formal diagrams that encode descriptions of dynamic processes. In a 1965 PBS visit to MIT Lincoln Labs, we see Bert Sutherland (now head of Sun Labs) showing off his thesis system -- a way of entering and running programs as flow charts. We hear Bert saying that programmers used to program with text but now they program directly with pictures. But Sutherland was wrong: despite 40 years of attempts to build visual programming systems, very few programs are visual. Though Visual Programming is a good fit for those few of us that are *both* formal and visual thinkers, visual programming didn't succeed because it doesn't go far enough.

Animated Programming is the idea that source code should not be a static abstract encoding of a process, but rather a dynamic and concrete description of a process. And the source code should be animated like a video game is animated, not an animated diagram or linked animated icons. Animated Programming is a bit like sign language or a game of Charades. And like sign language, there is no loss of expressivity; an animated programming language can be Turing-equivalent. As a matter of fact, it would be straightforward to build an animated programming language for Turing Machines since they are visual, dynamic, and concrete.

So what kind of programmer is Animated Programming good for? Those of us who are good at thinking about visual tangible things and processes. That may be a larger population than today's textual programmers, who are good at formal abstract thinking. Animated Programming may be the best means to "end user programming". It may be the best way to introduce children to the joy and power of programming. It might be the best first language for adults that want to learn to program.

Why should professional programmers care about Animated Programming? Because it provides a different way to think about programming. And having another way to think about programming enriches your understanding. And it may in fact represent the future of programming. Maybe the next generation of programmers will be programming from within virtual reality, by manipulating simulated objects.

ToonTalk - the only Animated Programming system today

To the best of my knowledge, ToonTalk is currently the only general-purpose animated programming language. There have been many special-purpose animated programming languages, such as The Learning Company's *Rocky's Boots* and *Robot Odyssey* from the early 1980s. These systems are limited to Boolean operations. While ToonTalk was designed for children, its ability to express and visualize concurrent algorithms has led to its use in university courses.

A brief introduction to the basic elements of ToonTalk

ToonTalk is an animated programming system whose design was heavily influenced by video games. The user of ToonTalk is a character in an animated world that resembles a modern city. She starts off flying a helicopter over the city. After landing she controls an on-screen persona. The persona is followed by a dog-like toolbox full of useful things.

An entire ToonTalk computation is a city. Most of the action in ToonTalk takes place in houses. Homing pigeon-like birds provide communication between houses. Birds are given things, fly to their nest, leave them there, and fly back. Typically, houses contain robots that have been trained to accomplish some small task. A robot is trained by entering his "thought bubble" and showing him what to do. New houses are constructed by loading a truck with a team of robots and a box for them to work on. (See Figure 1.) The truck will then drive off, and the crew inside will build a house. The robots will be put in the new house and given the box to work on. Boxes can have any number of compartments that can contain nearly anything, including other boxes.

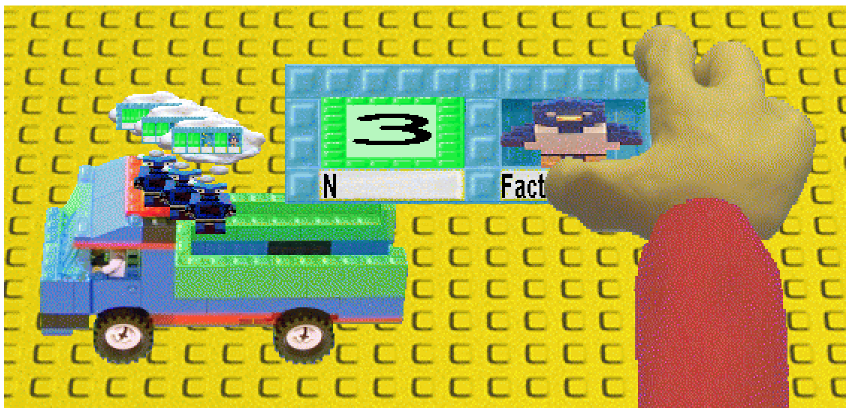


Figure 1 - A truck being loaded with robots and a box

The fundamental idea behind Animated Programming is to replace computational abstractions by concrete familiar objects. The behavior of a familiar object captures the essence of the corresponding computational concept. Even small children understand that loading a truck in ToonTalk will create a new house with robots working inside. Computer scientists understand this as a way of expressing the creation of a computational process or task, where the team of robots corresponds to the code to be run and the box to the initial state. ToonTalk programmers can build, run, and debug programs understanding only ToonTalk's concretizations. Computer scientists and programmers can have a deeper understanding by relating these concretizations to well-known computational abstractions as presented in Table 1.

