Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms

DAVID WEINTROP, University of Chicago
URI WILENSKY, Northwestern University

The number of students taking high school computer science classes is growing. Increasingly, these students are learning with graphical, block-based programming environments either in place of or prior to traditional text-based programming languages. Despite their growing use in formal settings, relatively little empirical work has been done to understand the impacts of using block-based programming environments in high school classrooms. In this article, we present the results of a 5-week, quasi-experimental study comparing isomorphic block-based and text-based programming environments in an introductory high school programming class. The findings from this study show students in both conditions improved their scores between pre- and postassessments; however, students in the blocks condition showed greater learning gains and a higher level of interest in future computing courses. Students in the text condition viewed their programming experience as more similar to what professional programmers do and as more effective at improving their programming ability. No difference was found between students in the two conditions with respect to confidence or enjoyment. The implications of these findings with respect to pedagogy and design are discussed, along with directions for future work.

CCS Concepts: • **Social and professional topics → Professional topics**; **Computing education**; **K-12 education**;

Additional Key Words and Phrases: Block-based programming, programming environments, design

ACM Reference format:
David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Trans. Comput. Educ.* 18, 1, Article 3 (October 2017), 25 pages.
https://doi.org/10.1145/3089799

1 INTRODUCTION

There is a growing recognition that computing is an essential skill for all students to develop in order to fully participate in an increasingly digital world. In response to this identified need, new initiatives are underway seeking to bring computer science courses into high schools around the world. The movement to bring computing to a growing number of students has resulted in new curricula for high school classrooms using the latest generation of introductory programming environments. Many of these curricula are choosing to use block-based environments to serve as students' initial introductions to the practice of programming. For example, the Exploring

Authors' addresses: D. Weintrop, Department of Teaching & Learning, Policy & Leadership, College of Education and College of Information Studies, University of Maryland, 2226D Benjamin Building, College Park, MD 20742; U. Wilensky, Departments of Learning Sciences & Computer Science, Northwestern University, 337 Annenberg Hall, 2120 Campus Drive, Evanston, IL 60208.

https://doi.org/10.1145/3089799

Computer Science Curriculum (Goode et al. 2012), the CS Principles course (Astrachan and Briggs 2012), and the materials produced by Code.org for classrooms, all incorporate the use of various block-based programming tools. Despite the rise in prominence of block-based programming in formal settings, open questions remain as to the strengths and drawbacks of this programming modality when used in the classroom. Notably, relatively little comparative work has been done evaluating block-based programming interfaces against more traditional text-based alternatives in high school classrooms. This includes both questions of learning outcomes and attitudinal and perceptual effects from using such tools. This article seeks to address this gap in the literature by answering the following three-part research question:

> *How does block-based programming compare to text-based programming in high school introductory computer science classes with respect to learning outcomes, attitudes, and interest in the field of computer science?*

To answer this question, we conducted a 5-week quasi-experimental study in which two classes at the same school worked through the same curriculum using either a block-based or text-based interface for the same programming environment. Pre- and postcontent assessments were administered along with attitudinal surveys, semistructured clinical interviews, and classroom observations. This study design provides the data to answer the stated research question and shed light on how the design of introductory programming tools affects learners in high school classrooms. In the next section, we review the prior work this study is built upon, specifically reviewing research on the design of introductory programming environments and the relationship between representations and learning. Next, the methods used in this work are presented; this includes a discussion of the programming environment used, the curriculum students followed, and information about the school and students who participated in the study. Following the methods, we present the data and outcomes from a comparative analysis of the block-based and text-based conditions of the study. The article concludes with a discussion of these findings, potential implications, and the limitations and future work needed to more completely understand the role of modality in introductory programming classrooms.

2 LITERATURE REVIEW

2.1 Block-Based Programming

The block-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, led by the popularity of Scratch (Resnick et al. 2009), Snap! (Harvey and Mönig 2010), and Blockly (Fraser 2015). These programming tools are a subset of the larger group of editors called *structured editors* (Donzeau-Gouge et al. 1984) that make the atomic unit of composition a node in the abstract syntax tree (AST) of the program. Giving authors the ability to work directly on nodes in the AST is in contrast to providing smaller element (i.e., a character) or larger conceptual chunks (a fully formed functional unit). In making these AST elements the building blocks and then providing constraints to ensure a node can only be added to the program's AST in a valid way, the environment can protect against syntax errors. Block-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 as their means of constraining program composition. Programming in these environments takes the form of dragging blocks into a scripting area and snapping them together to form scripts. 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. Along with using block shape to

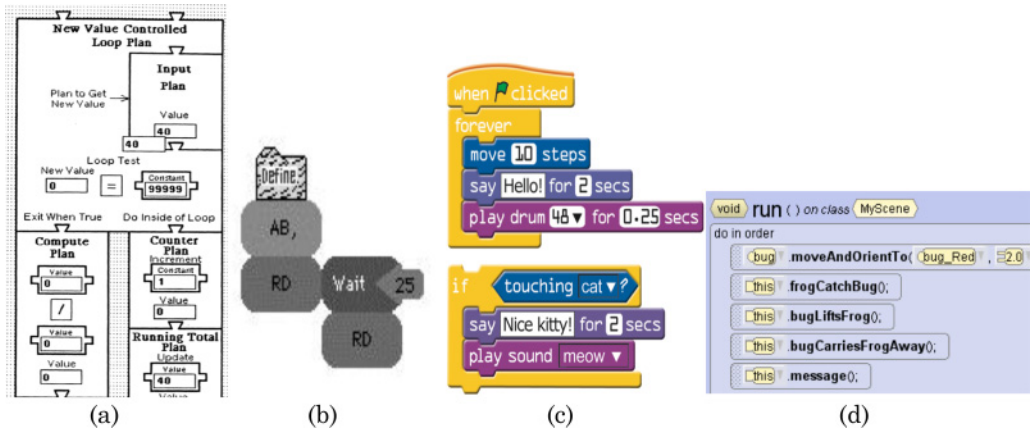

Fig. 1. Four sample block-based programming languages: (a) BridgeTalk, (b) LogoBlocks, (c) Scratch, and (d) Alice.

denote usage, there are other visual cues to help programmers, including color coding by conceptual use and the nesting of blocks to denote scope (Maloney et al. 2010; Tempel 2013).

Early versions of this interlocking approach include LogoBlocks (Begel 1996) and BridgeTalk (Bonar and Liffick 1987), which helped formulate the programming approach that has since grown to be used in dozens of applications. Alice (Cooper et al. 2000), an influential and widely used environment in introductory programming classes, allows learners to program 3D animations with a similar interface and has been the focus of much scholarship evaluating the merits of the approach. Agentsheets is another early programming environment that helped shape the block-based paradigm through its approach of providing users with an intermediate level of abstraction between high-level building blocks and low-level text-based language commands that alleviates the burden of syntax for novice programmers (Repenning 1993). Figure 1 shows programs written in a number of block-based programming tools.

In addition to being used in more conventional computer science contexts, a growing number of environments have adopted the block-based programming approach to lower the barrier to programming across a variety of domains. These include mobile app development with MIT App Inventor (Wolber et al. 2014) and Pocket Code (Slany 2014); modeling and simulation tools including StarLogo TNG (Begel and Klopfer 2007), DeltaTick (Wilkerson-Jerde and Wilensky 2010), NetTango (Horn and Wilensky 2012), and EvoBuild (Wagh and Wilensky 2012); creative and artistic tools like Turtle Art (Bontá et al. 2010) and PicoBlocks (The Playful Invention Company 2008); and game-based learning environments like RoboBuilder (Weintrop and Wilensky 2012), Lightbot (Yaroslavski 2014), and Google's Made with Code initiative. Further, a growing number of libraries are being developed that make it easy to develop application- or task-specific block-based languages (Fraser 2015; Roque 2007). This diverse set of tools and the ways the modality is being used highlight its recent popularity and speak to the need for more critical research around the affordances and drawbacks of the approach (Shapiro and Ahrens 2016; Weintrop and Wilensky 2015a). Designers are also looking beyond conventional block-based interfaces. There are a growing number of environments that support novices converting block-based programs to textual languages (e.g., Alice's Java Bridge (Dann et al. 2012) and Blockly's code generator feature (Fraser 2015)), as well as others that blend block-based and text-based programming approaches (e.g., Pencil Code (Bau 2015), Tiled Grace (Homer and Noble 2014), and Greenfoot's Frame-based editor (Kölling et al. 2015)).

