Remote Opportunities: A Rethinking and Retooling

Yağız Onat Yazır, Katherine Gunion[1], Christopher Pearson,
Celina Gibbs, Anthony Estey, Steven Lonergan,
Yvonne Coady
University of Victoria
{onat, pearson, celinag, aestey, stevenl, ycoady}@cs.uvic.ca
University of British Columbia[1]
katgun@interchange.ubc.ca

Abstract

Introducing technology as a sustainable means of creating, connecting, and collaborating reveals the need to carefully consider subtle aspects of deployment strategies and support in remote regions. In order to comprehensively address both cultural and technical issues for educational infrastructure, we consider two elements to be key: (1) a staged deployment approach, involving both educators and community members, coupled with (2) uniquely designed collaborative Integrated Development Environments (IDEs) to aid constructivism.

This paper presents our current experience with these elements in the context of a pilot project for aboriginal communities on the west coast of British Columbia. Currently, these local communities have been working alongside our group for a staged deployment of programs throughout southern Vancouver Island. In our next phase we will be extending this to more remote regions in the north island and coastal regions. By building on a philosophy of Community-Driven Initiatives for Technology (C-DIT), we hope to secure community involvement in the development and testing of necessary tool support. These tools specifically target IDEs for the development of programming skills, and support our long term goal to help secondary and post-secondary level students appreciate both the process and the art of programming.

1. Introduction

The simple intuition behind the proposed philosophy of Community-Driven Initiatives for Technology (C-DIT) is that by planting initiatives that are both sensitive to community-context and driven primarily from within, we can improve long-term sustainability of the projects involved.

The traditional role of elders in aboriginal communities in terms of leadership is clear. However, not surprisingly, the youth in the community are generally more technically savvy. Hence, in the context of technology, there is actually an important inversion taking place. By working with community members to develop a program where youth are offered enriched programs involving computer science, and also serve as teaching-assistants for elders, we believe we can effectively harness this inversion and connect the two groups in a collaborative initiative. We believe that, where possible, educational initiatives in remote regions must be sensitively integrated with cultural issues such as aboriginal elder leadership.

Given that representation and educational infrastructure from within a community is necessary for sustainability, we propose a rethinking and retooling of remote educational infrastructure based on a simple philosophy of *Community-Driven Initiatives for Technology* (C-DIT). For example, targeting the youth and stimulating their inherent interest in technology through a staged deployment of programs deisgned to be sensitive to cultural contexts, we believe we can create a demand for more advanced educational offerings.

Of course, supporting these advanced offerings in contexts that will largely rely on remote technology brings with it a whole new set of challenges. Here we believe the right tool support to promote understanding of both high-level abstractions and low-level architecture is the key. We propose an IDE that allows for transparent transitioning between graphical languages, textual representations, and computational elements in order to allow students to explore the process of programming and the artifacts involved.

The remainder of this paper is organized as follows. Section 2 overviews deployment strategies, providing an example from our own experiences of short-term localized initiatives and motivating community-driven programs. A model for C-DIT is provided, along with a concrete example of

such an initiative. Section 3 follows with the motivation and design for IDE support that is both collaborative in terms of process, and transparent in terms of representation. Section 4 closes with a discussion of future work and summarizes our conclusions.

2. Deployment Strategies

Before exploring C-DIT and its associated tool support, we briefly consider our previous efforts in educational initiatives with youth in local aboriginal communities in order motivate our adoption of a community-driven approach. Though arguably successful, our deployment strategy was localized and short-term.

2.1. Localized Success

In the summer of 2007, we hosted a camp for a group of 15 aboriginal youths (ages 12-17) for three days spanning three weeks. These students came from a near-by local community, where computers in the afterschool program are scarce, and students are given computer-time mainly on a reward system. At this camp we introduced these youth to the world of programming through graphical languages and IDEs such as those used by LEGO Mindstorms[18] and Scratch[19]. The students were interested in the projects and intensely engaged in programming. They appeared to be focused, and though no formal study was performed, we anecdotally observed that there were many unanticipated challenges along the way that were enough to discourage students from completing the task.