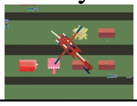
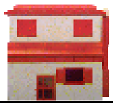
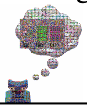
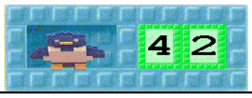
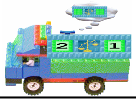
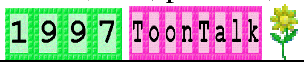
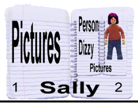
Computational Abstraction	**ToonTalk Concretization**
computation	city 
actor process concurrent object	house 
method clause	robot
guard method preconditions	contents of thought bubble 
method actions body	actions taught to a robot
message array vector	box 
comparison test	set of scales
process spawning	loaded truck 
process termination	bomb
constants	numbers, text, pictures, etc. 
channel transmit capability message sending	bird
channel receive capability message receiving	nest
persistent storage file	notebook 

Table 1 - Computer Science Terms and ToonTalk Equivalents

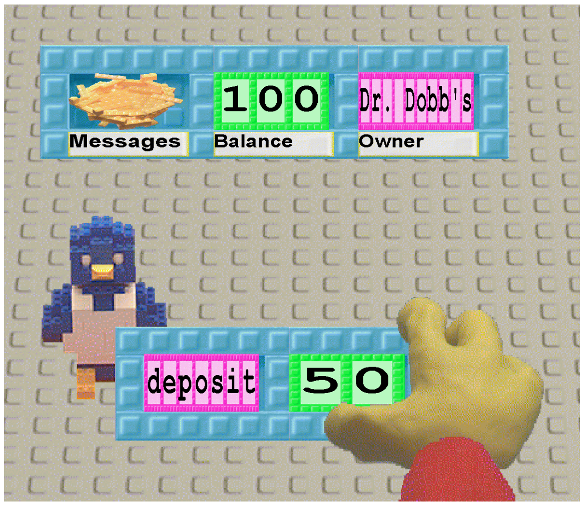
A robot behaves exactly as the programmer trained him. This training corresponds roughly to defining the body of a method in an object-oriented programming language such as Java or Smalltalk. A robot can be trained to

- perform 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);
- copy an item by using a magician's wand;
- change a data structure by taking items out of a box and dropping in new ones;
- send a message by giving a box or constant to a bird;
- spawn a new process by dropping a box and a team of robots into a truck (which drives off to build a new house); or
- terminate a process by setting off a bomb.

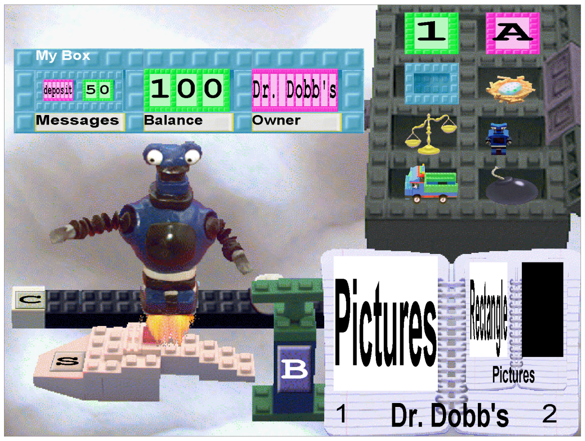
Consider the cognitive difficulty that many programmers have when they first learn about recursive procedure calls. Recursive process spawning is even harder to understand. Contrast this to the ease of understanding the idea of a robot being trained to load a truck with a copy of himself (or, more accurately, a copy of the team he is a member of).