2.2 Evaluating Block-Based Programming Environments

A variety of methodologies have been used to evaluate programming environments for novices, including heuristic evaluation, controlled laboratory studies, computational analyses of learner-created programs, and studies in classrooms and coding camps concerned with ecological validity. In evaluating the challenges of transition from block-based to text-based languages, researchers identified a number of features of the block-based approach to programming that facilitate novices, such as being more readable, relaxing the need to memorize commands or syntax, and easing the burden of typing by supporting dragging and dropping of commands (Kölling et al. 2015; Weintrop and Wilensky 2015b). Related efforts also identified drawbacks to block-based tools, such as not scaling well to larger programs and becoming cumbersome when defining commands with many components, such as mathematic formulae or complex Boolean statements (Brown et al. 2015). However, block-based tools generally score well on measures related to easing novice programmers' early struggles. A recent study applied three sets of heuristics to compare a block-based environment (Scratch) with two text-based languages, Java (in the Greenfoot environment) and Visual Basic, finding Scratch to have the fewest (or tied for the fewest) number of problems identified (Kölling and McKay 2016). These heuristics evaluate features of the programming tools such as engagement, clarity, learner-appropriate abstractions, and error avoidance.

Shifting to research evaluating block-based tools used in classrooms, we focus on Scratch (and Scratch derivatives) and Alice, as these two tools have the widest use in contemporary computer science education of the block-based environments listed previously. While both Alice and Scratch have been used in formal education environments, it is important to keep in mind that the two projects initially had different goals and different target age groups. Scratch, from its inception, was focused on younger learners and informal environments (Resnick et al. 2009), while Alice was targeted at more conventional computer science educational contexts and, as such, has been the focus of more initiatives to evaluate student learning of programming concepts (Cooper et al. 2000).

We begin by reviewing literature on Scratch, investigating its use as the language of choice in formal computer science environments. Ben-Ari and colleagues have conducted a number of studies on the use of Scratch for teaching computer science. Using activities of their own design (Armoni and Ben-Ari 2010), Meerbaum-Salant et al. (2010) concluded that Scratch could successfully be used to introduce learners to central computer science concepts including variables, conditional and iterative logic, and concurrency. While students did perform well on the posttest evaluation from this project, a closer look at the programming practices learners developed while working in Scratch gave pause to the excitement around the results. The researchers claimed that students developed unfavorable habits of programming, including a tendency for extremely fine-grained programming and incorrect usages of programming structures, as a result of learning programming in the Scratch environment (Meerbaum-Salant et al. 2011). Grover et al. (2015) conducted a classroom study with middle school learners, asking students to work through their Foundations of Computational Thinking (FACT) curriculum. The researchers found that students working with a block-based programming environment showed significant learning gains in algorithmic thinking skills and a mature understanding of computer science as a discipline (Grover et al. 2015; Grover et al. 2016), while also identifying misconceptions and challenges associated with introducing novices to programming (Grover and Basu 2017).

There is also a growing body of research on elementary-aged learners using Scratch and Scratch-inspired environments such as Snap! (Harvey and Mönig 2010), LaPlaya (Hill et al. 2015), and Scratch Jr. (Flannery et al. 2013). This work has identified design features early learners find helpful (e.g., using accessible language in the blocks (Harlow et al. accepted)), features that can be

challenging for learners (e.g., the inclusion of advanced mathematics concepts such as negative numbers and decimals (Hill et al. 2015)), and is starting to delineate a developmental trajectory for early programming instruction (Franklin et al. 2017). Other work looking at comparing block-based to text-based programming using Scratch has similarly found that Scratch can be an effective way to introduce learners to programming concepts, although it is not universally more effective than comparable text languages (Lewis 2010). Given Scratch's intention of being used in informal spaces and its emphasis on introducing diverse learners to programming, it is important to highlight Scratch's success in generating excitement and engagement with programming among novice programmers (Malan and Leitner 2007; Maloney et al. 2008; Tangney et al. 2010; Wilson and Moffat 2010).

Compared to Scratch, the Alice programming environment has a longer history of serving as the focal programming tool in introductory programming courses. Much of the motivation for using Alice in courses is based on findings that Alice is more inviting and engaging than text-based alternatives and improves student retention in CS departments (Johnsgard and McDonald 2008; Moskal et al. 2004; Mullins et al. 2009). Alice has also effectively been used by instructors who adopt an object-first approach to programming as it provides an intuitive and accessible way to engage with objects with little additional programming knowledge needed. Part of Alice's success and relatively widespread use is due to the fact that the creators of Alice have authored a number of empirically backed textbooks and curricula that can serve as texts for an introductory programming course (Dann et al. 2011; Dann et al. 2009).

A small but growing body of research is conducting systematic comparisons of block-based and text-based environments. In a study using the Snap! programming environment, Weintrop and Wilensky (2015) found that students perform differentially on questions asked in block-based form compared to the isomorphic text alternative. These differences were not universal, however, but instead were influenced by the concept under question, with students performing better on block-based questions related to conditional logic, function calls, and definite loops and finding no differences on questions related to variables, indefinite loops, and program comprehension questions. Another study investigating learning outcomes in isomorphic block and text environments found little difference in learning outcomes, but did report that students completed activities in the block-based environment at a faster rate (Price and Barnes 2015). This suggests that while the same learning can be achieved, it happens more quickly in block-based environments.

2.3 Representation and Learning

"The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities" (Dijkstra 1982).

As stated by the Turing Award–winning computer scientist Edsger Dijkstra in the above quote, 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. diSessa (2000) calls this material intelligence, arguing for close ties between the internal cognitive process and the external representations that support them: "we don't always have ideas and then express them in the medium. We have ideas *with* the medium" (diSessa 2000, p. 116, emphasis in the original). He continues: "thinking in the presence of a medium that is manipulated to support your thought is simply different from unsupported thinking" (diSessa 2000, p. 115). These symbolic systems provide a representational infrastructure upon which knowledge is built and communicated (Kaput et al. 2002). Adopting this perspective informs why it is so crucial to understand the relationship between a growing family of programming representations and the understandings and practices they promote.

In focusing on the relationship between the programming representations used in instruction and the learner, we take inspiration from similar work from the physics education community. Sherin (2001) investigated the use of conventional algebraic representations as compared to programmatic representations in physics courses and found that different representational forms have different affordances with respect to students learning physics concepts and, as a result, affects their conceptualization of the material learned. Sherin (2001) summarizes this difference as follows: "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" [p. 54].

Wilensky and Papert (2006, 2010) give the name "structuration" to describe this relationship between the representational infrastructure used within the 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 sometimes should, change as new technologies and ideas emerge. In their formulation of Structuration Theory, Wilensky and Papert document a number of restructurations, shifts from one representational infrastructure to another, including the move from Roman numerals to Hindu-Arabic numerals (Swetz 1989), the use of the Logo programming language to serve as the representational system to explore geometry (Abelson and DiSessa 1986), and the use of agent-based modeling to represent various biological, physical, and social systems (Wilensky et al. 2014; Wilensky and Rand 2014; Wilensky 2001). This work highlights the importance of studying representational systems, as restructurations can profoundly change the expressiveness, learnability, and communicability of ideas within a domain. The development and adoption of new programming modalities, representations, and tools demand that such analyses be conducted to better understand the effects of these emerging approaches to teaching, learning, and using ideas within the domain of computer science. Having reviewed relevant literature, we now continue with a description of the study design and methods.

3 METHODS

In this section, we present details on how, when, and with whom the study was conducted. We begin by detailing the design of the study, then present the two modes of the introductory programming environment used, then conclude this section with information about the participants and setting.

3.1 Study Design and Data Collection Strategy

This study uses a quasi-experimental setup with two high school introductory programming classes. The study follows each classroom for the first 5 weeks of a yearlong introduction to programming course. Each of the classes used a different variant of the same programming environment called Pencil.cc (a customized version of the Pencil Code environment). The difference between the two versions of the environment is in how programs are represented and authored. One class used a block-based interface, and the second used a text-based modality; further details about Pencil.cc are presented in Section 3.2. The study began on the first day of school with students in both classes taking preattitudinal surveys and content assessments.