Part of our goal for this project was to create a module that educators with no previous experience could use to deploy the same educational exercises to self-motivated learners. However, we concluded that this was impractical for the material we had developed. Accordingly, several experienced teachers all concluded that without positive reinforcement and encouragement, many of these students were not invested enough to persevere programming in the face of adversity. In particular, if the computers they were working on had any Internet access, the first sign of trouble was typically accompanied by abandonment of the task and surfing for a virtual social context such as email, chatrooms and most notably Bebo[3] for social networking.

From this experience we have reason to believe that these youth are vested as *consumers* of technology, but require encouragement and positive feedback if they are to become *producers* of technology. Though we believe opportunities to leverage remote education exist, we also believe they must be conducive to incremental progress.

These observations are not new, as for decades Interactive Radio Instruction (IRI) has been implemented to teach a variety of subjects to school children in countries such as Nicaragua, Thailand, Guatemala, and Kenya[8]. The IRI projects were often considered successful for two main reasons: (1) the integrated interaction with fellow students, and (2) a facilitator would be present in the room to focus the students and moderate question and answers. As a result, there is face-to-face interaction as well as an authority figure to help mediate and guide. These two factors were not part of our original vision for remote deployment, and are potentially only available in a limited capacity in most current tools.

2.2. Community-Driven Initiatives

After recognizing the short-term limitations of our technology camps, we have now adopted a philosophy of going into communities and allowing them to choose their preferred education. Not surprisingly, basic technical courses such as recovery from virus infection and establishing a network infrastructure ranked highly in terms of desirable offerings.

In order to ensure buy-in from the same group of youths that attended the camp, we designed a 20-hour workshop, *Beyond Bebo*, and focuses on skills that go beyond their current exposure to technology[32]. Topics include computer construction, programming, graphic and web design, slide show presentations, networking, and troubleshooting. It is not surprising that young consumers have only a minimal understanding of current Information and Communication Technologies (ICT) and its contribution to society. Several studies have shown this to be true in the general population[19, 13, 21]. The question remains however, as to whether or not this combination of topics establishes a community-driven demand for esoteric topics such as computer science when these regions are faced with more pressing and rudimentary needs. Assuming it does the next question is how to expose students to the fundamentals of computation without the infrastructure for sustained instruction.

3. Computation: A Wholistic View

Currently there is a plethora of programming languages targeting youth, such as Squeak[12], Scratch[19], Lego Mindstorms[18], Pico Crickets[27], EToys[15, 16] and Alice[1]. Figure 1 shows a few of these. In a variety of after school and summer activities, these environments are used as an anchor for technological education[31, 29]. Furthermore, these learning environments gain a special importance in economically disadvantaged communities with the emergence and support of the One Laptop per Child (OLPC) project[24].

Additionally, there are a number of attempts to apply these programming environments within preliminary computer science courses in post-secondary education[7, 28,

10]. Such courses use these environments to facilitate a smoother introduction to more traditional syntax laden textual environments. However, the transition from graphical to textual environments can be unexpectedly hard. Students tend to have difficulties getting accustomed to syntax and understanding the more formal treatment of object-oriented concepts[28]. This is particularly surprising when the concept of objects have been first and foremost in many of these environments.

This transition may become even more complicated if concurrency appears to be trivial in graphical programming environments designed to simultaneously animate objects. Mapping the equivalent support involves threads of execution, which typically does not get attention until the later years of post-secondary education. The irony is that modern architectures are hungry to exploit programs designed with concurrency in mind. In short, the transition from graphical to textual environments is a juncture where students may disengage because it is accompanied by disappointment when advanced concepts do not map well from one environment to the next.