Distributed programming in ToonTalk -- A Bank Account Example

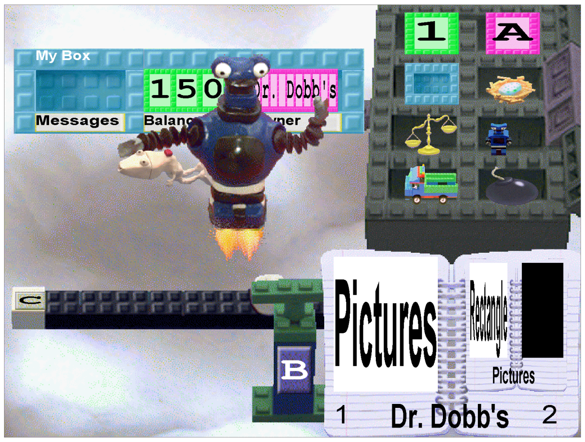
To better understand ToonTalk, consider an example of building a bank account. In the interests of brevity and pedagogy, this is a simple account that accepts deposit requests and balance queries. An account just consists of a current balance and the name of the owner. A realistic account could be implemented using the same principles and techniques.



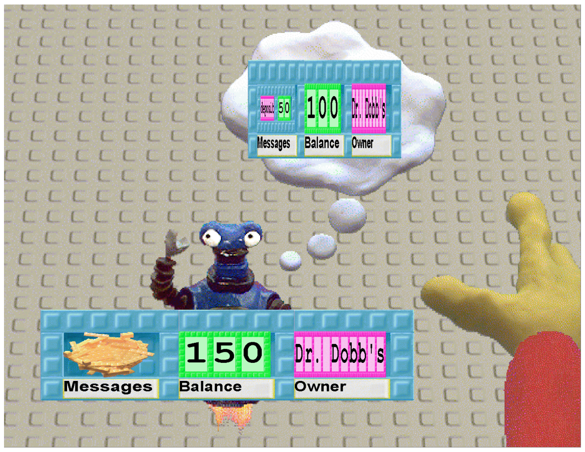
To start we fill a box with a nest, a number pad, and a text pad. Note that *Messages*, *Balance*, and *Owner* are just annotations. In the message compartment is a nest where messages will be delivered by birds. We next construct a sample deposit message and give it to a bird. A deposit message is just a box with a text pad displaying the word "deposit" and a number pad.



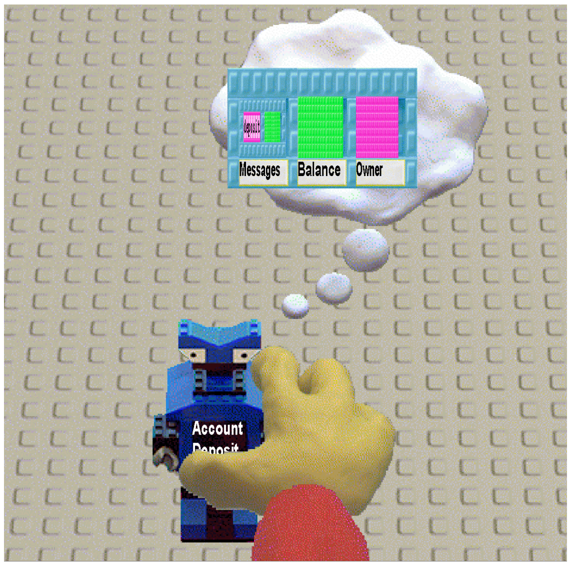
The bird takes this box and covers her nest with it. We then give the original box to a fresh robot and enter into his thought bubble. We are no longer controlling the programmer persona and instead are controlling the robot.



We train the robot to pick up the number pad with the amount of the deposit (50) and drop it on the number pad with the current balance (100). We then sit back and watch as a mouse with a very big hammer smashes the number pads together, producing a number pad with 150 on it. We pick up Dusty, the hand-held vacuum, and suck up the box representing the deposit message since we have completed processing it.



We press a button to indicate that we are finished training the robot and once again are controlling the hand of the programmer's persona. Since it is good practice to comment your code, we now give our robot the name *Account Deposit*. We test our freshly trained robot by giving him the box once again. This time he knows what to do and repeats the actions we taught him.



We give the bird a box representing a *deposit 25* message and we see our robot shrug and give up. The problem is that the box he is working on no longer matches the box in his thought bubble. We pick up Dusty, move him over the robot's thought bubble, and erase the number pads for the balance and deposit amount and the text pad for the owner. Our robot now processes the new message.

Figure 3 – Training a robot to handle deposit messages

From a conventional programming perspective, we have just defined a deposit method. It has a single argument of type integer. The method increments the object's state variable that represents the balance by the argument. We have also tested our method on two sample calls.