The Commutative Assessment (Weintrop and Wilensky 2015) was used for the content assessment. The Commutative Assessment is a 30-question multiple-choice test that covers the concepts students encounter during the 5-week curriculum: variables, loops, conditionals, and functions. The assessment also includes algorithm and comprehension questions. Each question on the Commutative Assessment takes the form of a short program, followed by five multiple-choice options, and asks the question: "What will the output of the program be?" The key feature of the Commutative Assessment is that the short program snippets in each question are presented in one of three

(A) Snap! (B) Pencil Code Blocks (C) Pencil Code Text

Fig. 2. The three forms programs may take in the Commutative Assessment.

modalities: Snap! blocks, Pencil Code blocks, or text (Figure 2). The comprehension questions are like the content questions, except that they ask: "what does this program do?" and challenge the student to go beyond just output to figure out its behavior and potential uses. Finally, the algorithm questions were posed in plain text and asked students to identify the order of steps in an algorithm or identify potential missing steps. Each assessment included a mix of modalities so students answer questions in all three forms for each content area. There were three versions of the assessment given at each time point, so every question was answered in each modality at each time point. This counterbalance design ensures an even distribution of modality, concept, and condition (i.e., every question was answered in each modality by students from both the blocks and text classes).

The attitudinal survey was loosely based on questions from the Georgia Computes project (Bruckman et al. 2009) with specific questions being added for this study. It included 10-point Likert scale questions and short response questions. At the conclusion of the study, students again took the attitudinal survey and Commutative Assessment. The post-content assessment was composed of the same questions as the preassessment, just in a different order and with different modalities for questions, while the attitudinal survey was largely the same, with the exception of a few additional reflection questions. The surveys were administered online during class time on consecutive days so as to minimize testing fatigue. The attitudinal survey took students around 20 minutes and the content assessment took close to 25 minutes. The assessments were given on the same day in both classes.

A number of qualitative data sources were also gathered as part of this study to complement the quantitative data just discussed. The major qualitative data source for this study was semistructured clinical interviews with students, which are included sparingly to supplement the quantitative findings presented in this article. These interviews occurred outside of class time throughout the 5 weeks of the study. Full versions of both of these instruments, as well as interview protocols, can be found in Appendices B, C, and D of Weintrop (2016).

3.2 Pencil.cc and the Curriculum

This study used a custom-designed programming environment called Pencil.cc. Pencil.cc is based off of the Pencil Code environment (Bau et al. 2015). Pencil Code is an online tool for learning to program. Its interface (Figure 3) is split into two panes: on the left is the code editor, while the right side is a webpage that can visually run the program the learner creates. The unique feature of Pencil Code that made it an ideal choice for this study is that the editor supports both block-based (Figure 3A) and text-based (Figure 3B) authoring. The block-based interface provides a visual overlay on top of the text interface. This means the block-based and text-based versions of the programs are the same with respect to the actual characters and commands in the program, but the block-based form has the visual characteristics typical of block-based programming and supports drag-and-drop composition. In this way, the block mode and text mode are completely

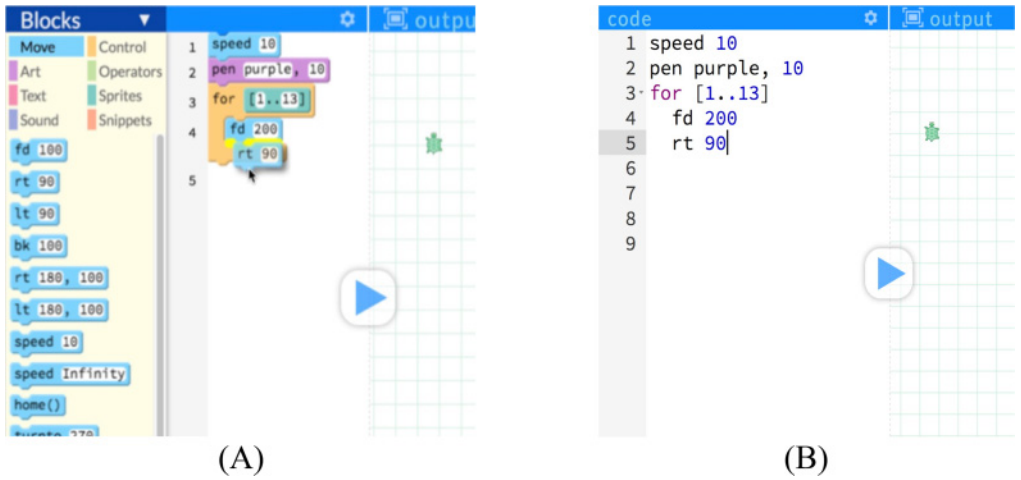

Fig. 3. The Pencil.cc interface. One class in this study used the block-based mode (A), while the other class used only the text-based mode (B).

isomorphic, and the two interfaces have the exact same semantics, syntax, and set of capabilities. The only thing that differs is how programs are composed and visually rendered. The major difference between Pencil Code and Pencil.cc is that in Pencil Code, the learner can freely navigate back and forth between the block and text modalities; however, given the questions being pursued in this study, Pencil.cc restricts students to a single modality, either the blocks modality or the text interface.

The blocks-only version of Pencil.cc includes many of the features that have been identified by learners as useful in block-based environments, such as the browsability of blocks in the palette and the ease of composition through the drag-and-drop interaction (Weintrop and Wilensky 2015b). Students in the text-only condition never saw the block-based interface and instead had to type all of their commands in character by character. The text editor does include syntax highlighting as well as basic compile-time error checking (this took the form of a red X to the left of the line number when students typed invalid commands). CoffeeScript was chosen as the programming language for this study as it is syntactically lightweight, a professional programming language lending authenticity to the activities, and supports Pencil Code's Turtle.js library, enabling Logo-style activities.

The 5-week curriculum for the introductory course is loosely based on the Beauty and Joy of Computing course (Garcia et al. 2015), along with an assortment of other introductory computing activities grounded in the constructionist programming tradition developed by Papert and others around the Logo programming language (Papert 1980; Harvey 1997). An emphasis of this design is giving students creative freedom within each assignment. Over the course of the 5 weeks, four major conceptual topics are covered: variables, conditional logic, looping logic, and procedures. Throughout the curriculum, care was taken to blend visually executing programs (like traditional Logo graphics drawing assignments or Scratch-style creative projects) and number or text processing activities that do not have a graphical component. Every concept had at least one visual and one text-only activity. The goal of this design was to not prioritize activities that are more conducive to one modality over the other. The curriculum was largely designed by the first author, but the teacher contributed ideas and lessons and customized the activities while teaching them. A full copy of the curriculum can be found in Appendix A of Weintrop (2016).

3.3 Setting and Participants

This study was conducted at a large, urban, public high school in a midwestern American city, serving almost 4,000 students. The school is a selective enrollment institution, meaning students have to take an exam and qualify to attend. In this school district, students are selected based on their performance on the admissions test relative to other students from their school (as opposed to all other applicants). As a result, students attend this school from across the city and there is equal representation of students from underresourced schools as from schools in more affluent neighborhoods. The student body is 44% Hispanic, 33% white, 10% Asian, 9% black, and 4% multiracial/other. A majority of the students in the school (58.6%) come from economically disadvantaged households, with the student body also including second language learners (0.6%) and diverse learners (4.5%).

The experiment was conducted in an existing Introduction to Programming course. Historically, the class spent the entire year teaching students the Java programming language. To accommodate the study, Java instruction began in the sixth week of school, after the conclusion of the 5-week curriculum presented earlier. Each class had 30 students and each student was assigned a laptop computer, which they used every day for the duration of the study. Students sat in individual desks that were on wheels that allowed them to move around the room. The same teacher taught both sections of the course in the same classroom in back-to-back periods (seventh period for blocks and eighth period for text), allowing us to control for teacher effects. The teacher holds an undergraduate degree in technical education and corporate training. The year she participated in the study was her eighth year of teaching, and third at this school.

The computer science course used for the study is an elective class but historically has attracted students from a variety of racial backgrounds and been taken by both male and female students. A total of 60 students participated in the study across all 4 years of high school (nine freshman, nine sophomores, 16 juniors, and 26 seniors). The self-reported racial breakdown of the participants was 41% white, 27% Hispanic, 11% Asian, 11% multiracial, and 10% black. The two classes in the study were composed of 11 female students (five in blocks, six in text) and 49 male students (25 in blocks, 24 in text). This gender disparity is problematic, but as recruitment for the courses was out of the control of the researchers, there was little that could be done to address this. Of the students participating in the study, almost half (47%) speak a language other than English in their households.

