Metaphor Design

Case Study of an Animated Programming Environment

This paper was published in the proceedings of the 1995 Computer Game Developer Conference
April, 1995, Santa Clara, California

Ken Kahn
Animated Programs
1748 Monticello Road
San Mateo, CA 94402
Kahn@CSLI.Stanford.edu

Abstract

There are many examples of bad use of metaphors in interfaces, for example, requiring users to put an icon for the floppy drive into the trash can icon to eject a floppy on a Macintosh. Good designs should use metaphors predictably and consistently; for entertainment software the metaphor elements should also be fun.

I discuss the history and rationale of the metaphor design of my ToonTalk™ system -- an animated programming environment for kids. For example, I needed to provide the user with the ability to copy, remove, restore, grow and shrink items. The first design used three different colored magic wands. The wands were further distinguished by displaying an abstract animation of their function. Preliminary informal user studies showed that this scheme was non-obvious and users got the wands confused despite the animation. Because of this I replaced the size changing wand with a bike pump and the removing and restoring wand by a hand-held vacuum. Other examples of ToonTalk metaphors discussed are the use of carrier pigeons for communication, cities on the back of pictures for sprite programming, and a mouse with a big hammer for arithmetic.

What's the problem?

The idea that the interface of a computer application could be based on metaphors goes back more than twenty years. Such metaphors provide mappings from abstract computational entities and actions to something concrete. Interface software maintains virtual analogs of these concrete objects. For example, in the Mac Finder the computational notion of directories is mapped to folders and the act of deleting a file is mapped to the act of dropping an icon on a garbage can icon. However, most games don't need metaphor since they aren't concerned with giving the user access to computational entities. The point of most games is to provide a virtual world for the user's entertainment, not to get something done in the computational realm.

There are some exceptions, for example, SimCity 2000 provides a magnifying glass for zooming in and out of one's city. Carmen San Diego provides the user with a notebook for storing facts. The SimCity magnifying glass is a clear case of metaphor since the mayor of a city doesn't pull out a magnifying glass to see where in the city a fire is. But the Carmen San Diego notebook is arguably not a metaphor since as a detective one typically has a notebook for note taking.

If we broaden our horizons to entertainment software then metaphor is much more prevalent. Consider KidPix and all of its tools. The user is trying to make a picture and many of the tools borrow heavily from the metaphor of an artist with paint and brushes. Some are a bit more far out: click on an eraser and one gets a choice of many ways of getting rid of things including using dynamite. A game making system like Klik & Play is full of metaphors from movie making, animation, painting, and spreadsheets.

If you need to design or choose a metaphor for your entertainment software, you should try to meet these goals (presented in increasing order of importance):

1. Thematically consistent
2. Aesthetic
3. Fun
4. Easy to learn and use
5. Matches semantics well

Thematically consistent. As soon as more than one mapping is needed, the issue of consistency arises. Many desktop metaphors for file deletion differ from Apple's garbage can: they range from SGI's dumpsters to NeXT's black holes. Fear of law suits may have motivated this, but such things clearly do not belong on a desktop. (Even Apple's trash can would be more appropriately a waste basket.)

Aesthetic. Some metaphors offer more opportunity for aesthetics than others. A desktop is dull and mundane while a room metaphor (such as Magic Cap or Microsoft's Bob) provides more opportunities for aesthetics.

Fun. This is ignored by most metaphor designers who are focused on office workers rather than home users. For example, a science fiction theme, perhaps incorporating transporters and tricorders, can be much more enjoyable than a desktop theme.

Easy to learn and use. Metaphors that depend upon familiar items will be easy to learn if there is a good mapping from the properties of computational abstractions to the properties of the familiar items. A waste basket is easy to learn to use -- just drop things in it. A user is likely to figure it out by exploration and experimentation. But few users guess that to eject a floppy on a Mac, one drops the floppy's icon in the trash can. Familiar doesn't necessarily imply real-world. Various well-known fantasy, science fiction or fairy tale worlds can work well. It is worth remembering that ease of learning often does not lead to ease of use. Many users find it easier to select a file and hit the "delete" key than to drag an icon of that file over to a waste basket.

Matches semantics well. This is the biggest challenge for metaphor design. For example, the desktop metaphor for a file system is a poor match. Although the metaphor adequately represents creating, moving, and destroying files and directories, how about when the user moves something from a floppy to the hard drive? Consistency would argue that the file is deleted from the floppy. This is not what the Mac Finder or Microsoft Windows File Manager does. (The designers probably chose "copy" semantics over "move" because copying is more frequently desired and easier to undo.) An even worse mismatch arises when the file system has more capabilities. Unix has symbolic links which are very useful for aliasing files, but the semantics of this just doesn't map well to any desktop.

A Brief Introduction to ToonTalk™

The rest of this paper examines these issues of metaphor design by presenting a case study of ToonTalk, an interactive animated programming environment for kids. ToonTalk presents its users with a working metaphor for computation. The underlying computation model is from concurrent constraint programming which has its roots in logic programming. But the kids don't know or care that they are learning about unification-based process synchronization or recursive data structures. They understand these things in terms of a concrete metaphor of a city full of houses, where carrier pigeons carry items to their nests, where construction crews build new houses, where robots can be trained to put things in and take things out of boxes, and the like.