We next construct a box for querying the current balance. It contains a text pad with the text "current balance?" on it and a new bird. The bird is there to receive the answer and bring it to her nest. This new nest is where we'll receive a reply to our query. We give the box to a fresh robot and train him to pick up the Magic Wand and copy the number pad in the box and give the copy to the bird. (See Figure 4.) We then train the robot to vacuum up the incoming message. In conventional programming terms, we have just defined a query method that replies with the balance state variable.

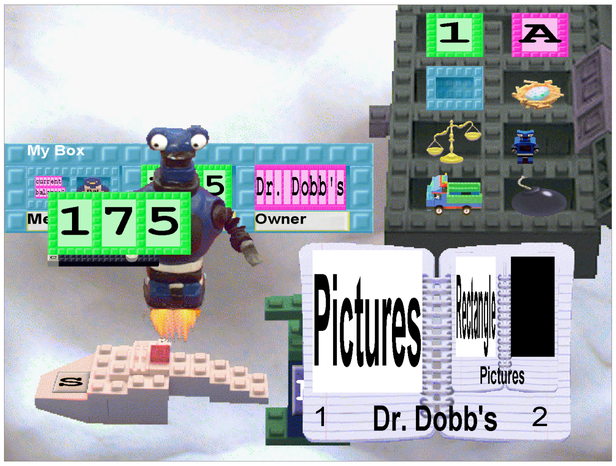


Figure 4 – Training a robot to respond to balance queries

We drop one robot on the other and they form a team. We place our team in our notebook so we can get a copy of our robots wherever we are. We add a comment on the opposite page. We then go on to train robots to implement a bank with lots of accounts. Each account is created by a robot that drops our bank account team of robots in a truck together with a box containing the initial balance and name of the owner. All parts of the system that need access to an account retain a copy of a bird who flies to the nest in the box used by the account robots.

How is this distributed programming? Because a bird can be saved and reconstructed on a different computer running ToonTalk. Birds can fly to the computer where their nest is and deliver deposit or query messages. And a bird in a query message can fly back to deliver a reply. If a bird is delivering something to her nest and it is already covered then she places her item underneath the earlier ones. The nest acts like a queue of incoming messages that our team of robots process in the order they arrived. There are no race conditions, locks, or deadlocks to worry about.

ToonTalk's model of computation

The idea of Animated Programming is to provide a mapping from the components of a model of computation to concrete objects and actions. ToonTalk is not based upon the widespread model consisting of the sequential composition of procedures that operate upon shared mutable data. It is instead a model of computation consisting of the concurrent composition of processes that operate upon un-shared mutable data and communication channels. The model has its origins in concurrent constraint programming and concurrent logic programming [Saraswat 93]. It is close to the actor model of computation [Lieberman 87].

Many are surprised to hear that a language designed primarily for young children is concurrent. They say that programming a single thread is hard enough without having to worry about interactions between concurrent activities. It is true that adding concurrency to a programming language based upon shared mutable data leads to easy to make and hard to fix bugs. In ToonTalk data is not shared; a team of robots has its own private box; boxes are not shared. This would severely restrict the expressive power of the language if there weren't communication channels. As we saw in the bank account example, multiple birds can share a nest, providing many-to-one communication. This doesn't lead to concurrency bugs since all that a bird can do to a nest is enqueue a message. There is no way for a conflict to arise.

What works well

The large number of children that have learned to program in ToonTalk provides evidence that Animated Programming can help a wide class of people understand how to program. A large three-year European research project called Playground (www.ioe.ac.uk/playground) has been helping 6 to 8 year olds to build computer games in ToonTalk.

At the other extreme, ToonTalk has been used in a few advanced university courses as an aid in teaching concurrent algorithms. This exploits the fact that not only can these concurrent algorithms be built using Animated Programming, but it provides a way to visualize the execution of these programs. In ToonTalk one can get an overview of the process structure, the inter-process communication, and the patterns of process spawning of an algorithm by flying the helicopter over a city running a program implementing the algorithm. One can view the computation at smaller scales by flying down and entering houses. The ToonTalk program can even provide layout information so that the houses (i.e. processes) are laid out in a grid or tree or other patterns.

