Give Them the World: From Sandbox to Shoreline

Figure 1: Introduce youth to the computational shoreline - still a place for play, imagination, and exploration, but irrationally beautiful, enormously broad, and open to an ocean of information and danger.

Josh Sheldon
Hal Abelson
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
jsheldon@mit.edu
hal@mit.edu

Mark Sherman
University of Massachusetts Lowell
Lowell, MA 01854
msherman@cs.uml.edu

Abstract
This paper explores the idea that adults, out of fear for or of youth power, often fail to provide authentic learning environments for computing. We propose two principles that might inform the design of these authentic environments, and then consider several designed environments with these principles in mind. Finally, we consider the lineages of some computing environments designed for learning and what lessons there are to learn from such lineages in this realm

Author Keywords
Computing education; computing environments; youth empowerment; authentic inquiry; constructionism.

ACM Classification Keywords
K.3.2 Computer and Information Science Education: Computer science education.

Introduction
"They might access inappropriate content"

"They could hack something and get into real trouble."

Many adults both fear the raw power of youth and discount the ability of youth to make positive change. As a result, adults often seek to constrain the power of youth and limit their ability to make any change. This generational mistrust is a story as old as stories, may not be explicitly recognized, and is played out in many arenas, including, we posit, in computing education.

To emphasize this point, who, in working with schools, computers, and computing education has not heard statements similar to, if not more pernicious than the two that open the introduction? And so, the tools that are built for learning are often sandboxes. In fact, that term is actually used to describe the environments in which we often allow (or perhaps require) students to compute. These "safe" but limited environments set fences around what programmers can do - preventing them from wreaking havoc, but also preventing them from having real impact on the world.

Further, every person, be they child, adolescent or adult, knows that the interesting things are on the other side of the fence. By making the "real world" of

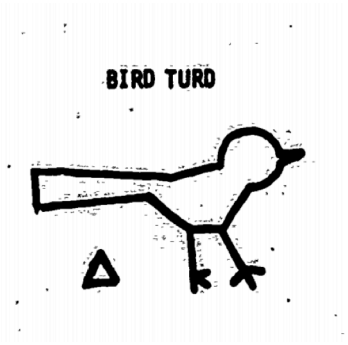

Figure 1: An example of the transgressive activity computer programming can afford and that was encouraged by Papert, from Papert & Solomon, 1971.

computing off limits, rather than teaching how to approach it with thought and responsibility, young people are tempted to enter it illicitly and without guidance.

In this sense, as in many others, young people often live exactly up to, or down to our expectations for them. So, the authors argue that as educators, we should strive to set high expectations, scaffold but not fence off environments that help youth meet or exceed those expectations, and empower those youth to compute and impact their world.

Rather than a sandbox, then, why not introduce youth to the computational shoreline - still a place for play, imagination, and exploration, but irationally beautiful, enormously broad, and open to an ocean of information. A setting off point for new lands and adventures. Yes, shorelines can be dangerous places - there will be rocks and cliffs, storms of all magnitudes, and sometimes even pirates (who may be recruiting). Isn't it better to learn to avoid and navigate these dangers side-by-side with experienced guides, though, than it is to pretend those with whose education we are entrusted will never encounter them?

Implications

What does this metaphor imply for the design of a computing environment? The environment should support authentic problem solving. Again, youth know the difference between set problems and true inquiry and exploration. They chafe against the former yet thrive when presented with opportunities to address the latter in developmentally appropriate ways.

Supporting authentic problem solving can be broken down into two implementation principles. First, ideally, the environment is designed to allow programmers to quickly wrestle with the core logic of the challenge they choose to tackle. Instead of forcing them to learn an arcane system of syntax, and construct high level abstractions, *the environment can quickly and easily provide high level abstractions that can be combined to perform powerful functionality* (Principle #1). Second, *the environment should allow programming software for the real computing ecosystem we use today* (Principle #2). In other words, it should be able to produce artifacts that make use of necessary computing affordances to perform the desired function (today, that often means using a mobile device), and take advantage of the unprecedented connectivity and availability of information that the Internet provides.

Designed Environments

These principles, whether explicit at the time or not, have been at the heart of our individual work and collaborative work, for years or even decades. Most recently, these principles have informed the design and implementation of MIT App Inventor, on which we all collaborate.

