On the Generality of Boxer Principles of Spatial Metaphor and Naive Realism

Jeremy Roschelle
Digital Promise
jroschelle@digitalpromise.org

Abstract

Reflecting the principled design of programming languages for the express purposes of being *understandability* and *useful*, the Boxer programming language rested on dual principles of spatial metaphor and naive realism, as well as a broader principle of "reconstructability." These principles are over 35 years old and the language itself is mostly a museum piece today. Are they relevant today? In a prelude, I discuss how and why these principles were incorporated into the design of Boxer's graphics system, a project I completed as an undergraduate student. Then in an interlude, I consider how these principles played out in non-programming language contexts in my work as a learning technology researcher. Finally, the postlude considers how these principles might play out today and into the future

Introduction

Why can't I easily control my router? Why is it so hard to program my smart house to do more than trivial "its sunset; turn light on" routines? Why can my music software programmatically manipulate the notes to add some swing, but I cannot inspect what it doing to the rhythm and I can't change how it swings if I want to? Why is my laptop so full of different automation capacities—AppleScript, Automator, Bash, "Shortcuts," Google AppScript—and I rarely use any of them despite my MIT computer science degree? It's well over 35 years since it was clear to me that I was capable of writing little bits of code to do useful things—as long as the devices met me halfway. Yet despite all the intervening advances in technology, my life is full of highly-engineered user interfaces that still fail to meet my expectations.

My entry to this workshop on the history of the Boxer language is a playful reflection. The latin word "lude" means playful and is a common English suffix. Alluding to a central joy of Boxer—its playfulness—the Prelude section will reflect on my own early history with Boxer, its principles and its graphic system. These importantly address how to design a programmable system that is learnable and understandable. Then an Interlude section will discuss how similar principles were included and influential in my subsequent work as a learning technology researcher. Finally, a Postlude section will comment on the ludicrous state of affairs

in my computationally-adept house that eludes my understanding, perhaps with brief allusion to my early days spent with Music Logo and secluded attempts to master today's music composition software. I'll conclude with an attempt at a lucid thought about ludic (playful) computer interfaces.

Prelude

I went to MIT as an undergraduate because I liked to make things as a child by taking toys apart and putting them together in new configurations. Much to my sister's chagrin, this included all manners of little alarms and surprises that might be triggered by her movements, say, opening a closet door. An engineering school seemed right as I was very clear that although I loved making music, making music would never pay the bills.

At MIT, I thought I'd be a physicist because the big ideas were so exciting to me. Combined with another fact, this may explain how I ended up in the orbit of another physicist, Andy diSessa. My career as a physicist ended quickly, yet my time with Andy continued on for many years. Although I fondly remember a few classic MIT physics lectures, such as Walter Lewin on rainbows, as best I could tell the physics department was hellbent against learning. Each day, the sliding blackboards of lecture hall 26.100 were meticulously filled with equations from the first board, top left, to the ninth and last board, bottom right. An hour of lecture and not a moment devoted to making sense of all the formalism. This is exactly what motivated me to become a researcher in learning technology—I knew there had to be a better way to teach those who were enthralled by STEM concepts but not in love with equation manipulation.

I landed in computer science because Hal Abelson and Gerry Sussman excelled in explaining big ideas in the legendary Scheme course, 6.001, which at the time was supported by a paper binder of lecture notes, yet to become a book (Abelson & Sussman, 1985). Functional programming, lambda expressions, closures—oh my!—I thrilled to learn that computer science was more than glorified hacking. Hacking, of course, was also in abundance at MIT. Indeed students who were up a bit too late hacked *buildings* well before we had the possibility of smart homes. In the differences between computer science and backing, I learned a first principle: "side effects" or a program that changed some external state or relied on external state was the antithesis of functional programming.