4 FINDINGS

Next we present our analysis of the data collected in this study. This section begins with a statistical analysis of student performance on the Commutative Assessment. These findings are then discussed with respect to modality and concept. Next, we shift to perceptions, beginning with perceived ease of use of the tools, then doing a systematic presentation of student responses to questions from the pre- and post-attitudinal survey.

4.1 Learning Outcomes by Condition

The first objective of this section is to show there is no difference between the two conditions in their performance on the first administration of the assessment that might skew later findings. On the pre-content assessment, the mean scores by condition were 54.3% (SD = 12.2%) for blocks and 51.6% (SD = 14.5%) for the text condition. Running a t-test on these two scores shows them to not be statistically different from each other: $t(57) = 0.78$, p-value = 0.44, $d = 0.20$ (note: when calculations show a number of participants below 60, it is due to student absence on the day of the survey or test). This lack of difference means that the two classes are not different from each other with

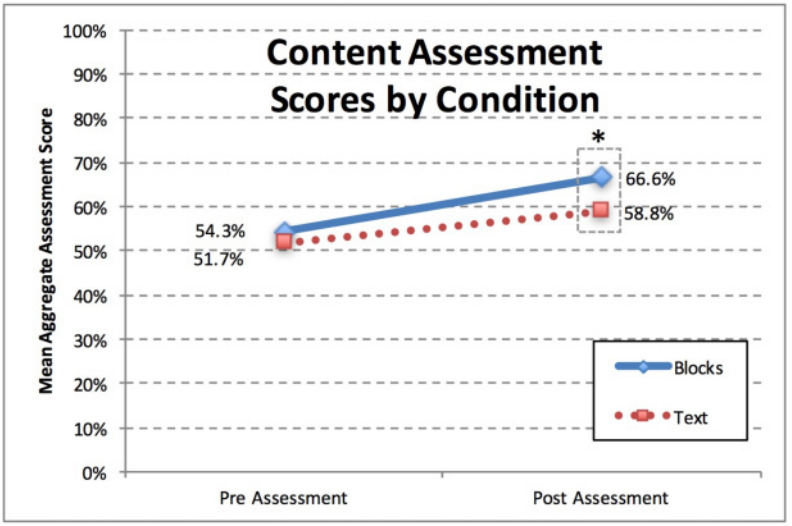

Fig. 4. Student Commutative Assessment scores by condition over time.

respect to their incoming programming knowledge. With that established, we now move forward with our analysis of learning gains by condition. Figure 4 shows average scores for students across the two conditions on the pre- and post-Commutative Assessment administrations.

On the postassessment, the mean score for the blocks condition was 66.6% (SD = 13.4%), while the text condition had a mean score of 58.8% (SD = 14.6%). The positive slope for both conditions between the pre- and postassessments means that, in aggregate, students in both classes performed better on the postassessment than they did on the preassessment. Given that this was an introductory class, this is not surprising, but still noteworthy and an encouraging sign given that these two conditions cover most of the modalities used to introduce learners to programming. For both conditions, the improvement on test scores from the pre- to the postassessment was significant (blocks: $t(24) = 6.11$, $p < .001$, $d = .96$; text: $t(26) = 3.70$, $p = .001$, $d = .50$). While the improvements are significant, the blocks condition saw a larger absolute gain.

Running a t-test comparing the scores on the postassessment reveals a statistically significant difference in scores between the two groups: $t(52) = 2.03$, p-value = 0.041, d= 0.58. This means that students who spent 5 weeks working in a block-based programming environment, in aggregate, performed better than students working in an isomorphic text-based environment. As a reminder, the study design controls for teacher effects, time on task, setting, and the set of activities used. This leads to the conclusion that the design of the programming interface affects student learning; in other words: modality matters.

4.1.1 Condition by Modality. To better understand the learning gains found in the previous section, we now take a closer look at the data to try and understand the source of these learning gains, in hopes of attributing these findings to the modalities used in the introductory learning environments. We do this by first looking at differences in outcomes by the modality of the question being asked, and then by looking at differences by concept. Figure 5 shows mean student scores on the postadministration of the Commutative Assessment grouped by modality and condition. In other words, for the two conditions (blocks and text), how did students perform on questions that presented their code as Pencil Code text (Figure 2C), Pencil Code blocks (Figure 2B), and Snap! blocks (Figure 2A)?

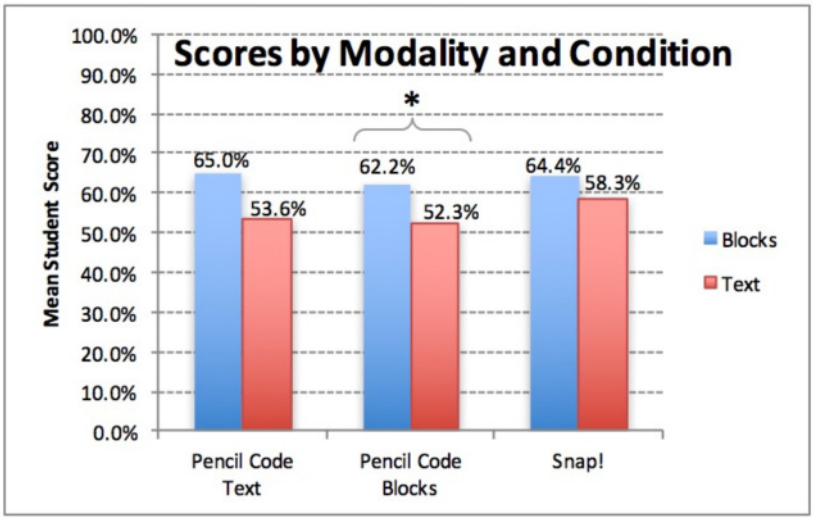

Fig. 5. Student scores on the Commutative Assessment grouped by modality and condition.

The first thing to note from Figure 5 is that across all three modalities that questions were asked, the blocks students outperformed the students who had used the text-based interface. Of these three modalities, only the difference on the Pencil Code blocks questions is statistically significant (Pencil Code blocks: $t(52) = 2.72$, $p = .008$, $d = .73$; Pencil Code text: $t(52) = 1.52$, $p = .134$, $d = .41$; Snap! blocks: $t(52) = 1.17$, $p = .246$, $d = .32$). Despite the lack of significance across all three groups, we can still draw some conclusions from this data. First is the fact that the blocks condition outperformed the text condition in all three modalities, including, most surprisingly, the Pencil Code text questions. In other words, students in the blocks condition scored higher on a modality they had not used, compared to students who had just spent the previous 5 weeks using that modality. This finding suggests that the understanding that forms in one modality is not tightly coupled to that modality. An alternative interpretation of this finding is that the ability to make sense of programs developed in the block-based modality is not tightly coupled to that modality. This suggests a potential form of near transfer from the block-to-text modality, similar to findings from related work (Grover et al. 2015). A second conclusion we can draw from this data is that students performed comparably across the three modalities, as opposed to seeing a pattern where students perform the strongest in the modality they have been using most recently. This is a second piece of data showing that learners' emerging understandings are not tightly bound up with the modality they used. Finally, the fact that students in the text condition scored higher on the Snap! block questions suggests there are additional affordances to the Snap! presentation of blocks over the Pencil Code blocks, some of which can be seen in Figure 2, like the natural language commands and a larger variety of slots and block shapes. A further investigation of the Snap! versus Pencil Code blocks difference is outside of the research questions being pursued in this article but is an intended avenue of future research.

4.1.2 Condition by Concept. The next analysis of this data investigates differences in conceptual understanding by condition. In doing so, we answer the question: does modality affect how students learn specific concepts? In other words, are certain concepts more easily learned through working in one modality versus another? Figure 6 shows student performance across the six concepts assessed on the post-Commutative Assessment.

