SIGCSE : G : Minding the gap between blocks-based and text-based programming: Evaluating introductory programming tools

David Weintrop
Northwestern University
2120 Campus Drive, Ste. 332
Evanston, IL 60208
dweintrop@u.northwestern.edu

1. INTRODUCTION & MOTIVATION

Computation is changing our world. From how we communicate and how we make decisions, to how we relax and how we shop - few aspects of our lives have been left unaffected by the long reach of computation and the technologies that it enables. Smartphones, tablets, and laptops have become the lenses through which we see, organize, and interpret the world. As such, for young learners growing up in this technological landscape, being able to recognize the capabilities and limitations of these technologies, and most critically, to be able to contribute in this technological culture is essential. Programming is the skill that enables this participation. Programming, and the critical thinking and problem solving skills that accompany it, constitute a new 21st century literacy that will need to live alongside reading, writing, and mathematics as essential competencies to empower today's students to fully engage with our technological world. These skills have far reaching benefits as they underpin and enable new forms of creative expression, support learning in computational contexts across a wide range of disciplines, and provide the foundation for future careers in our increasingly computation-driven economy.

Bringing programming into K-12 education is a critical step for introducing learners to this fundamental skill. A long-standing question faced by educators is deciding where to start on day one: What programming language to choose? In what environment? An increasingly popular approach to the design of introductory programming tools is the use of graphical, blocks-based programming environments that leverage a primitives-as-puzzle-pieces metaphor and support drag-and-drop composition (Figure 1). In such environments, learners can assemble functioning programs using only a mouse by snapping together instructions and receive visual feedback on how and where commands can be used and if a given construction is valid. The use of this programming modality has become a core feature of introductory computer science curricula and programming interventions targeted at young learners. Notably, national curricular efforts including Exploring Computer Science [19], the CS Principles project [2], and Code.org's curricular materials utilize blocks-based tools to introduce students to programming.

Despite its growing popularity and widespread use, little work to date has focused on the conceptual and affective benefits of using blocks-based tools in formal educational contexts. Open questions remain on the effectiveness of the approach for helping students learn basic programming concepts and whether or not blocks-based tools are effective for preparing students for future computer science learning opportunities that utilize conventional text-based languages. Further, it is unclear what the strengths and weaknesses of block-based programming tools are compared to isomorphic text-based alternatives, and relatively little work has explored the design space blending text and blocks-based features.

(a) (b) (c)

Figure 1. Three example blocks-based programming languages: (a) Snap!, (b) Scratch, and (c) Alice.

The goal of the work presented here is to better understand the affordances and limitations of using blocks-based environments in formal computer science contexts, and then apply that knowledge to the design a new tool for learning to program that draws on the strengths of both the blocks-based and text-based modalities. To answer these questions we conducted a study comparing three versions of a blocks-based programming tool in high school introductory programming classes. In doing so, we seek to provide evidence to better inform educators who are tasked with making consequential decisions around how learners are introduced to computer science and, more generally, to contribute to our understanding of the relationship between programming tools and the understandings and practices they promote.

2. RESEARCH QUESTIONS

This work seeks to answer a pair of interrelated research questions. The first pertains to how the representations used by a programming environment effect students' resulting understandings of programming concepts and what programming practices they engendered in the learner. There is a growing body of literature on the effects, both positive and negative, of using blocks-based programming environments with novices in formal education environments [1, 11, 22, 27, 28]. This work largely relies on pre/post test results, providing little in the way of mechanistic explanations of the findings that attend to features of the block-based programming approach. Our approach to answer this question uses a mixed method design including conventional assessment measures along with qualitative and computational data sources to gain a deeper understanding of how learning happens with these representational tools.

The second set of questions look at the effectiveness of introductory programming tools for preparing students for future computer science learning opportunities. While blocks-based programming environments have been found to be successful at engaging students in programming activities and providing learners with early successes with little or no formal instruction

[24, 25], educators have had difficulty transitioning learners from these graphical environments to conventional text-based programming languages. Studies have found little transfer in knowledge between graphical environments intended to introduce learners to programming and the text-based environments that serve as the core modality for further computer science studies [9, 10, 17, 31]. These findings are not universal as there have been some successful efforts bridging graphical approaches and text-based programming [1, 12, 26]. Like with the first research question, much of the work done in this area has used pre/post test measures to identify successful transfer. In our study, we combine these measures with microgenetic approaches and a quasi-experimental design to comparatively studying learning as it happens to advance our understanding of if and how concepts and practices do or do not carry over from introductory graphical tools to more conventional text-based programming contexts.