Later, my musical interests attracted me to Building 20, a somewhat ramshackle quonset hut, because there were very nice practice pianos there. And down the hall was Jeanne Bamberger, doing wonderful things with Music Logo (Bamberger, 1991). What Jeanne taught with Music Logo blew my mind. Constructivist music appreciation! I had thought of music as *transmitted* from the composer through the instructions to the performer to the ears of the passive listener. Jeanne turned this completely upside down, enlightening me as to how music was instead *constructed*. Constructed by the listener. Constructed by the performer. And only loosely guided (especially with regard to its emotional impact) by the composer. The instrument of my constructivist re-birth was a programming language that represented music as lists which could be inspected, played by the Apple II, and playfully transformed either by typing or by writing little snippets of Logo code. For a term paper, I analyzed Beethoven's Fifth by seeing how far I could get with only inputting four notes directly ("ba ba ba bum") and producing the rest of the first movement by writing code to transform those four notes. By *constructing* my own 5th facsimile I not only came to better understand Beethoven's composition as playful transformation of motif, but also where he artfully inserted non-motif notes and phrases to make it all work (Roschelle, 1982).

Hal Abelson happened to be my undergraduate advisor and Jeanne was my mentor, and thus it should be no surprise that I eventually met Andy diSessa. Andy cared about those two things that were most driving me. He cared about Big Ideas. And he cared about how people could Learn Big Ideas. When it came time to do a senior thesis, Hal and Andy observed that Boxer at the time had no graphics and likely thought to themselves: "well, let's stick an undergraduate on it, maybe we'll get lucky," So I was asked to make the graphics for Boxer.

Of course, Andy and Hal wanted turtle graphics in Boxer. They had recently published another mindblowing treatise, Turtle Geometry (Abelson & diSessa, 1981), in which they demonstrated that very abstract and high level mathematical concepts could be understood playfully by investigating the travels of the lowly turtle. And herein, I ran into a dilemma. Turtle graphics was *all side effects*. The little triangular drawing tool on the screen, called a "turtle," was the anti-hero to which every good Scheme-devotee trained in the art of functional programming must take offense. In Logo, you write code and the turtle does things; it is state-dependent, but

only reluctantly reveals its state to those who inquire via mysterious incantations. Is the turtle hiding or showing now? Is its pen up or down? I don't know, do you?

Boxer, of course, was made of boxes. Therein I found a solution (Roschelle, 1985). The first part was obvious: I would create a graphics box in which graphics could be drawn by a turtle (renamed "sprite"). This merely partitioned the screen to create a place for graphics. Further, I wanted to reflect the containment of the turtle within the graphics box in my design. However, I did not want to mess up the nice clear drawing surface by littering it with code. This tension led me to a new invention for Boxer, which was the idea of a box that could be "flipped" to reveal its backside. On the backside of a graphics box, I made a box for the turtle.

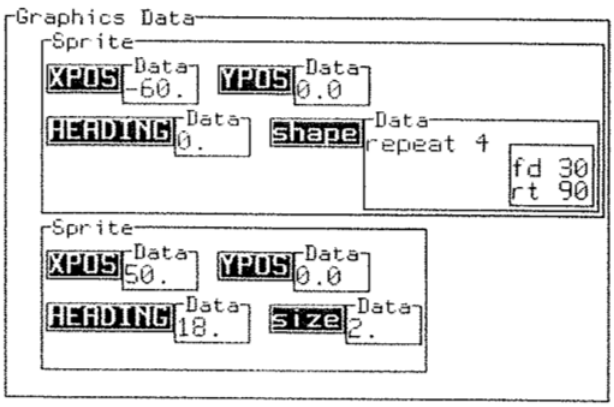


Figure 3-6: A Graphics Data Box with Two Sprite Boxes

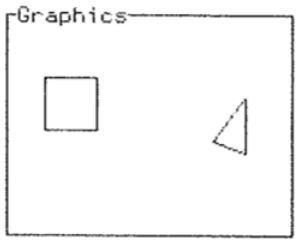


Figure 3-7: Toggling to a Graphics Box

This design decision had two very positive consequences. First, because the graphics and the turtle were now boxes, I could use the existing box-naming convention to give them names. I could call my turtle "fred" and send it commands by writing "tell Fred [Square 40]" leading Fred to draw a square of size 40. Second, I could put more boxes inside of Fred to correspond to the turtle's state. A turtle has position, and so there were x-position and y-position boxes for every turtle's coordinates. Likewise, a turtle has a direction it points, and thus there was a heading box. Likewise, there were boxes for pen state (up or down) and whether the turtle was visible (hidden or shown). These boxes were specially recognized by my code so that the correspondence between the backside and frontside were maintained. If you changed a turtle's orientation with a command like "left 65," you could see *both* the turtle spin to a new orientation and see its numerical heading change in the heading box. If you typed a new number in the heading box, the turtle would likewise pivot to the new heading.