Animated Programming also meshes well with Programming by Example. As in the bank account example, programming is done with tangible analogs of sample data. We didn't create a box containing variables for the current balance and owner. Instead we used a box with sample values in the box. Rather than rely upon some clever heuristics to make our programs general, we explicitly abstracted our program by removing details from the thought bubble of robots [Kahn 2001].

Animated Programming isn't just about giving programs an unusual syntax. It is also a way to build, debug, and visualize program execution in an integrated consistent manner. When your animated program has a bug, you need to search around in a virtual space to find something misbehaving. Frequently just watching it is enough to see what is wrong and how it needs to be fixed. ToonTalk provides an integrated program development environment that has the look and feel of a video game.

What problems and shortcomings remain

One shortcoming of Animated Programming is the lack of a way to edit programs. In ToonTalk a trained robot can't be edited to perform differently from his training. The best you can do is retrain him. When retraining a robot he repeats what he was trained to do until you explicitly take over. In theory, this means on average half the work of retraining is automated away. In practice, it is a bit more since one frequently forgets to train a robot to do something and it is easy to add to a robot's training. Editing at higher levels such as rearranging teams of robots or altering the contents of boxes can be done directly.

One can imagine a hybrid system that automatically converts between animated programs and textual ones. In such a system one could covert a robot to a textual form, edit him, and convert back. An intriguing alternative is to convert between animated programs and a storyboard or comic book format. The more static sequential comic book form is better suited for editing. And it provides an alternative way to understand what a program does. Rather than watch a program fragment evolve over time, the program fragment can be spread out in space and understood more readily. A comic book format would also solve the problem of how to present animated programs in print (e.g. this article). The idea of a comic book format for source code is the subject of a doctoral thesis. (www.ida.liu.se/~mikki/comics)

Another shortcoming of Animated Programming is that it is hard to support shared data structures. The three-dimensional world we live in does not contain objects that are simultaneously at different locations. A small version of this problem can be seen in the problems that people have with graphical desktops that support short cuts or file linking. The same file can be accessed from different folders. Confusion ensues when people rename or delete one of the file icons.

A system could be built where animated program fragments share the same data structure by moving to the unique location where the tangible analog of the data is. In a concurrent framework this would be a nice way to visualize race conditions. Pointers or sharing within data structures are hard to make tangible. Arrows can be used to represent pointers. How well this would work in practice is unclear. ToonTalk avoids this problem by limiting sharing to birds and their nests. Many birds may share the same nest but you don't see this sharing since it is in "the brains of the birds".

Garbage collection of unreferenced data is tricky in Animated Programming. By making data tangible you make its disappearance obvious. It is confusing and annoying to put something down and then later find it isn't there. In ToonTalk there is no need for garbage collection since data isn't shared. Explicit destruction of something is accomplished by vacuuming it up. Explicit termination of an activity and its local state is accomplished by setting off a bomb.

Where to go from here

ToonTalk is continually being improved. It is a success as a programming language for children. And it introduces concurrent programming in a gentle and intuitive manner. But ToonTalk is just one instance of the idea of Animated Programming. It was designed for children. It is based upon an exotic model of computation. Maybe the world needs more animated programming languages. Some could be for end-users, others for learning conventional programming concepts, and still others for professional programmers. And many of the disadvantages of Animated Programming can be avoided by supporting an auxiliary textual format. The challenge is to find a good coherent set of tangible objects and actions that capture completely the semantics of the desired programming language.

For many years I've been hearing that virtual reality systems will soon become mainstream. If that happens, I can't imagine a better way to do programming while inside of virtual reality than Animated Programming. What could be more natural than manipulating tangible objects and using tools to construct, run, and debug programs?

References

More information, a trial and a beta version of ToonTalk, and several papers are available at www.toontalk.com.

[Kahn 2001] Ken Kahn, “Generalizing by Removing Detail: How Any Program Can Be Created by Working with Examples”, in *Your Wish is My Command: Programming By Example*, Henry Lieberman, ed., Morgan Kaufman Publishers, 2001

[Lieberman 87] Henry Lieberman, “Concurrent Object Oriented Programming in Act 1”, in *Object Oriented Concurrent Programming*, Aki Yonezawa and Mario Tokoro, Eds. MIT Press, Cambridge, Massachusetts, 1987

[Saraswat 93] Vijay A. Saraswat, *Concurrent constraint programming languages*, Doctoral Dissertation Award and Logic Programming Series, The MIT Press, 1993.