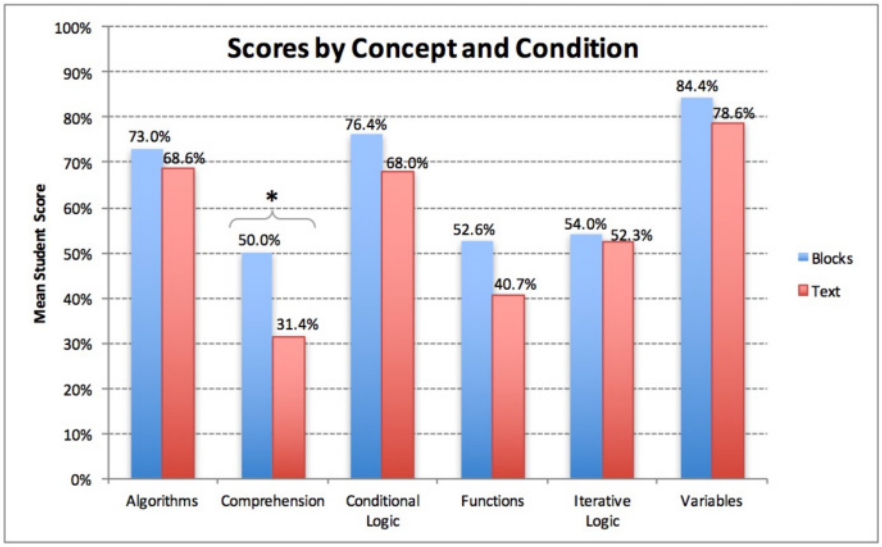

Fig. 6. Student performance on the midpoint administration of the Commutative Assessment grouped by condition and concept.

As with the previous analyses looking at performance by modality, when looking at performance by concept, the students in the blocks condition, in aggregate, outperform students in the text condition in every category. Running a t-test calculation on each concept category shows only a significant difference between the two conditions for the comprehension questions: $t(52) = 3.21$, $p = .002$, $d = 86$. The lack of significant differences in the other five categories, but the pattern of the blocks condition scoring higher than the text condition, suggests that learning in a given modality does not help with a specific concept as much as it is generally useful across all concepts. The finding that students scored particularly low on the comprehension questions echoes similar work looking at the relationship between modality and conceptual understanding (Weintrop and Wilensky 2015) and matches prior work on students' difficulties in drawing larger meaning and purpose when reading programs (Robins et al. 2003). That the scores on these questions were the most different suggests that comprehension may be one place that learning with a specific modality may be helpful. One possible explanation is that when composing programs with block-based tools, the learner is working with compositional units that match larger cognitive building blocks (the command itself) as opposed to needing to assemble those blocks one character at a time. As such, the blocks modality requires less cognitive effort to be expended on the implementation of that idea, giving the learner more practice thinking at a conceptual level.

4.2 Perceived Ease of Use of Concepts by Condition

Along with actual performance on the Commutative Assessment, we are also interested in how students perceive the ease of use of the two modalities. Specifically, we are interested in how students view each concept in the two modalities. The goal of this line of inquiry is to understand if some concepts are viewed as easier in one modality versus the other, and then, whether or not these perceptions match students' performance on the content assessment. This shift toward perceptions serves as a bridge between the previous section on learning outcomes and the sections to follow on attitudinal outcomes from the study. On the attitudinal assessment given at the conclusion of

Table 1. Distribution of Ease-of-Use Responses

		Variables		Loops		Conditional Logic		Functions	
		Blocks	Text	Blocks	Text	Blocks	Text	Blocks	Text
Likert Response	1 (Very Easy)	13	10	11	3	14	8	5	2
	2	7	4	6	7	8	6	6	5
	3	4	5	3	9	2	6	9	7
	4	2	4	3	4	0	2	3	6
	5	0	2	1	3	2	3	1	2
	6	1	2	3	1	0	2	1	3
	7 (Very Hard)	0	1	0	1	1	1	2	3

the study, there were a series of 7-point Likert scale questions about perceived ease of use of the various programming concepts covered. The distribution of responses to the Likert questions is shown in Table 1. Each cell in the table reports the number of students that chose that Likert scale value.

For all four conceptual categories, students in the blocks condition viewed the concept under question as easier to use than students in the text condition, as can be seen by the larger quantities in the blocks columns in the top rows of Table 1. Running a Mann-Whitney-Wilcoxon test for each conceptual category finds two of the four concepts to be statistically different between the two groups: Conditional Logic, $U = 252.5$, $p = .03$, and Iterative Logic, $U = 266$, $p = .05$ (Functions $U = 275$, $p = .08$, and Variables $U = 283$, $p = .10$ fail to reach statistical significance at the $p = .05$ level). Due to the study design, these aggregate differences between blocks and text conditions are best explained by the modality itself. This means that students found using conditional logic and iterative logic in the drag-and-drop blocks modality to be easier than the all-text condition.

Comparing Table 1, which shows the ease of use of concepts, and Figure 6, which shows scores by concept and modality, for the four concepts that overlap, there is a correlation between how easy a concept is perceived to be and how well students did on questions on that topic. Running a Spearman rank-ordered correlation returns a value of $r_s = .81$, showing a high correlation between students' perceived ease of use of a concept and their performance on the assessment of that concept. This suggests that students' own perceptions of ease of use match their ability to answer questions about that concept.

4.3 Perceptions and Attitudes by Condition

To understand how students' attitudes and perceptions were affected by the modality, we turn to responses from the attitudinal assessment. This survey included a number of questions asked on a 10-point Likert scale. In this section, we look at four dimensions of students' attitudes and perceptions of computer science: confidence, enjoyment, perceived difficulty, and interest in future computer science learning opportunities. Please note, in this section, we report both significant and nonsignificant findings as there are times the lack of significant results is counter to commonly held beliefs. When comparing the blocks and text conditions to each other, a Wilcoxon Rank Sum test is used (reported as a U statistic). This test is appropriate as the two samples are independent and the underlying data is nonparametric and ordinal in nature. In cases where the analysis looks at changes within a group between the pre- and postsurveys, a Wilcoxon Signed Rank test is used (reported as Z statistic). This test is appropriate given the ordinal nature of the Likert responses and because it is a nonparametric test used to compare paired samples.

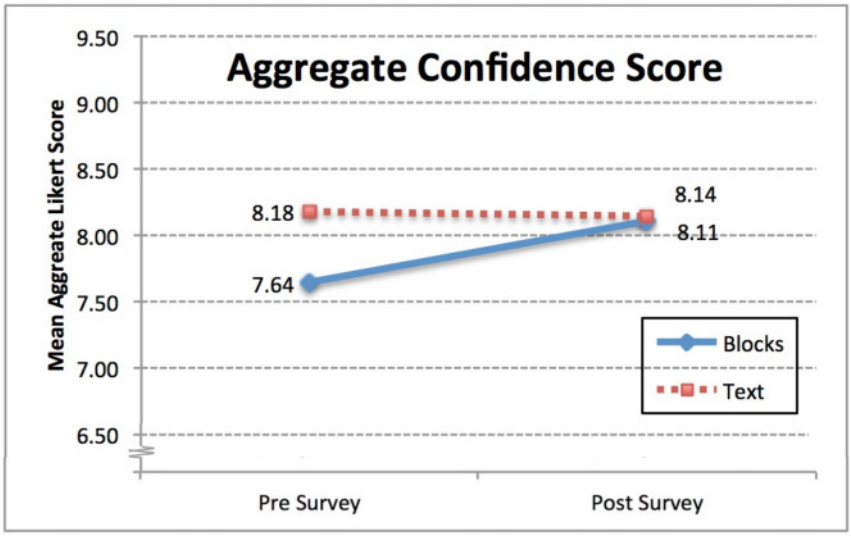

Fig. 7. Mean aggregate confidence score for students by condition.

4.3.1 Confidence in Programming Ability. To calculate a reliable measure of confidence, student responses to the following two Likert scale statements were averaged together: I will be good at programming (or I am good at programming on the posttest) and I will do well in this course. These questions show an acceptable level of correlation, having Cronbach's α scores of .82 on the presurvey and .80 on the postsurvey, which are right at the .8 threshold commonly used to define an acceptable level of reliability. The aggregated confidence measure at the pre- and post-points in time are shown in Figure 7. Please note that all figures in this section are on the same scale but do not cover the same range, so they can be compared relatively, but not absolutely. Also, please note that the y-axis on each of these figures does not start at 0; this is done to make the chart more readable.