Hal and Gerry used the term "syntactic sugar" for programming conventions that avoid really awkward code, and I sprinkled two kinds of sugar in my design. I made up a new Boxer

convention of “transparent” boxes, which grouped a portion of the screen but did not make a new scope for the variables within. Hence, instead of awkwarding writing “tell graphics [tell fred [right 65]],” one could just tell Fred directly. I also took advantage of portals, which allowed a box to be shown in more than one place. In my design, one could make a portal to a particular turtle variable of interest and thus give that variable a new name in a different place. For example, when making two turtles dance, one might want to easily access both their headings without lots of tell statements. This could be accomplished by two portals, one called “Fred-Heading” and the other called “Beth-Heading,” each which would allow access to get or set the turtle’s orientation from within a program. (Another little bit of sugar was that one did not have to show ALL a turtle’s state variables; they could be removed or reinserted as needed; this avoided clutter).

There was a final element that bugged me about my early design. The drawings of the turtle on the graphics box were still side effects; one could only clear all the previous trails of the turtle pen, but could not programmatically move a drawing to the left or the right. Also, the turtle was only sort of an object. It was an object from a programming standpoint, but users could not click on the turtle, for example, to pick it up and move it or to cause it to do something. Boxes to the rescue!

Around this time, I was programming on a LISP machine in the corridor of the AI lab, because the way an undergraduate student could gain access to a machine was that the terminals were left in the hall. Seymour Papert had a photo shoot and apparently needed to be captured working with a student. Apparently I fit the bill and thereby secured my 5 minutes of Warholian fame in which I made my turtles dance in the presence of Seymour Papert.

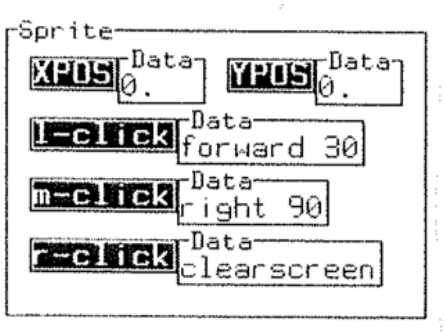


Figure 3-11: A Sprite Box with Click Boxes

Back to boxes. I decided that the turtle’s own shape did not have to be a triangle, but rather by creating a “shape” box inside the turtle, one could provide a program that would draw the turtle. This enabled turtles to look like arbitrary drawings (and even text), and thereafter moving a turtle on the screen could translate an entire drawing to a different portion of the screen. Another kind of box was a “click” box, and when the user clicked on a turtle’s shape on the graphics side of the box, the code in the backside would be executed. Thus the turtle was now a real object both in the code and in the user interface.

In 1986, a year after I left MIT, Andy and Hal published a nice piece that explained the principles which we had been talking about and I had implemented (diSessa & Abelson, 1986). The graphics design was consistent with Boxer's design as a *reconstructable medium.* That is, if you come upon a graphics box that does something you like, you can flip it over to see all the code responsible for that behavior. In the paper, the authors give an example of a simple rocket-launch video game in which one can flip over the box to see what makes the rocket move. Of course, all the code is also editable, so you could copy the box and edit it to do what you want. Consistent with *spatial metaphor*, the graphics design made use of containment so that turtle's state variables were inside turtles and so that turtles were inside graphics boxes. The continuation of the spatial metaphor allowed for consistent ways to name and "tell" commands to turtles, to support multiple turtles in one graphics box, to use portals to access variables, and more. The idea of "flipping" a box allowed for the relationship between a new user interface (like the rocket game) and its code to be expressed spatially. Finally, *naive realism* was present in how a box that looked like a turtle's heading or position acted as if it actually directly controlled the turtle's visible orientation and location.