A possible way of preventing this sudden exposure to complexity would be to make use of a common programming environment to integrate the abstract understanding in graphical environments with more technical representations of problem solving. IDEs such as Eclipse[9], where the user has access to various tools within the same medium as plug-ins, can substantially ease the transition by giving simultaneous access to both graphical and textual tools. Furthermore, the advantages gained from such integration can be maximized through the development of a plug-in that is designed specifically to expose mappings between graphical and textual environments in an incremental manner. We believe better integration of these equivalent textual representations with graphical environments can in fact be beneficial to students struggling to create a mental model of computation.

Equally important to accentuate in first year post-secondary education is the fact that code design, implementation and analysis are intensely social processes. Growing popularity of real-time collaboration within industrial strength IDEs to facilitate team work is evidenced by recent work such as Jazz[4]. At the system level, providing such collaboration can become particularly challenging in geographically isolated areas where wireless alternatives to classical wired connections need to be considered[17].

In the remainder of this section, we first provide background on well-known graphical programming environments. Next, we propose a platform for integrating graphical and textual languages as an incremental transformation, including further deep transformations into computational representations involving modern architectures. Finally, we focus on the necessity for and nature of collaboration in such an educational platform.

3.1. IDE Support

In the last decade, particularly since various researchers have underlined the insufficient fluency of society with respect to information technology, primary and secondary school education has become a major focus within educational software communities. Attempts to introduce technology to youth have expanded to after-school activities where students have a chance to work with the current popular tools in informal and relaxed environments[19]. Some of the focus has been on audio-visual tools applied within creative settings. However, students mastering these creations were often barely introduced to computer programming and objects, which arguably motivates key concepts such as algorithmic thinking[17, 25]. The intuition behind this kind of introduction stemmed from the fact that many early attempts to teach programming to youth had been plagued with difficulties due to the relatively complex syntactical structures of popular programming languages.

As a result, many efforts within the educational software communities began to focus on the use of graphical programming environments, where children can get more comfortable with graphical building blocks rather than the ordinary textual instructions. Several graphical languages and their corresponding platforms have been introduced according to this theme which tends to hide issues of syntax from the programmer. For instance, Scratch emerged as a graphical and educational version of Squeak, based on SmallTalk80 where blocks from a script are represented as graphical programming blocks that can be used in a drag and drop manner. Squeak and others like Scratch, E-toys and Alice have been heavily used in after-school and summer activities as development platforms. In addition, tools like Lego Mindstorms, and Pico Crickets have introduced their own graphical environments to ease the use of hardware entities such as light and sound sensors.

Throughout our research, we have used most of the graphical languages mentioned for various outreach activities and witnessed their remarkable impact on introducing information technology to youth. In particular, during a weekly set of outreach activities with students in grades 2-7, we found that students made substantial progress in areas such as variables, looping, concurrency, and input/output. This rapid and comfortable progress directly relates to the impact of graphical programming environments as the students are isolated from the syntax-oriented complexities that naturally exist in textual programming languages[31]. Overall, not only did these graphical languages give the children an impressive arsenal of computational tools, but also equipped them with enhanced algorithmic-thinking skills, and a new perspective towards information technol-