Along with these two questions, with this work we are beginning to explore the design space the blends blocks-based and text-based programming approaches. This is an area of active research as a number of introductory programming tools are attempting to do this, often in an effort to support the blocks-to-text transition [3, 12, 26]. Our goal for this design work is to develop an environment we can confidently advocate for classroom use and advance a set of design principles for creating effective hybrid programming tools based on the findings from this study.

3. BACKGROUND AND RELATED WORK

3.1 Representations and Learning

"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities." [13]

As stated by the Turing Award winning computer scientist Edsger Dijkstra in the quote above, the tools we use, in this case the programming languages and development environments, have a profound, and often unforeseen, impact on how and what we think. The role of representations on cognition has been studied across a variety of representational systems and their influence explored on various cognitive tasks. For example, Sherin [33] studied students learning basic physics concepts using either conventional algebraic notation or using programming languages to represent physics ideas and found that the two representational forms had different affordances with respect to students learning physics concepts and, as result, affected their conceptualization of the concepts learned. "Algebra physics trains students to seek out equilibria in the world. Programming encourages students to look for time-varying phenomena, and supports certain types of causal explanations, as well as the segmenting of the world into processes" [33]. Similar work has investigated other such relationships, including the effect of textual literacy on thought [8, 23, 29], numerical representation (Roman vs. Hindu-Arabic numerals) on learnability and expressive power [14, 35], and programming notation on reader comprehension [18]. Wilensky and Papert [41] use the term structuration to describe the relationship between the representational infrastructure used within a domain and the understanding that infrastructure enables and promotes. While often assumed to be static, Wilensky and Papert show that the structurations that underpin a discipline can, and sometime should, change as new tools and ideas emerge. Given the rise of blocks-based programming in introductory learning contexts, it is critical we understand how this representational shift is affecting learners so we can make informed decisions about if and how it can most effectively be incorporated into used in classrooms.

3.2 Blocks-based Programming

The blocks-based approach of visual programming, while not a recent innovation, has become widespread in recent years with the emergence of a new generation of tools, lead by the popularity of Scratch [32], Snap! [20], and Blockly [16]. These programming tools are a subset of the larger group of editors called *structured editors* [15] that make the atomic unit of the composition tool a node in the abstract syntax tree (AST), as opposed to a smaller component (like a character or word) or a larger element (like a fully formed functional unit). In making AST elements the compositional building blocks, then providing constraints to insure nodes can only be added to the program's AST in valid ways, the environment can prevent syntax errors. These constraints are imposed in a number of ways. Blocks-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus preventing syntax errors but retaining the practice of assembling programs instruction-by-instruction. This features is especially relevant in this study, as graphical programming proponents boast that the lack of syntax is a key features that contributes to its appropriateness for young learners [32], but research is finding this approach does not solve the syntax problem, but only delays it [30, 31]. Along with using block shape to denote usage, there are other visual cues to help novice programmers including color coding blocks by conceptual use, and nesting of blocks within scripts to denote scope. Blocks-based programming is perceived as easier by learners, with a number of these visual features cited as reasons for its relative ease-of-use [39].

Early versions of this interlocking blocks-based such as LogoBlocks [4] and BridgeTalk [6] which helped formulate the programming approach which has since grown to be used in dozens of applications. Alice [11], an influential and widely used environment used in introductory programming classes, uses a very similar interface and is widely used in undergraduate introductory programming courses. In addition to being used in more conventional computer science contexts, a growing number of environments have adopted the blocks-based programming approach to lower the barrier to programing across a variety of domains including mobile app development with MIT App Inventor and Pocket Code [34], modeling and simulation tools including StarLogo TNG [5], DeltaTick [42], and EvoBuild [37], creative and artistic tools like Turtle Art [7], and PicoBlocks, commercial educational programming applications like Tynker and Hopscotch, and game-based learning environments such as RoboBuilder [38], Lightbot and the activities included in Code.org's Hour of Code and Google's Made with Code initiative. Collectively, all these new blocks-based tools further reinforce the need to better understand the cognitive and affective affordances of the modality.

4. RESEARCH DESIGN

4.1 UNIQUENESS OF APPRAOCH

Given the increasing prominence of graphical, and in particular, blocks-based programming tools, our goal is to provide insight into the consequences (both positive and negative) of choosing a given programming modality for learners' first programming experiences. To date, much of the work in this space has looked at the use of blocks-based tools in informal spaces (afterschool, summer camps, etc.) as opposed to classrooms, and largely

focused on issues of engagement, recruitment and attitudes. Also, our focus is on high school computer science classes, an age group understudied relative to younger learners and undergraduate students, who more often serve as the participants in similar studies. Additionally, we have designed a mixed method, multi-modal study, including the use of an innovative assessment tool, to allow us to comprehensively answer our stated research questions.