The following table shows the mapping of computational abstractions to concrete analogs underlying ToonTalk:

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)	cubby holes
comparison tests	scales
agent spawning	loaded trucks
agent termination	bombs
constants	number pads, text pads, pictures
channel transmit capabilities	birds
channel receive capabilities	nest
program storage	notebooks

Rather than present the design rationale of each of these mappings, the rest of this paper focuses on a few examples and their history.

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.

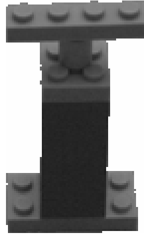
Copy Wand　　Expand/Shrink Wand　　Remove/Restore Wand

Figure 1 -- First version: 3 Magic Wands (color original)

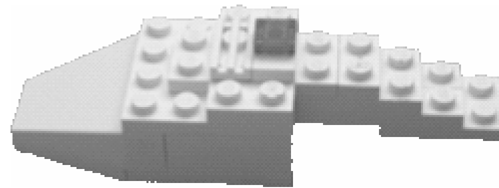
Preliminary informal user studies showed that this scheme was non-obvious and users got the wands confused despite the colors and animation. In order to make the functionality of each tool easier for users to guess and to make the tools easier to distinguish from each other, I replaced the size changing 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 2 -- 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 3 -- Bike pump and vacuum as animated characters

ToonTalk Communication

A completely different example from the design of ToonTalk is the metaphor design for communication. The initial design 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 birds and nests 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 Sprites

The computational model is accessible by the kids as concrete analogs. A hard design question was what about the sprite library that we wanted to make available to the child programmers. A computer scientist would argue that the sprite library, like all external services, should be available via message passing. Some birds know about the sprite library, others the music primitives, and so on. If you give such a bird a message it flies off, the action is performed somehow, and the bird returns. This is adequate but not easy to learn or use. The library becomes an API (application programmer interface) that must be mastered.

I first considered generalizing ToonTalk houses to mobile homes or trucks. A house in ToonTalk is a place where a thread of computation happens -- why not allow the house to move and to change its appearance? It could have a dashboard for controlling its position, size, and appearance and for sensors to deal with collisions. Unfortunately this only maps *some* of the computational abilities to the concrete analog. The sprite library supports 2 and 1/2 D graphics -- how could the notion of layers work? Are these trucks driving in a glass multi-story parking lot? And how do new sprites suddenly appear somewhere? And what about sprites that just pass through each other?

I also considered a design inspired by the "painters algorithm" from computer graphics, which paints the furthest object first. Maybe a sprite library is best thought of as an artist (or group of artists) that paints images. And they paint quick enough to generate animation. Users can tell the artist to paint something at a particular location with a particular size and so on. However, collision detection and reporting don't fit too well here. Also, the details of the communication with the artists seemed likely to be hard for children to learn.

The third alternative I considered was inspired by computer-controlled Lego™ blocks. These Lego systems, in addition to the usual blocks, have lights, motors, and sensors under computer control. The idea was to provide the functionality of a sprite library via a simulation of computer-controlled Legos. Sensors can be set up to signal collisions,

motors can be attached for moving and turning, and so on. While this scheme was appealing, it too was only able to capture some of the functionality that a good sprite library should provide. The biggest problem was finding a good fit for the dynamic creation, destruction or alteration of sprites.

Another scheme included x-ray cameras and associated monitors that one can enter to alter the internal workings of a sprite. This lead to the current sprite facility in ToonTalk in which pictures or animation cycles are just another "data type" in the world. Users can copy them with the magic wand, alter their size with the bike pump, and so on. Users can also flip them over. On the back side is a notebook full of remote controls for that sprite. Most of them act as a two-way control: change the sprite and the corresponding numbers and displays change, change a remote control number and the sprite changes accordingly. There are remote controls for position, size, speed, appearance, and collision detection. Furthermore, the back of a sprite is the place for its local behavior and state to be kept (in the form of robots and boxes). 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 ...).

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. All together 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 2+2

Wrapping Up

While it is useful to try to extract general principles or criteria for good metaphor design, at this stage of the field's development consideration of case studies is probably more productive. One can learn a lot by studying others' designs but even more from studying the history of the development of those designs. I learned a lot in the process of evolving the design from one based upon a postal delivery system for communication, with three kinds of magic wands for tools, and vehicles for sprites to one based on birds and their nests, a variety of tools which come to life, and cities on the back of pictures. I hope that by writing this paper I have shared some of that learning with you.

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. And thanks to Matty Kingsland for her many editorial comments.

More information. There was a paper in last year's Computer Game Developers Conference about ToonTalk. A more up-to-date version was written for educators and it will appear in the Proceedings of the National Educational Computing Conference, June 1995, Baltimore MD. This paper is also available from anonymous ftp from csli.stanford.edu. See the ToonTalk entry in /pub/Preprints/INDEX for more information.