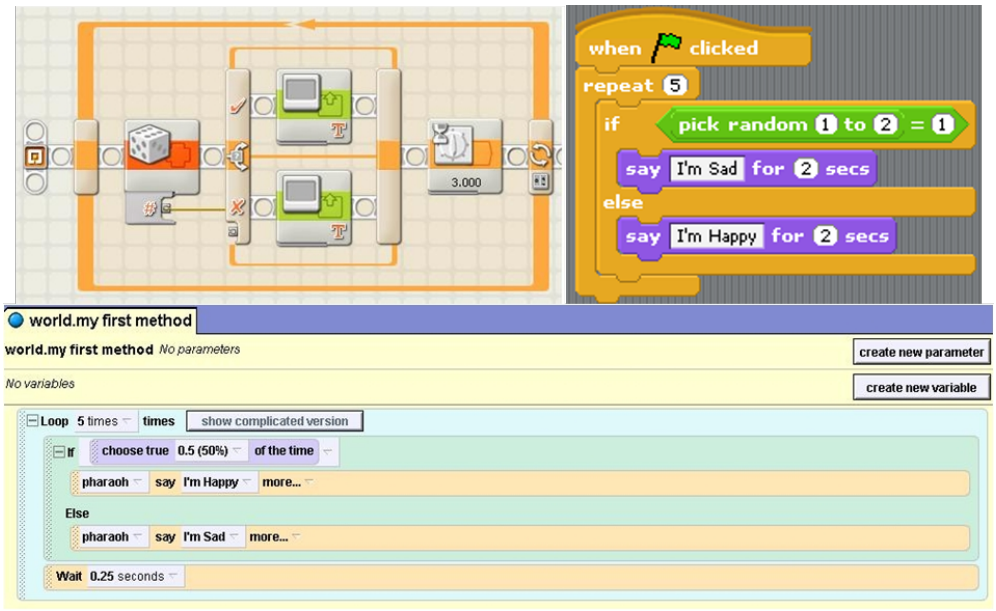

Figure 1. Snapshots, from LEGO Mindstorms (top-left), Scratch (top-right) and Alice (bottom), of a section of code which randomly selects between two phrases and display them repeating five times.

ogy.

3.2. Motivation and Related Work

Graphical environments generally make the logic of programming more accessible to beginner programmers by alleviating the burden of syntax[30]. There are many programming environments designed to teach the basics in the software field simultaneously. The members of the Lifelong Kindergarten Group from the MIT Media Lab have been working with domain specific representations such as Systems Thinking Blocks and other physical and graphical environments[20]. Additionally, this group has established afterschool programs and organize a network of Computer Clubhouses[29, 6].

Programs such as these have been instrumental in the development and implementation of multiple learning tools. For example, in the Pico Crickets programming environment, PicoBlocs gives the user the option to program textually or graphically. However, the user is unable to switch between the two representations, or view them simultaneously. In addition, Squeak contains similar features, such as the ability to view graphical blocks in their textual format, yet users are unable to modify the graphical representation through editing the textual representation.

Projects such as BlueJ[33] and Greenfoot[11] focus on teaching the principles of object oriented programming through mapping graphical entities, as in UML diagrams, to textual representations. Further, jGrasp[14] is an educational tool built in Java using visual representations to improve software understanding.

3.3. Transition: Graphical to Textual

Our experiences over the last year have revealed the advantages of graphical environments. The question remains as to whether the transition from these graphical environments to textual representations, and finally to a mental model of computation, could in fact be enhanced by tool support.

Recent research on teaching introductory level computer science courses with Alice revealed that although the students had positive attitudes towards the graphical programming experience, they had major difficulties with the transition to high-level languages like C++ and Java[28]. Some of the challenges during this transition were associated with difficulties in understanding objects even though every visual entity in the Alice environment is essentially an object. The difficulties appear to be associated with the transition to syntactical development and debugging. It is not unreasonable to assume that the same impact will be associated with other graphical environments. Throughout our study, we did not come across a platform that provides an incremental and transparent transition from graphical languages to textual languages.

In order to provide a more transparent transition, we pro-

pose to integrate a graphical environment into a platform which also actively supports textual programming. This support could take the form of a plug-in specifically designed to support these mappings.

The approach we plan to implement in a project called *GlassOnion*[34] is based on the intuition that this integration could be done using a tabbed interface to isolate the student from the associated intellectual context switching involved in multiple representations. Additionally, this plug-in could perform on-the-fly translations along a spectrum of languages. This would facilitate a completely transparent transition where the programmer can view the textual representations of his/her graphical program. Enhancing the visualization at the source code level we plan to use BRICS[5], a system for introducing visual object-like structure to method level control structures.

Another component that we envision is a step-through debugger to allow users to walk through their programs in multiple languages simultaneously. Such a component would explicitly map between graphical and textual representations. Along the same lines, runtime visualization support can range from views of object interaction and memory allocation for data structures in higher level languages to a view of registers in assembly language.