4.2 Study Design

Drawing inspiration from the learning sciences, we designed a mixed-methods, quasi-experimental study to answer our stated research questions. The study took place at an urban, public high school and ran for the first 10 weeks of the school year in three sections of an Introduction to Programming class. For the first five weeks of the course, each class used a slightly different programming environment based on Snap! [15]. Snap! is a blocks-based programming tool that is very similar to Scratch, but adds a few features (notably Snap! has first-class functions), and is completely implemented in JavaScript. The first class served as a control and used an unmodified version of Snap! The second class used a version of Snap! that had the ability to right-click on any block or script and open up a window showing a JavaScript implementation of the selected block or script (Figure 2). This served as a hybrid, blocks/text read-only environment, as students were able to read, but not edit or write, text-based versions of the programs they constructed with the blocks.

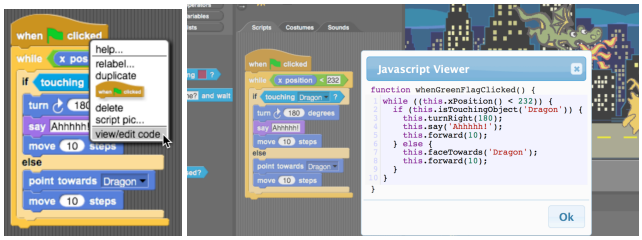

Figure 2. Side-by-side blocks and text in our version of Snap!

The third class used a version of Snap! that allowed students to read their programs in text, like the read-only condition, and added the ability to define the behavior of new custom blocks in JavaScript. This served as a hybrid blocks/text read/write environment, as students could both read a text-based version of their own blocks, as well as write the behaviors of new blocks in JavaScript. The usual workflow for defining new blocks was for students to author the behavior with blocks, view the JavaScript equivalent, and then copy/paste the text into their new block. In this way, students in the read-write condition were usually not writing JavaScript from scratch, but instead doing more tinkering and tweaking of the textually defined behaviors. It is important to note in this condition, students were only writing small snippets of code (usually 4 lines or less) to define custom block behaviors and then integrating the text-defined custom blocks into larger scripts.

All three classes worked through the same set of activities in their respective environments. The five-week curriculum for the introductory course was based on the Beauty and Joy of Computing course and covered fundamental programming concepts including variables, procedures, looping, and conditional logic. Starting in week six, students transitioned from the introductory tool to Java, the language they used for the remainder of the school year. We followed students for the first 5 weeks of learning Java. For the Java portion of the course, students followed the same curriculum the course had historically been taught with, based on the Java Concepts: Early Objects book [21].

4.3 DATA COLLECTION

A variety of data were collected as part of our mixed-methods study design. Attitudinal surveys and content assessments were administered three times during the study: at the outset (beginning of week 1), at the conclusion of the first phase of the study after students had completed working with the introductory environments but before they had started with Java (end of week 5), and at the conclusion of the study, after students have been learning Java for five weeks (end of week 10). The attitudinal surveys were based on existing, validated surveys used in similar research studies [36]. The content assessments were designed specifically for this project and included a set of 30 "reversible" questions (Figure 3) that covered the topics students encountered during the five-week introductory portion of the study. Each question presents a short program (3-6 lines) for the student to read. The code is presented as either a blocks-based program or written in a text-based language. The modality students see alternates both within and across assessment.

Figure 3. Two "reversible" assessment questions.

Along with our quantitative data, we also conducted semi-structured clinical interviews throughout the study. For these interviews, a researcher sat alongside a student as they both face a computer that had a version of the introductory programming tool on screen (Figure 4). Students were asked to read existing programs written in various modalities, as well as author new programs in both blocks and text. Interviews were recorded using software that captured both what is displayed on the screen as well as the student (via the onboard camera). We also conducted teacher interviews at regular intervals through out the study and did observations of all three classrooms.