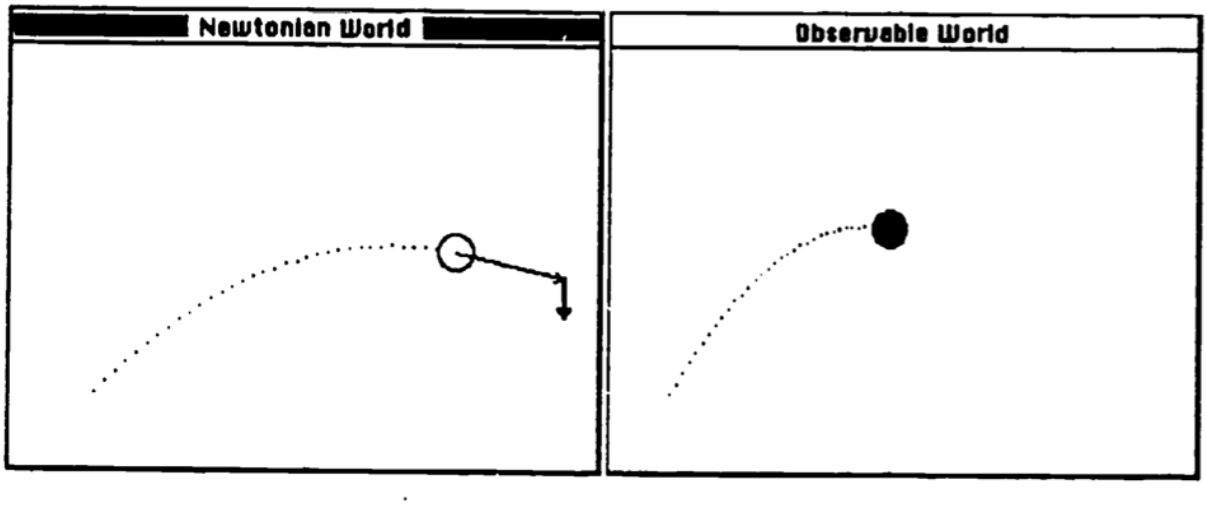
Interlude

I left MIT for the wild west of Berkeley, following Andy diSessa's own move to become a professor in the School of Education. In graduate school, I did not continue with Boxer, but instead returned to my original motivation: why was physics so hard to learn and how could technology make it easier?

At Berkeley, I became interested in theories of mental models (Johnson-Laird, 1983) and how they support learning (Gentner & Stevens, 2014) as well as ideas about visual representations to support learning (Kaput, 1992) and qualitative reasoning (Bobrow, 1984). These learning principles strongly relate to the Boxer design principles of naive realism and spatial metaphor. I was thrilled to read about a meeting of Einstein and Piaget (Miller, 1992), in which they discussed how playful exploration of space was both intrinsic to child development and Einstein's own construct of special relativity. It was not just spatial metaphor and naive realism, but also a constructive learning-by-doing approach that invited the learner to be playful. In a ludic turn, I realized play was fundamental to learning, and that human cultures have always designed play environments for their young. But what kinds of principles allow playful learners to learn some of the difficult math and science concepts we ask them to learn today?

In my dissertation work, I continued with the Boxer principles in the design of the "Envisioning Machine," software which supported high school learning about velocity and acceleration

(Roschelle, 1991). In the Envisioning Machine, students could directly manipulate vectors (using a mouse to change the direction and length of arrows) that represented velocity and acceleration. They would use this ability to model an animated motion in one screen called “the Observable World” using the animated vectors in another screen called “the Newtonian World.” I discovered that students learned about velocity and acceleration because these spatial visualizations enabled them to bring familiar metaphors like “pulling” and “stretching” to bear (Roschelle, 1991); these metaphors were also termed “p-prims” (1983) and linked to qualitative reasoning. I also found that spatial metaphors and naive realism were good for supporting collaborative learning; two students could readily use the pragmatics of conversation to together construct shared meanings for Newtonian vectors (Roschelle, 1992). This happened as students played a game (which scientists would call modeling) in which they tried to make the Newtonian World do the same thing they they saw in the Observable World.