3.4. Transition: To Computational

Focusing on abstract views of computation is an arguably proficient way of motivating interest through an easy-to understand high-level view in first year post-secondary courses. However, balancing the level of abstraction relative to modern multiprocessor architectures emerges as another, and perhaps a more challenging issue.

In addition to multi-lingual representation of an algorithm graphically and textually, the design of GlassOnion further includes components that visualize computational activity at the hardware-level in order to provide a mapping between high-level and low-level aspects of computation during program execution.

Our goal is to express technical details through simple animations in order to help create a more aware and ready generation of Computer Scientists. Many simulations such as these for physics have been proven valuable to students[26], and hope to connect students with architectural appreciation in much the same way.

Throughout the design of such a component it is key to carefully select the right entities and activities to visualize. These visualizations and activities must help the students to get a better grasp of the nature of computation in general. Another important point is to decide on an optimum way of presenting these visualizations which will not overwhelm the user with unnecessary content. Instead, we envision a customizable outlook that is based on the encapsulation of different views in a hide/show manner used by former simulators such as ARMSim[2].

The architectural visualization component is comprised of two subcomponents: (1) storage visualization component, and (2) process visualization component. The storage visualization component consists of memory, cache and virtual memory views, while the process visualization component consists of onchip storage and chip activity views.

In memory view the user can follow the activity in specific memory locations depending on how the allocation is performed. On one end the stack subview simulates static memory usage, while on the other heap subview outlines the dynamic memory allocations performed by the program. Through these views, the user can actively follow the usage of memory through assignments, modifications, allocations and disposals. Visualization of memory can be particularly effective in teaching concepts such as variables and data structures. Furthermore, memory related concepts that are considered to be somewhat complex, such as pointers in C programming language, can visualised at runtime.

This technical representation can be extended to include assembly language with the animation of data flow through registers and logic units, and simulation of virtual memory address translation. This deeper representation gives students support for understanding underlying mechanisms of computation.

3.5. Transition: Individual to Collaborative

In addition to the need for a transitional tool, it is also important to note the necessity of support for collaboration within the supported views. Throughout our experiences with outreach activities, we have observed the continuous communication that children naturally used to enhance their projects. This informal communication at the development stage has motivated us to consider whether collaboration without context switching is possible.

Here, it is very important to note that the idea of collaboration is not new in the domain of graphical environments. Two examples are the Squeak (e.g. Nebraska[22]and OLPC) and Scratch (e.g. NetScratch[23]). However, these two examples are rather restricted to their environments. Furthermore, projects like Jazz have integrated collaboration into an IDE where a group of developers can actively communicate during the project development. Motivated by these ideas and approaches, we envision a collaborative environment that not only facilitates communication but also regulates and visualizes access to shared entities in a collaborative project. Such visualization can be provided through the use of abstract representations similar to a subset of UML. For example, actively shared entities can be rendered using different colors or different shapes. A user's artifact is identified by colour. Users can then simply look at the