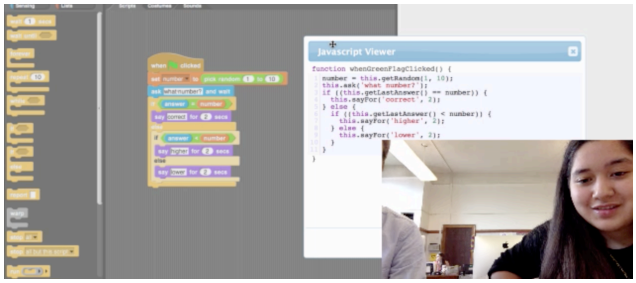

Figure 4. A screen shot from an interview

Finally, we recorded all of the student-authored programs over the course of the study. This included programs written across the three introductory tools as well as Java programs authored in the subsequent weeks. The programs will be used as a data corpus for analyses using educational data mining techniques.

This fall we spent ten weeks collecting data at a selective enrollment, urban public high school. A total of 90 students across three sections of the course participated in the study, which included 67 male students and 23 female students. The students participating in the study were 43% Hispanic, 29% White, 10% Asian, 6% African American, and 10% Multi-racial - a breakdown comparable to the larger student body. The classes included one student in eighth grade, three high school freshman,

43 sophomores, 18 juniors, and 25 high school seniors. Two-thirds of the students in these classes speak a language other than English in their homes. Over the course of the ten-week study we gathered 88 sets of pre/mid/post attitudinal assessments and over 8,500 responses to our reversible content assessments. We also conducted 32 interviews (9 pre, 10 mid, 8 post, 5 teacher) and collected over 73,000 Snap! programs and over 10,000 Java programs.

5. RESULTS

5.1 STUDENTS' PERCEPTIONS OF BLOCKS-BASED PROGRAMMING TOOLS

Our first investigation was into the question of how students perceive the differences between blocks-based and text-based programming [39]. This analysis drew on data collected in the mid and post attitudinal surveys and was supplemented by the interviews we conducted. We found that a majority of students (92%) viewed blocks-based programming as easier than text-based programming. Students cited a number of features of blocks-based programming as contributing to this, including the ease of the drag-and-drop composition, the lack of needing to memorize commands, and the fact that in the tools we used, blocks were closer to natural language than text-based programming languages are, making them easier to read. This view was not universal as 4% of students thought text-based programming was easier, with the remaining 4% saying they thought the two environments were comparable with respect to ease-of-use. Students also identified drawbacks of the blocks-based programming approach, including issues of authenticity and the difficulty of building large complex programs with block-based tools. This finding provides evidence that reinforces our intuition that the blocks-based modality is easier for novices and gives us insight into what features are seen as the sources of this ease-of-use through the eyes of a learner. Finally, these findings provide guidance for the design of future learning environments as it reveals what features are critical and what features are problematic for high school students about blocks-based tools.

5.2 STUDENT PERFORMANCE ON REVERSIBLE ASSESSMENTS

Using responses to our reversible assessments, we can investigate the relationship between modality (blocks-based or text) and programming concepts. Students answered reversible questions about four basic programming concepts: variables, iterative logic, conditional logic, and procedures, along with program comprehension questions. Five questions from each category were asked, with the questions being presented in a blocks-based or text-based way. For our analysis, we grouped student responses by concept and modality to see how the representation affected students' performance (Figure 5). We found that students performed significantly better on blocks-based questions when the questions pertained to conditional logic $t(178) = 2.80$, $p < .01$, iterative logic $t(178) = 10.64$, $p < .001$ and procedures $t(178) = 2.79$, $p < .01$. Student also performed better on variables in the graphical condition, although not significantly. Interestingly, there was no difference in performance on comprehension questions between the two modalities. This suggests that while the graphical representation supports students in understanding what a construct does (i.e. what the output from using it is), that support does not better facilitate learners in understanding how to use that construct or how it fits into a larger algorithm. This is potentially a very consequential finding and we are in the process of conducting a deeper analysis to explain this finding.

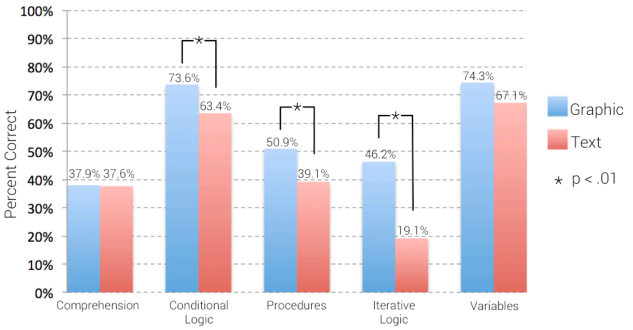

Figure 5. Student performance on reversible assessment questions grouped by modality and concept.