The mean confidence scores at the outset of the study was M = 7.92 (SD = 1.47) on a 10-point Likert scale, which is rather high and can be attributed, in part, to the fact that this was an elective course. The difference in scores between conditions on the presurvey is not statistically significant (U = 307.5, p = .09). After spending the first 5 weeks of school working in Pencil.cc, the overall average confidence scores inched up to M = 8.13 (SD = 1.62), a change that is also not statistically significant (U = 268.5, p = .08). Looking at differences by condition, we find no statistically significant difference in levels of confidence between students who spend 5 weeks working in the text-based version of the environment versus the block-based interface (U = 395, p = .78). Focusing on the changes in levels of confidence between the pre- and posttest within the two conditions, we do see a significant gain for the blocks condition (Z = 46, p = .05), but not for the text condition (Z = 98.5, p = .82). Given the positive slope of the change in the blocks condition, this difference can be interpreted as showing that students in the blocks condition saw a significant increase in their confidence in their own programming abilities, which cannot be said for students in the text condition. This outcome matches prior working comparing confidence in block-based and text-based environments (Price and Barnes 2015).

The significant change in confidence for the blocks condition is consistent with other noncomparative studies that suggest that the block-based programming interface is effective in increasing students' confidence in their own programming ability (Maloney et al. 2008; Smith et al. 2014). The

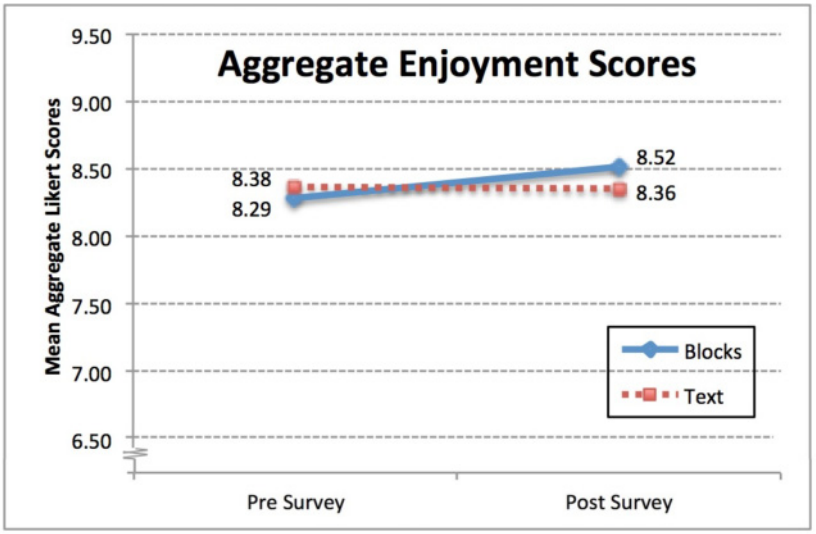

Fig. 8. Aggregate levels of students' enjoyment of programming by condition at three points in the study.

lack of a positive trend for the text condition can be interpreted in a few ways. One explanation is that the text modality does not improve students' confidence in programming, for which a number of possible explanations could be given (e.g., they find it difficult or did not feel successful in their time with it). A second plausible explanation for these data is that there was a ceiling effect, meaning the students started with a high level of confidence, so there was little room for them to become more confident, which was less the case in the blocks condition. As with other conclusions drawn from the study, this finding indicates that future work needs to be done with students who have less initial confidence in their programming ability.

4.3.2 Enjoyment of Programming. The second attitudinal dimension is whether or not students' enjoyment of programming differed based on the modality they used. To calculate a measure of enjoyment, responses to the following three Likert statements from the pre- and postsurveys were averaged I like programming, Programming is fun, and I am excited about this course. These three questions were found to reliably report the same underlying disposition at both time points (pre-Cronbach's $\alpha = .88$, post-Cronbach's $\alpha = .80$). Figure 8 shows the aggregated enjoyment scores for students across the two conditions at the beginning and end of the study.

As can be seen by Figure 8, there was little difference in students' enjoyment of programming based on modality. There was no statistical difference between the two conditions at the outset ($U = 399$, $p = 0.75$) or the conclusion of the study ($U = 408$, p-value $= 0.62$). Likewise, neither condition saw a significant difference in their reported levels of enjoyment between the pre- and post-surveys (blocks: $Z = 70.5$, $p = .20$; text: $Z = 93.5$, $p = .68$). This lack of significant finding by condition suggests that modality plays a relatively small role with respect to perceived enjoyment of programming. An alternative explanation is that some other characteristic of the class, such as the teacher or curriculum, played a much larger role in terms of student enjoyment. Either way, this finding suggests that the narrative of block-based programming being more fun than text-based programming does not hold for high-school-aged learners in formal classrooms.

4.3.3 Programming Is Hard. The attitudinal survey included the Likert statement: Programming is hard. Initially, this was intended to be part of the confidence aggregate score, but ended up not

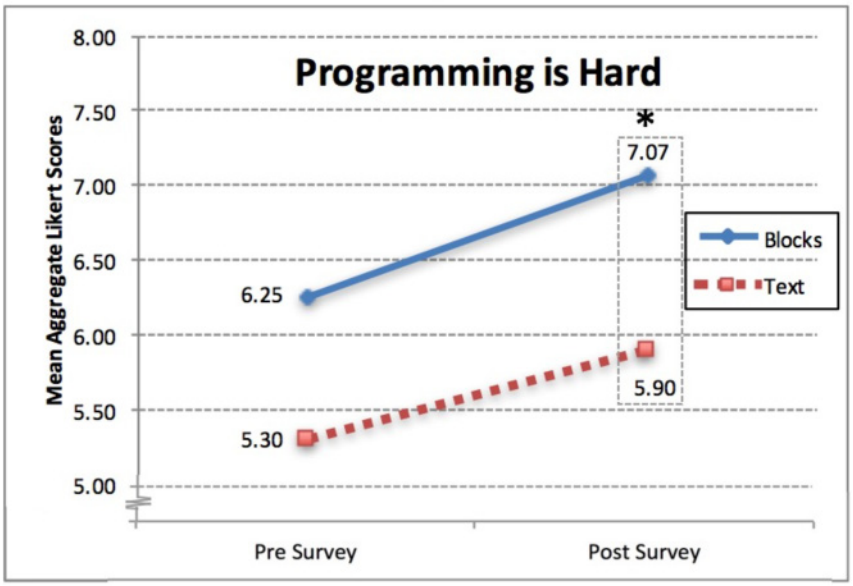

Fig. 9. Average responses to the Likert statement: Programming is hard.

correlating with the other two confidence questions (pre-Cronbach's α = 0.48, post-Cronbach's α = 0.57) so is treated independently. Figure 9 shows the pre- and postscores for students grouped by condition for this question.

Unlike the last two questions, on the question about perceived difficulty of programming, we see a significant difference between the two conditions after the 5-week curriculum. On the preattitudinal survey, we find a difference between the two conditions' average reported score, but not one that reaches statistical significance (U = 521, p = .11). On the postsurvey, that gap between the two conditions grows to reveal a statistically significant difference in perceived difficulty (U = 524.5, p = .01). Looking at changes within the conditions, both groups saw programming as harder on the postsurvey, but only the blocks condition's responses were statistically significant (blocks: Z = 42, p = .02; text: Z = 96, p = .51).

This finding is interesting given the fact that students in the blocks condition performed significantly better than students in the text condition on the post-content assessment. In other words, those who did better on the posttest also thought programming was more difficult. This means that this perceived difficulty of programming does not match the blocks students' performance on the assessments. One possible explanation for this outcome is that students see a difference between what they were doing in the block-based interface and the "programming" that the question is asking about. It is worth noting that this view might not be specific to modality as the text condition also viewed programming as more difficult after programming in text in Pencil.cc for 5 weeks, despite improved scores on the content assessment. This suggests the perception is also shaped by other aspects of the environment, including the graphical execution and the browser-based interface, both of which differ from the conventional view of programming.