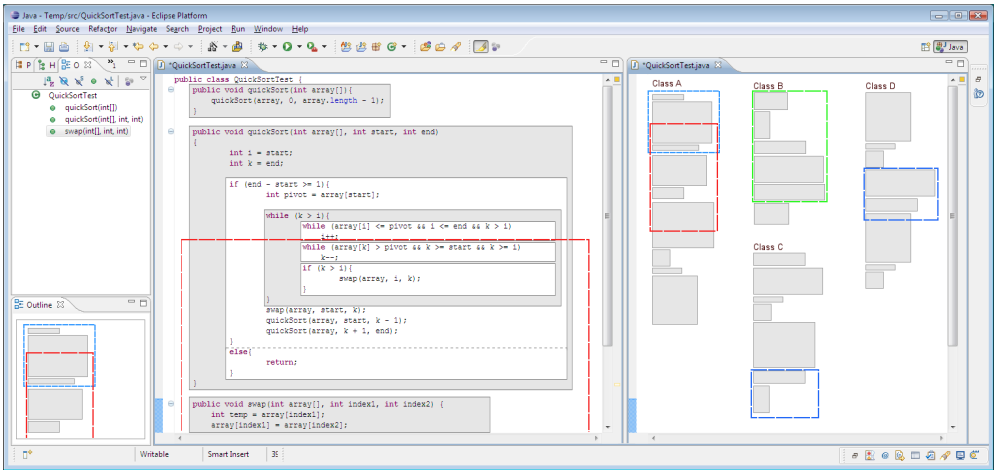

Figure 2. The vision of collaboration as an Eclipse plug-in, showing the source code and overview windows, and what the rest of the team is viewing.

visualization to know where their team members have been working. Figure 2 illustrates how the collaboration is visualized. Colours on the right-hand-side correspond to active team members and gives awareness of where they are currently working in the shared code-base.

At the lower levels, challenges with facilitating this kind of collaboration must be implemented using a limited amount of computational resources. We also envision this system to work with both centralized and wireless ad hoc peer-to-peer settings.

It is important to note that our vision is to embed active collaboration into a tool, or preferably a comprehensive plug-in, that is specifically built for supporting transition from graphical to textual environments and a deeper exploration into computational entities. We suggest that such an approach would not only facilitate an understanding of syntactical restrictions but also motivate teamwork by providing a natural and positive context for informational sciences for both secondary and post-secondary curricula.

We believe this approach is feasible given the huge shift in display technologies over the past few years. LCD screens have become less expensive than comparable CRT monitors, and this has lead to a change in the way computers are produced, marketed and sold. Notebook computers are now priced competitively with desktop computers, and those desktop computers now have a wide selection of displays with significantly higher resolution. As real-time collaboration software requires awareness, and awareness requires display area, this change will usher in significant improvements in real-time collaboration software. Further, we envision that enhancing educational laptops with the capability to scale by linking multiple devices together to share both monitor real-estate and processing capacity is not out of the question.

The educational scaffolding supplied by this nature of support has direct ramifications for remote instruction and mentoring. Given a representation such as that mocked up in Figure 2, it would be easy to accompany another person in a debugging session, bringing attention to certain code segments or demonstrating coding technique. Monitoring capabilities, so as to collect statistics about the ways in which programmers develop their solutions to assignments and lab exercises, is also a possibility.

4. Future Work and Conclusions

In this paper we have outlined the need for community based initiatives and provided concrete examples from our own project working with aboriginal communities on the west coast of British Columbia. We further identified criteria we believe to be key in terms of transparent and collaborative IDE support for programming when access to educational support is limited.

By supporting transparency and collaboration at all levels of abstraction, from high-level graphical languages to low-level computational representations, we believe we can provide a sustainable educational infrastructure package. Further, by deploying this package within cultural contexts, we believe C-DIT can plant initiatives that take hold in remote communities. The role of collaboration is fundamental, as it could allow for immediate feedback with projects that involve active teamwork, peer interest, or mentoring advice.

Our goal is to increase appreciation for the inherently

multi-faceted nature of programming. We believe that this incremental mapping of both computational elements and software development processes will allow students to more easily move from highly-constrained environments to more realistic, dynamic and exciting settings, thereby increasing their appreciation of, and investment in, the art of programming.

References

[1] Alice, http://www.alice.org/ Retrieved, 2007.

[2] ARM Simulator. http://sim.sagepub.com/cgi/content/abstract/80/4-5/221, Retrieved, 2007.

[3] Bebo, http://www.bebo.com/ Retrieved, 2007.

[4] L. Cheng. Jazzing up Eclipse with collaborative tools. In *Proceedings of ACM SIGPLAN Conference on Objected Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 45–49, Anaheim, California, USA, October 2003. ACM Press, New York.

[5] C. Pearson, C. Gibbs, and Y. Coady. Intuitive Source Code Visualization Tools for Improving Student Comprehension: BRICS, *Objected Oriented Programming, Systems, Languages, and Applications Workshop 1: Process in Object Oriented Pedagogy*, October 2007.