5.3 PROGRAMMING PATTERNS IN BLOCKS-BASED TOOLS

The final set of analyses we have conducted uses the log data we collected to investigate different patters of interaction of students in our three introductory tools. During the 10-week study we recorded all of the code written by students. With this data we can follow the paths learners took as they worked through a problem. Figure 6 shows the trajectories of three students as they progressed through one assignment.

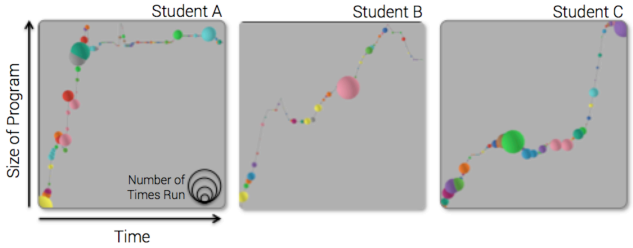

Figure 6. Three student trajectories in blocks-based tool.

The x-axis of these graphs shows each successive version of the student's program, while the y-axis depicts the size of the program in terms of number of blocks used. The size of the dot at each step shows how many times that specific version of the program was executed. These three graphs depict three distinct approaches to developing programs with a blocks-based tool. Student A started by very quickly added a lot of blocks to her program (shown by the steep positive slope). Upon getting all the pieces in place, she ran the program a large number of times (the large dots before the slope decreases), before proceeding with a large number of small changes as she worked toward the final version of the program (the relatively flat portion of the graph). Student C took the opposite approach, starting with a few blocks and judiciously adding them while running each early version of the program many times, only making large additions to her program at the end. Student B's graph lives somewhere between these two approaches, building up his program at a relatively constant rate, and with one notable exception, running each version of the program roughly the same number of times before making the next set of changes. This is only an initial pass through our program log data and we have plans to conduct more sophisticated and nuanced computational analyses in the future, including measuring AST distance and the use of key computational constructs to accomplish specific tasks.

5.4 CONTRIBUTIONS

We expect the findings of this study will compliment existing work done on introductory programming environments and give

us a broader picture of how the latest generation of block-based programming tools fit into more formal, structured educational spaces, as well as provide insight into the cognitive and procedural dimensions of such tools. The other contribution we see this work making is the introduction of a new environment or set of design principles, informed by research, that addresses some of the shortcomings of existing introductory blocks-based programming tools.

We are at a critical juncture in the history of computer science education. The ability to program is a central skill all students should develop, but it is currently absent from the coursework of many of today's students. To address this gap, educators, school administrators, and state and national legislators are all taking action to bring computer science into the classroom. The practices, tools, and curricula that are being developed today, will become the standards used for years to come. Therefore, it is critical that we are confident that the curricula and environments we advocate for today are effective at teaching the core concepts, engaging learners from diverse backgrounds, and successful in preparing students for the computational endeavors they will face in the future. The findings from this study will advance our understanding of how best to introduce students to these core 21st century skills and contribute new tools that will prepare students to be successful in the computational futures that await them.

6. REFERENCES

[1] Armoni, M. et al. 2015. From Scratch to "Real" Programming. *ACM Trans. on Computing Ed*. 14, 4, 25.

[2] Astrachan, O. and Briggs, A. 2012. The CS principles project. *ACM Inroads*. 3, 2 (2012), 38–42.

[3] Bau, D. and Bau, D.A. 2014. A Preview of Pencil Code: A Tool for Developing Mastery of Programming. *Proc. of the Programming for Mobile & Touch*, 21–24.

[4] Begel, A. 1996. *LogoBlocks: A graphical programming language for interacting with the world.* EECS. MIT.

[5] Begel, A. and Klopfer, E. 2007. Starlogo TNG: An introduction to game development. *Journal of E-Learning*.

[6] Bonar, J. and Liffick, B.W. 1987. A visual programming language for novices. *Principles of Visual Programming Systems*. S.K. Chang, ed. Prentice-Hall, Inc.

[7] Bontá, P. et al. 2010. Turtle, Art, TurtleArt. *Proc. of Constructionism 2010* (Paris, Fr.).

[8] Boroditsky, L. 2001. Does Language Shape Thought?: Mandarin and English Speakers' Conceptions of Time. *Cognitive Psychology*. 43, 1, 1–22.

[9] Chetty, J. and Barlow-Jones, G. 2012. Bridging the Gap: the Role of Mediated Transfer for Computer Programming. *Proc. of Comp. Sci & Inf. Tech*. 43.