I did this research by videotaping students as they collaborated with the Envisioning Machine and I needed a way to ease the burden of analyzing the videotapes. Spatial metaphor and naive realism again came into play as I developed two video analysis tools, VideoNoter and CVideo (Roschelle, Pea & Trigg, 1990). Each used a timeline to allow spatial control of the motion of the video transport, and then supported annotations placed in time alongside the timeline. One could both see where the video was currently playing (while watching relevant annotations) or drag the time indicator to a new location (e.g. near a target annotation) and the video would rewind or fast-forward to play from that location.

I also taught a discussion section of the Berkeley version of 6.001 (using the Abelson & Sussman, 1985 book). In so doing, I drew Boxer-like diagrams to explain many of the scheme concepts related to variables. The students responded particularly well to using boxes to

explain environments and closures, programming language features which otherwise are very abstract to understand.

Later, I was involved in development of SimCalc, which was a tool to support student learning about rates of change, a topic in Calculus (differentiation and integration) but also important to middle school mathematics (proportional reasoning and linear function). SimCalc allowed students to directly manipulate slopes in graphs (dragging the slope with their mouse) and to see resulting motions, for example, of a soccer player moving along a field. Students have a lot of trouble at first separating rates from totals. For example, in the common phase “the economy is down” do we mean that Gross Domestic Product is shrinking (really bad) or only that the rate of economic growth is slowing (not so bad)? This kind of confusion between a rate and amount often leaves a student who is learning about rate in the dust.

Figure: Two students using SimCalc

Whereas most teaching in middle school mathematics relies heavily on how to manipulate algebraic expressions, we found that introducing rate and proportionality via manipulation of spatial, visual representations was more successful. In a classic exercise that is very commonly used as a diagnostic in national or international math tests, students who first played with SimCalc were later able to distinguish (a) a point of intersection between two graphs (being at the same position) from (b) parallel slopes of two graphs (moving at the same speed). Later a team built 7th and 8th grade curriculum units with SimCalc, and we conducted a randomized controlled trial across schools throughout Texas. We found greater learning for students who

used SimCalc curriculum units (Roschelle et al, 2010), a finding which supports the power of Boxer principles of naive realism and spatial metaphor.

Although none of the Envisioning Machine, CVideo, nor SimCalc were written in Boxer, the principles carried forward to support learning of difficult math and science topics.. Indeed, looking back now at the diSessa and Abelson (1986) article, I realize I was chasing the same broad problem. Their writing about programming languages complained that the designers of programming languages were applying criteria — formal simplicity, efficiency, rigor, uniformity — that were great for experts, but inappropriate for learners. Similarly, instructors in mathematics and physics at the time tended to emphasize algebraic forms which are concise, efficient, rigorous and uniform, but terrible for providing conceptual insight to learners. Those 9 sliding blackboards in my undergraduate physics class in 26.100 were filled with algebra that was not only hard for ME to learn, but really hard for most anyone to learn. And 26.100 is not just at MIT, but its also what every middle school mathematics classroom is like. And we wonder why students are turned off by math in middle school!

The 1986 article instead called for a focus on understandability, usefulness to do things the learner wants to do, and tightly integrated designs for user interaction with graphical visualizations. This call to action applies to more than Boxer; it also to what I was finding worked for learners of physics (the Envisioning Machine) and mathematics (SimCalc). And by conducting both rigorous qualitative and quantitative research, I had found that the principles really worked for mathematics and physics learners, especially when coupled with enabling playful engagement in problem solving.

Coda

During this time, I also decided to eat my own dog food by now trying to learn something that had eluded me back at MIT, quantum mechanics. This time I would try to learn this difficult topic with spatial metaphor and naive realism. Despite the awful instruction I had experienced, some MIT instructors excel at teaching and one of the most famous in this regard was Richard Feyman. As part of an online distance-learning course, I read his book QED (Feynman, 1985). As I learned about photons, I was shocked to realize that quantum mechanics could be understood through *geometry* — space — and not only through *formal algebraic manipulations*. It occurred to me that The Geometer's Sketchpad, a constructive environment for building interactive, dynamic geometric drawings, could be used to build my own models of QED. With the help of Sketchpad's inventor, Nick Jackiw, I built geometric diagrams in which I could draw the many possible paths of a photon from a source to a destination. Each path supported the travel of a spinning clock hand (unit vector), and the intensity of the light at a destination was calculated by the magnitude of vector sum of all such spinning clock hands. The vector sum appears as a curling object which converges pretty quickly to the final intensity prediction, even with just tracing a handful of possible photon paths. I could interactively play with my diagrams and see the resulting predictions for the intensity of light and thereby understand many intriguing phenomena with interference patterns and the like. *I was able to understand Feyman's explanations of quantum mechanics!*