[6] Computer Clubhouse, http://www.computerclubhouse.org/ Retrieved, 2007.

[7] A. Conway and K. Christiansen. Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 486–493, The Hague, The Netherlands, April 2000. ACM Press, New York.

[8] D. Eastmond. Realizing the Promise of Distance Education in Low Technology Countries. *Educational Technology Research and Development*, 48(2):100–111, 2000.

[9] http://www.eclipse.org/ Retrieved, 2007.

[10] D. Hendrix. Designing a First-year Project Course to Engage Freshman Software Engineers: An Experience Report. In *Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET)*, 2006.

[11] P. Henriksen. Greenfoot: Combining object visualization with interaction. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 73–82, Vancouver, BC, Canada, November 2004.

[12] D. Ingalls and A. Kay. Back to future: the story of Squeak, a practical SmallTalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Objected Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 318–326, Atlanta, Georgia, USA, October 1997. ACM Press, New York.

[13] International Technology Education Association. *Technological Literacy*, 2000.

[14] JGrasp, http://www.jgrasp.org/ Retrieved, 2007.

[15] A. Kay. Etoys and SimStories in Squeak, http://www.squeakland.org/projects/etoys/etoyssimstories.004.pr Retrieved, 2007.

[16] A. Kay. Squeak EToys, Children & Learning, Online Article http://www.squeakland.org/pdf/etoys_n_learning.pdf Retrieved, 2007.

[17] Kay, Alan. Computers, Networks and Education. Scientific American, September 1991, p. 138-148.

[18] LEGO Mindstorms. http://mindstorms.lego.com/ Retrieved, 2007.

[19] J. Maloney and M. Resnick. Scratch: A Sneak Preview. In *Proceedings of the Second International Conference on Creating, Connecting, and Collaborating through Computing*, pages 104–109, Kyoto, Japan, 2004.

[20] MIT Media Lab: Life Long Kindergarten, http://llk.media.mit.edu/projects.php Retrieved, 2007.

[21] National Academy of Engineering and National Research Council. *Technically Speaking: Why All Americans Need to Know More About Technology*. National Academy Press, Washington, DC, 2002.

[22] Nebraska. http://web.media.mit.edu/ tstern/netscratch/index.html Retieved, 2007.

[23] NetScratch. http://web.media.mit.edu/ tstern/netscratch/index.html, Retrieved, 2007.

[24] One Laptop per Child, http://www.laptop.org/ Retrieved, 2007.

[25] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., 1980.

[26] Physics Education and Technology. http://phet.colorado.edu/new/index.php Retrieved, 2007.

[27] Pico Crickets. http://www.picocrickets.com/ Retrieved, 2007.

[28] K. Powers. Through the looking glass: teaching CS0 with Alice. *SIGCSE Bull*. 39, 1 (Mar. 2007), 213-217.

[29] M. Resnick. A Networked, Media Rich Programming Environment to Enhance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities. NSF Proposal, 2003.

[30] Science Venture, http://scienceventure.uvic.ca/home/ Retrieved, 2007.

[31] A. A. St Pierre. Young Minds Stroming Through Challenging Computer Science Concepts. In *Proceedings of Western Canadian Conference on Computing Education (WCCCE)*, 2007.

[32] Tsawout First Nations Computer Workshop, http://outreach.cs.uvic.ca/cside/novcamp/index.html Retrieved, 2007.

[33] K. Van Haaster. Teaching and Learning BlueJ: an Evaluation of a Pedagogical Tool. In *Information Science + Information Technology Education Joint Conference*, Rockhampton, QLD, Australia, June 2004.

[34] Y. O. Yazır, S. Lonergan, K. Gunion, and Y. Coady. Looking Through a Glass Onion: Transparent Layers for Concrete Abstractions, *Objected Oriented Programming, Systems, Languages, and Applications Workshop 1: Process in Object Oriented Pedagogy*, October 2007.