MIT App Inventor allows programmers to use a collection of predefined components to build mobile apps for Android devices. Some components are visible UI features of an app, and others expose functionality of the mobile devices the app will run on. In light of the high-level. They allow users to quickly perform what might otherwise be complex programming tasks. For example, one component allows quick access to information about the physical location of the device via the best sensor available. Another provides the ability

to interact with the Twitter API with very little configuration necessary. Note that these two examples bring us to two facets of Principle #2 mentioned above - taking advantage of necessary affordances of (mobile) computing and interacting with the Internet.

We contend that these principles should inform the design of other computing environments, and that in some cases they do, to differing degrees. We posit that an ecosystem of tools will end up producing better programming environments in the long run than a single tool can without healthy competition. There are, though, serious challenges to developing tools that embody these principles. Building a development environment from the ground up is a serious and costly undertaking. Further, the divide that exists between mobile operating systems presents a dilemma. By choosing either major operating system, you immediately divide the possible impact factor of the apps that will be produced. This flies in the face of the second half of Principle #1: allow programmers to build powerful programs. On the other hand, true cross-platform development without customization for each has yet to be demonstrated well, which violates the first half of the same Principle: that access to high level functionality be quick and relatively easy.

Learning From Forebears

It is also worth noting that there are and have been other examples of environments that seem to embody these principles, and that there are lessons to learn from those examples. Most notable, perhaps, are Logo and Scratch, though many others could be considered. Logo, which can be called the ancestor of Scratch, App Inventor, and many other programming environments, was built with the belief that youth should be able to program computers. At the time it was built, 1967, this was a revolutionary thought. However, Logo did enable young people to program computers. Subsequently, with the advent of personal computers in homes and schools, Logo grew to be used quite widely.

With that wide use came an interesting observation. While the initial developers of the language believed firmly in learning that would come to be called constructionism, Logo could be used in many ways. In some cases, the people facilitating student use stayed true to the constructionist origins of the tool. In others, however, teachers were much more didactic, going so far at times as to ask students to use Logo to perform calculations to solve arithmetic problems. As one might imagine, learners who were free to explore and generate their own questions, and were scaffolded in their quests to answer those questions with Logo, were highly motivated and often continued with their learning through programming. On the other hand, students who did math drills were much less motivated to continue (Liddy Neville, personal communication, March 27, 2014).

This is evidence to support an important point. It may seem obvious, but is still worth noting that motivation and learning depend not only the tool, but on how young people are encouraged (or sometimes instructed) to use it. We know that there are classrooms where App Inventor is used by students to duplicate patterns of blocks programs from a diagram into their own project by rote and with little thought or understanding. Though the evidence is still anecdotal, it is almost certain that these students are much less motivated to continue programming and may never see App Inventor or programming as a tool for solving real

problems. In contrast, there is strong evidence, supported by several years of evaluation research, that learners given the opportunity to solve real problems with App Inventor have a higher likelihood to continue programming. In fact, such learners are much more likely than the general population to later major in computer science in post-secondary studies. (Technovation, 2015)

It is also instructive to note that Logo gave rise to a staggering number of "offspring" programming environments. Different descendents had different features, were optimized for different types of problem-spaces, and did in fact make up an ecosystem of programming environments for learners. And it is certain that some of these descendants improved on the original features of Logo.

A few of the more widely used modern descendants include Scratch, Snap!, PocketCode, NetLogo, StarLogo (TNG & Nova), and App Inventor. As well, there are other members of the ecosystem that may not be directly descended from Logo, including Alice and variations upon it as well as Greenfoot and Bluejay, among others. Few, however have fully made the jump to programming for mobile computers. And so, there is room for healthy competition - a new or adapted set of tools, informed by these initial principles, to make the entire "population" of programming tools (for learners) stronger, to help refine the nuances of these principles and distill new ones.

Acknowledgements

We thank the many, many people who have informed this discussion, as students, peers, interviewees, and in many other roles. Eric Klopfer played a large role in the development of these ideas. Special thanks go to Lyn Turbak, Ralph Morelli, Dave Wolber, Fred Martin, Eni Mustafaraj, and others from the Mobile Computational Thinking research team. We also thank Ben Shapiro for pointing us toward the 1971 Papert and Solomon Memo. Finally, thanks are owed to early readers of this position paper, including Barbara Ericson, Daniel Wendel and Warren Buckleitner.

References

1. Seymour Papert & Cynthia Solomon. 1971. Twenty Things to Do With a Computer. *MIT Artificial Intelligence Memo*. Number 248.
2. Technovation. 2015. Technovation: Results. Retrieved March 31, 2015 from http://www.technovationchallenge.org/results/