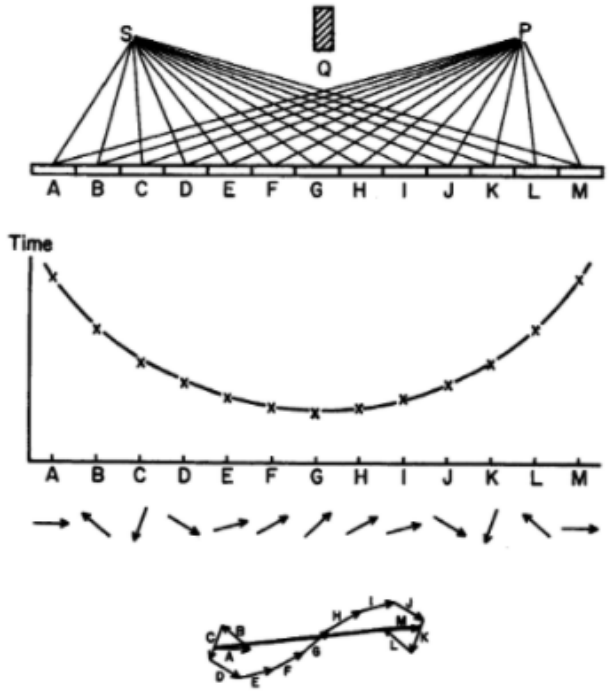


FIGURE 24. *Each path the light could go (in this simplified situation) is shown at the top, with a point on the graph below it showing the time it takes a photon to go from the source to that point on the mirror, and then to the photomultiplier. Below the graph is the direction of each arrow, and at the bottom is the result of adding all the arrows. It is evident that the major contribution to the final arrow's length is made by arrows E through I, whose directions are nearly the same because the timing of their paths is nearly the same. This also happens to be where the total time is least. It is therefore approximately right to say that light goes where the time is least.*

So in the end, playful engagement with spatial metaphor and naive realism — Boxer's principles — enabled me to come to an intuitive, constructive understanding of the very concepts which had caused me to abandon undergraduate physics.

Postlude

The Boxer principles fit with other cognitive and social principles, such as mental models, collaborative learning, and constructivism. Today, one obvious application is supporting students to learn computational thinking and computer science. Such work is being ably carried forward in environments like Scratch and AppInventor and by new robotics toolkits that recall the mechanical turtles in 1960s Logo. As part of my work, I visit schools all over the country. I've been amazed to see very young children such as first graders able to explain the code they built to me. Obviously the principles that Logo and Boxer brought us are still very important to a central dilemma of 21st century education: how to teach all students about computer science.

And yet I wonder if the use of these principles shouldn't be for more than for children and for more than computer science learning. I wonder if the success of graphic user interfaces in our everyday lives is reaching a limit, and whether ordinary people might benefit from approaches that are enabled by allowing people to write code that augments the machines they use everyday.

Clearly Apple thinks so; it has provided my MacOS with at least four different scripting approaches: AppleScript, Automator, Shortcuts, and the shell. While powerful and "simple," I find these are all terrible things to use and it's hard for me to find any fellow Mac users who use them. Each scripting language simplifies the syntax of programming but does nothing for understandability. And I wonder why. The MacOS has a rich naive realism metaphor of a desktop with folders containing folders (like boxes in Boxer) and you can even attach "Folder Actions" to folders to cause them to do something when their contents change. AppleScript, like Boxer, uses "tell" to navigate the objects in a hierarchy. Yet in my experience, it is so hard to make sense of what an AppleScript is doing and so mysterious when it fails. Simplifying programming syntax without an accompanying design-for-understandability appears clearly insufficient for people like me. I can't "play" with any of these approaches; if I have to use them for a project, I now know that I'm going to have hours of painful debugging awaiting me.