4.3.4 Interest in Future CS. The last attitudinal category we present looks at whether or not the modality used in the introductory programming environment affected students' interest in enrolling in future computer science courses. More specifically, students were asked, on a 10-point

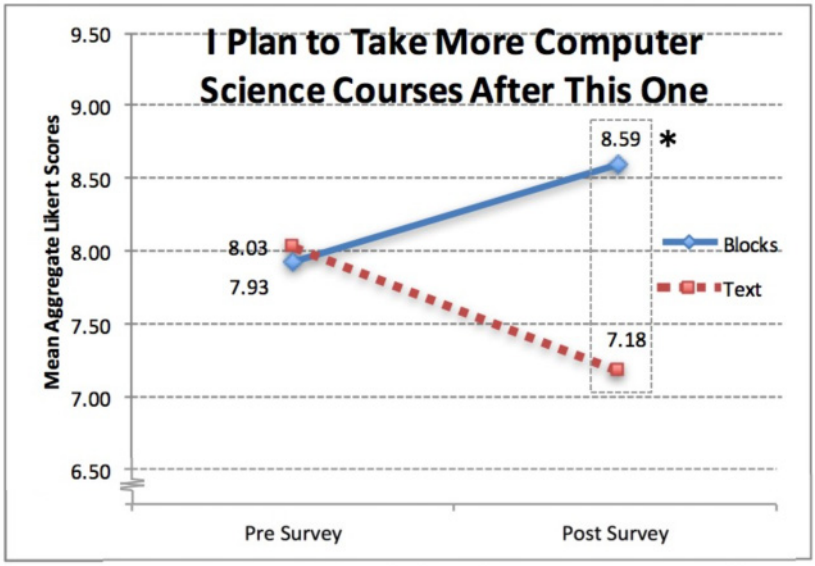

Fig. 10. Average responses to the Likert statement: I plan to take more computer science courses after this one, grouped by condition.

scale, how much they agreed (10) or disagreed (1) with the following statement: I plan to take more computer science courses after this one. Figure 10 shows the average response for students grouped by condition.

At the outset of the study, there was no difference between the two conditions with respect to interest in taking future computer science courses ($U = 444.5$, $p = .70$). Students in both groups reported a relatively high interest in future coursework in the field, with students in the blocks condition reporting an average of $M = 7.93$ ($SD = 3.1$) out of 10 and text students reporting an average of $M = 8.03$ ($SD = 2.2$) out of 10. After 5 weeks working in the Pencil.cc environment, student-reported levels diverge, with blocks students reporting a higher level of interest in future courses ($M = 8.59$ out of 10, $SD = 2.1$), while text students were less interested ($M = 7.18$ out of 10, $SD = 2.7$). This difference in average scores is significant ($U = 491.5$, $p = .04$). Looking within the conditions, neither the positive change for blocks students nor the negative change for text students is statistically significant at the .05 level (blocks: $Z = 25$, $p = .15$; text: $Z = 110$, $p = .29$).

This result shows that students became more interested in computer science after working in a block-based programming environment and less interested in computer science after working in a text-based interface. After starting with a similar predisposition to future programming coursework at the start of the study, the two student populations had different levels of interest after working in the different modalities. This suggests that modality does have an impact on students' interest in future computer science coursework. This finding matches other work showing that block-based approaches to programming can help with student retention (Moskal et al. 2004) and has implications with respect to creating environments that are intended to broaden participation in computing and encourage students to start and continue with computer science instruction.

4.4 Perceptions of Introductory Environments

Along with pre-/postsurvey questions pertaining to student-held attitudes toward programming, the postsurvey also included a pair of questions asking students to reflect on the effectiveness and

authenticity of the introductory environment they used during the 5-week study. After 5 weeks of working in Pencil.cc, students were asked respond to the following prompt: Pencil.cc is similar to what real programmers do. Responses were given on a 10-point Likert scale. The intention of this question was to understand perceived authenticity of the two environments, as previous work on block-based programming with high school students has found authenticity to be a drawback recognized by some high-school-aged learners (Weintrop and Wilensky 2015b). For students working in the text condition, the average reported response to this question was M = 6.71 (SD = 1.82), while students in the blocks condition had an average response of M = 5.89 (SD = 2.22). Despite the difference in values, a Wilcoxon Rank Sum test does not show these two scores to be significantly different from each other: U = 448.5, p = .23. The survey also asked students how effective they found Pencil.cc to be in terms of improving their programming ability. Specifically, the 10-point Likert prompt read: Pencil.cc made me a better programmer. The average response for students in the text condition was M = 7. 79 (SD = 1.85), while students in the blocks condition had a slightly lower average score: M = 7.44 (SD = 2.23). These two scores are also not statistically different from each other (U = 400, p = .71).

While these numbers do not report statistically significant findings, by drawing on other data sources, particularly the free response questions on the postsurvey, we can get a fuller sense of whether students' perceived authenticity and effectiveness are salient for some students. For example, one student, in reflecting on his time with the introductory environment and projecting forward to what was to come, said: "I feel like Java will be more useful in the long run than what (Pencil.cc) could offer me." This view was echoed by another student who said in his post interview: "(Pencil.cc) is a bit too limiting for someone who goes into this class thinking I'm going to make something that is going to be used in industry." In these quotes, the students' long-term plans with programming can be seen and how Pencil.cc does not necessarily fit into them. Taken together, these data suggest differences may exist, particularly with respect to the question of authenticity, but these views may not be universally held. Instead, students with more prior experience or more serious intentions for future computer science instruction appear to be more critical of the authenticity and utility of block-based introductory tools. The takeaway from this analysis is that perceptions of modality by high school students appear to differ by student, but more work needs to be done to understand this dimension of student perceptions of introductory tools.

5 DISCUSSION

The first contribution from this study is the finding that students using a block-based modality showed significantly higher learning gains after 5 weeks of classroom instruction compared to their text-based peers. It is important to keep in mind that for this study, students worked through the same curriculum, with the same teacher, in the same classroom, and had the same time on task, meaning that, as much as possible, external factors beyond the modality were controlled for. Digging into this finding revealed a consistent pattern of students in the blocks condition outperforming their text-based peers. When looking at performance by the modality of the question being answered, students from the blocks condition performed better for all three code formats (Pencil Code blocks, Pencil Code text, and Snap! blocks). This was surprising as it meant the students in the blocks condition did better on the Pencil Code text questions than the students who had exclusively been using the text interface of Pencil.cc for the previous 5 weeks. Likewise, the blocks condition did better on the Snap! questions, which used an interface neither condition had seen. There are a few possible ways to interpret these numbers. One interpretation is that there is some form of near transfer occurring from the Pencil.cc blocks interface both to another blocks interface (Snap!) and to a similar (or syntactically identical) text interface (Pencil Code text). A slightly different interpretation is that the learning that happened by students in the blocks condition is

not so tightly coupled to the interface that it cannot be used across languages. The distinction between these two interpretations is whether the learning that occurred was about the modality or the underlying concepts. This suggests one direction for future work that will be discussed later.

Just like with the questions-by-modality finding, students in the blocks condition outperformed their text counterparts in all six content categories on the postassessment. This means that the utility of learning to program in a block-based interface is not confined to one specific concept or another. This is further evidence in support of the finding that block-based programming does provide learning supports for novices in introductory contexts. That last clause "introductory contexts" is important, as we do not yet know how these advances will support novices as they transition to more conventional programming languages like Java or Python. This means that stronger claims about the power of block-based tools beyond the context and programming environment in which they are situated cannot yet be made.

This article also reports on learners' perceptions of ease of use of programming constructs by modality. Here again the blocks condition outperformed the text condition across all four concepts, meaning that students in the blocks condition found various programming concepts easier to use than students who had used the text-based interface. This suggests that the benefits of block-based interfaces extend beyond comprehension to include compositional dimensions of programming. The pattern observed for questions relating to ease of use did not persist as we investigated other attitudinal questions, where the findings were less clear.

Our analysis of attitudinal survey results found that, after spending 5 weeks working in the introductory modalities, students using the block-based interface reported higher levels of interest in future computing courses as well as a higher reported score for the perceived difficulty of programming. On questions of enjoyment and confidence, we found no difference by modality. The finding of no difference in confidence and enjoyment runs counter to the traditional narrative and some prior work (Lewis 2010), a difference that could potentially be explained by the shift in age (middle school to high school) as well as setting (informal to formal contexts).

Taken together, the findings that students using a block-based modality performed better on content assessments and also showed a higher level of interest in taking future computer science classes suggest that block-based tools are a productive strategy for introducing learners to the field of computer science. At the same time, the finding that students in the blocks condition viewed programming as more difficult suggests that a gap still exists between what the students view themselves as doing in the block-based tools and the broader world of programming—a gap that will be confronted should they continue in the field of computer science.