[10] Cliburn, D.C. 2008. Student opinions of Alice in CS1. *Frontiers in Education Conference, 2008*, T3B–1.

[11] Cooper, S. et al. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of CS in Colleges* 15, 5.

[12] Dann, W. et al. 2012. Mediated transfer: Alice 3 to Java. *Proc. of the 43rd SIGCSE*, 141–146.

[13] Dijkstra, E.W. 1982. How do we tell truths that might hurt? *Selected Writings on Computing: A Personal Perspective*. Springer. 129–131.

[14] diSessa, A.A. 2000. *Changing minds: Computers, learning, and literacy*. MIT Press.

[15] Donzeau-Gouge, V. et al. 1984. Programming environments based on structured editors: The MENTOR experience. D. Barstow et al., eds. McGraw Hill.

[16] Fraser, N. 2013. *Blockly*. Google.

[17] Garlick, R. and Cankaya, E.C. 2010. Using Alice in CS1: A quantitative experiment. *Proc of the 15th ITiCSE,* 165–168.

[18] Gilmore, D.J. and Green, T.R.G. 1984. Comprehension and recall of miniature programs. *Int. Journal of Man-Machine Studies*. 21, 1, 31–48.

[19] Goode, J. et al. 2012. Beyond curriculum: the exploring computer science program. *ACM Inroads*. 3, 2, 47–53.

[20] Harvey, B. and Mönig, J. 2010. Bringing "no ceiling" to Scratch. *Proc. of Constructionism 2010* (Paris, Fr.), 1–10.

[21] Horstmann, C.S. 2012. *Java Concepts: Early Objects*. Wiley.

[22] Lewis, C.M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proc. of the 41st SIGCSE* (New York, NY), 346–350.

[23] Luria, A.R. 1982. *Language and cognition*. Winston ; Wiley, Washington, D.C. : New York ; Chichester :

[24] Malan, D.J. and Leitner, H.H. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, 223–227.

[25] Maloney, J.H. et al. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin*. 40, 1, 367–371.

[26] Matsuzawa, Y. et al. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment., 185–190.

[27] Meerbaum-Salant, O. et al. 2011. Habits of programming in Scratch. *Proc. of 16th ITiCSE* (Darmstadt, Ger.), 168–172.

[28] Meerbaum-Salant, O. et al. 2010. Learning computer science concepts with scratch. *Proc. of the 6th ICER*, 69–76.

[29] Ong, W. 1982. *Orality and Literacy: The technologizing of the world*. Routledge.

[30] Parsons, D. and Haden, P. 2007. Programming osmosis: Knowledge transfer from imperative to visual programming environments. *Proc. of the 12th NACCQ Conference* (Hamilton, New Zealand), 209–215.

[31] Powers, K. et al. 2007. Through the looking glass: teaching CS0 with Alice. *ACM SIGCSE Bulletin*. 39, 1, 213–217.

[32] Resnick, M. et al. 2009. Scratch: Programming for all. *Communications of the ACM*. 52, 11, 60.

[33] Sherin, B.L. 2001. A comparison of programming languages and algebraic notation as expressive languages for physics. *Int. Journal of Computers for Mathematical Learning*. 6, 1.

[34] Slany, W. 2014. Tinkering with Pocket Code, a Scratch-like programming app for your smartphone. *Proc. of Constructionism 2014* (Vienna, Aus).

[35] Swetz, F. 1989. *Capitalism and arithmetic: The new math of the 15th century*. Open Court.

[36] Tew, A.E. et al. 2012. Toward a validated computing attitudes survey. *Pro. of the 9th ICER*, 135–142.

[37] Wagh, A. and Wilensky, U. 2012. Evolution in blocks: Building models of evolution using blocks. *Proc of Constructionism 2012* (Athens, Gr.).

[38] Weintrop, D. and Wilensky, U. 2012. RoboBuilder: A program-to-play constructionist video game. *Proc. of Constructionism 2012* (Athens, Gr.).

[39] Weintrop, D. and Wilensky, U. 2015. To Block or not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. *Proc. of the 14th IDC* (Boston, MA.).

[40] Wilensky, U. and Papert, S. 2010. Restructurations: Reformulating knowledge disciplines through new representational forms. *Proc of Constructionism 2010* (Paris, Fr.).

[41] Wilkerson-Jerde, M.H. and Wilensky, U. 2010. Restructuring Change, Interpreting Changes: The DeltaTick Modeling and Analysis Toolkit. *Proc. of Constructionism 2010* (Paris, Fr.).