Likewise, I've played with many devices in my smart home. The smart home systems all use a quasi-spatial metaphor, for example, devices are in rooms and zones contained in a home. And yet none of the smart home hubs I own (considered among the industry's best) present a visual spatial metaphor to help me understand where my things are, or to locate the "routines" (programmability) adjacent to the devices being programmed. No wonder that I can get as far as "when its sunset, turn some lights on" but none of my programmable smart home devices has more than a handful of user routines in it. Lacking the spatial metaphor, I get overwhelmed

with the complexity of a long list of rules. And trying to fit my automation ideas into someone else's narrow rule formalisms quickly gets frustrating.

I also long for Music Logo. Musical production software is amazingly powerful and yet I find learning it to be too much of a career for the amateur. All the software organizes music in timelines, and the notes you you play shows up as boxes that you can move around and manipulate. And yet the only way I can see what is in the box is through the software's chosen graphical interface, and this is even though the notes are stored via Midi as numbers in lists—the very same numbers in lists that I was able to enjoyably manipulate in Music Logo on an Apple II. 40 years later and it was easier to programmatically manipulate music on an Apple II than a modern Mac.

Although it obviously appears ludicrous to many in the software industry, I wonder if were aren't up against some complexity barriers in everyday products that could be well-addressed by re-organizing the products for understandability and for giving the user the ability to write small bits of code, presented through a spatial metaphor and naive realism, and offered with support for learning by play. Enabling everyday users to play with code they can learn and reconstruct, when supported by artful design, isn't ludicrous. Indeed, to conclude, I believe what Boxer tells us is this: the ludic path is the lucid path to the future.

References

Abelson, H & Sussman, G, (1981). Turtle geometry. MIT Press. ISBN: 0-262-51037-5

Abelson, H & Sussman, G, (1985). Structure and interpretation of computer programs. MIT Press. ISBN: 0-262-51087-1

Bamberger, J.S. (1991). The mind behind the musical ear. Harvard University Press. ISBN: 0-674-57607-1

Bobrow, D. G. (1984). Qualitative reasoning about physical systems: an introduction. Artificial intelligence, 24(1-3), 1-5.

DiSessa, A. (1983). Phenomenology and the evolution of intuition. In Gentner & Stevens (Eds). Mental models.

diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. Communications of the ACM, 29(9), 859-868.

Feynman, R. (1985). QED: The Strange Theory of Light and Matter. Princeton University Press. ISBN: 9780691083889

Gentner, D., & Stevens, A. L. (Eds.). (2014). Mental models. Psychology Press.

Johnson-Laird, P. N. (1983). Mental models: Towards a cognitive science of language, inference, and consciousness (No. 6). Harvard University Press.

Kaput, J. (1992). Technology and mathematics education. In Handbook of research on mathematics teaching and learning

Miller, A. (1992). Imagery and intuition in creative scientific thinking. Creative people at work: Twelve cognitive case studies

Roschelle, J. (1982). Music, Martians and Computers. Unpublished term paper.

Roschelle, J. (1985). The design of a graphics subsystem for Boxer. Unpublished thesis.

Roschelle, J. (1991). Students' construction of qualitative physics knowledge: Learning about velocity and acceleration in a computer microworld. Unpublished doctoral dissertation, University of California, Berkeley

Roschelle, J. (1992). Learning by collaborating: Convergent conceptual change. Journal of the Learning Sciences, 2(3), 235-276. https://doi.org/10.1207/s15327809jls0203_1

Roschelle, J., Pea, R., & Trigg, R. (1990). VideoNoter: A tool for exploratory video analysis (Report #90-0021). Palo Alto, CA: Institute for Research on Learning

Roschelle, J., Shechtman, N., Tatar, D., Hegedus, S., Hopkins, B., Empson, S., Knudsen, J. & Gallagher, L. (2010). Integration of technology, curriculum, and professional development for advancing middle school mathematics: Three large-scale studies. American Educational Research Journal, 47(4), 833-878 https://doi.org/10.3102/0002831210367426

Acknowledgements

Thank you to my mentors in Boxer, Andy diSessa, Hal Abelson and Jeanne Bamberger, and to Antranig Bosman for prompting these reflections.