5.1 Implications for Learners with Differing Prior Experience

A complicating aspect of modality choice in formal education spaces is the fact that students are entering their first computer science learning opportunities with an increasingly diverse set of prior programming experiences. In this study, some students had never programmed before, while others had just spent the summer trying to learn professional software development frameworks. Given that all students in the same class usually learn with the same environment and are asked to complete the same set of assignments, keeping advanced learners engaged while also not leaving true novices behind is a challenge. Modality choices made to support one type of learner may negatively affect the other. This came up a few times in this study, when advanced students lamented having to use a block-based modality, instead wanting to go straight into learning Java. Modality does not inherently make a language more or less powerful; instead, it just shifts how one interacts with it. However, as this study shows, modality does influence perceptions, suggesting that work does need to be done to engage more advanced students should they be asked to use block-based tools for instructional purposes. Further, much of computer science is less concerned with syntax

and details of a programming language and instead focuses on issues related to problem solving and critical thinking. Modality choice directly impacts learners, but through framing and carefully selected activities, the drawbacks of advanced learners using modalities designed for novices may be mitigated without sacrificing the benefits they hold for the novices they were designed for. This suggests that pedagogy plays a key role in framing the introductory tool as well as shifting focus from surface features of the environment or language toward the underlying concepts and leads to another audience impacted by this choice, the teacher.

5.2 Implications for Teachers

The choice of modality will have a large impact on the experience of the teacher and their experiences in the classroom. Modality can influence classroom culture and pedagogical approaches and, in part, shapes the curriculum that is followed (Weintrop and Wilensky 2016). In choosing a given modality, the teacher is defining various aspects of the course and his or her own position in it. As just discussed, modalities designed to support novices in programming independently will impose different challenges on the teacher compared to a modality with fewer beginner-oriented features. A teacher's preference for direct instruction versus letting learners discover and explore on their own should be taken into account when choosing a modality. When working in a modality designed for beginners, the learners' reliance on the teacher for guidance is decreased, and thus the teacher can spend more time in one-on-one support. At the same time, if students are better able to make progress on their own, there is less potential for teachable moments—instances when students ask questions that lead to productive class discussion. This point was made salient by the teacher who, in reflecting on her experience teaching in the block-based modality, said: "the point of the environment is that it shouldn't generate a whole lot of questions, like 'how do I do this?'—it's more intuitive." The teacher went on to explain that while this is empowering for the learner, it results in fewer opportunities to engage in student-prompted productive discussions on different aspects of programming.

Just as modality choice shapes the role of the teacher in the classroom, it can also shape the curriculum. Modalities designed to facilitate exploration and creativity allow for different types of assignments compared with modalities designed for efficiency or clarity. If a teacher prefers every student to author a program that looks the same, choosing a modality that makes discovery easy may prove counterproductive to the teacher's desired form of assignment. There are also class management and grading considerations in choosing a modality. If assignments are open ended or assigned in a modality that makes it easy for students to go beyond what has been covered in class, the teacher is more likely to encounter a variety of solutions or solutions that include extra features beyond what was asked. This was a frequent occurrence during this study, especially among more advanced students who sought to challenge themselves on assignments they were able to complete quickly.

Along with impacting students and the role of the teacher, modality can also shape classroom culture. As our teacher pointed out: "(Block-based programming) creates a different feel to the room … Blocks take away the foreign feel, it looks friendly, and it's something you can do right away, and because of that, the culture in the room is different, kids are more prone to talk to their neighbors, more prone to feel OK about joking around." While modality is not the only contributor to a classroom culture, more inviting and playful tools can help shape a certain set of classroom norms.

A final, potential afterthought for a teacher in choosing a modality is considering the larger technological infrastructure of the class. Are assignments going to be submitted in a specific online format? Is the teacher planning on running all of the students' programs to make sure they work and meet the requirements of the assignment? The environments used in the introductory portion

of this class were all browser based, which made it tricky for students to submit their work as they did not have a local copy of their program to submit to the teacher. Instead, the teacher did her grading by walking around the room asking students to show her their work. While this worked for the purposes of this teacher, it had its limitations as the teacher could only spend a few seconds on each program and did not have a way to give detailed or written feedback to students.

5.3 Limitations and Future Work

While this study provides insights into the relationship between modality and learning in the domain of computer science, it does have limitations with respect to the claims that can be made. This section reviews the limitations and discusses potential future directions that may be taken to try and address them.

The first limitation of this study relates to the specific languages and materials that were used. For this work, Pencil Code's blocks editor was used to represent the block-based modality, while its text interface served as the canonical text editor. In the case of the blocks editor, other block-based modalities, like Scratch and Snap!, include additional features not supported by Pencil Code. This includes displaying different-shaped slots for each argument type (i.e., ovals for numerical inputs and hexagons for Boolean inputs) and having text labels closer to natural language (as can be seen in Figure 2). Likewise, Pencil Code's text editor included some built-in scaffolds like syntax highlighting and automatic indentation, but not others that are common in text-based coding tools like auto-complete. Also, the choice of CoffeeScript to serve as the underlying language in the introductory condition is only one of many possible ways such instruction could take place. In choosing one tool or language over another, we are naturally constraining the generalizability of the findings, but this study makes a contribution toward a more complete understanding of the block-based modality. The exploration of different languages and different underlying syntaxes is a natural next step and is work that is actively being pursued. Just as the choice of language and specific modality influence the findings, so too does the programming paradigm used and the design of the curriculum. The fact that roughly half of the introductory assignments relied on drawing or Logo-inspired Turtle Geometry activities in some capacity does change the way students interact with the modality. This fact necessarily constrains the generalizability of this study to this set of materials used (or similar materials). Further work is needed to generalize the findings beyond the specifics of this study.

The second limitation of this study is related to the students who participated in the two conditions. The school where the study took place was a selective enrollment institution. This means that all of the students who participated in the study have historically been successful in formal educational contexts. Thus, the findings of this article do not necessarily apply to underperforming students who have not had success in conventional classroom settings. A second, similar limitation is the fact that this study took place in an elective class. This means the students who participated in the study had chosen to take part in a computer science learning opportunity, suggesting they showed a predisposition for being more interested or placed a higher value on the concepts being taught. The effects of this decision can be seen in the relatively high values reported on the preadministration of the attitudinal survey. A final participant-related limitation of the study has to do with the gender breakdown of the study. In this study, female students made up less than one-third of the students in the class. The gender breakdown was beyond the control of the researchers as student recruitment for the classes was outside of the scope of the study, but is nonetheless not representative of the greater student population. All three of these limitations can be addressed by conducting future iterations of the study at different schools where these limitations are not necessarily true. In some school districts around the country, computer science is becoming a graduation requirement for high school. Conducting a similar version of this study

at a nonselective enrollment school where all students must take the class would directly address all of these limitations and is one intended future direction for this work.

A final set of limitations of this study relate to the teacher. Finding a teacher who was willing to teach the same curriculum using different modalities was difficult. Any teacher willing to take on such a challenge will have a level of confidence and experience that is rare among in-service computer science teachers. Understanding how modality affects less experienced and less confident teachers is an open and important question to answer. Just as the way to address the student-related limitations of this study was to replicate the study at a different site, the solution to the generalizability of the findings due to the teacher can similarly be addressed this way. Working with a less accomplished and experienced teacher (or set of teachers) is another direction of future work that goes hand in hand with working with a different population of students and is an intended avenue of future work.

Along with the future work directions suggested by the limitations of this study, there are also other major outstanding questions to answer. First among them is the question of if and how these tools helped prepare learners for the transition to text-based languages. Does the fact that students in the block-based condition performed better on the content assessment mean they will continue to perform better in a professional text-based language like Java? Alternatively, do students who spend time working in a low-threshold text-based tool have an easier transition to a language like Python? Work has started to investigate this question of transitioning from blocks to text (e.g., Dann et al. (2012) and Armoni et al. (2015)), but many questions remain. A related question speaks to the potential of hybrid block/text programming tools. A growing number of tools allow learners to program in either block-based or text-based interfaces (Matsuzawa et al. 2015; Bau et al. 2015; Homer and Noble 2014), while others seek to blend blocks and text features into a single tool (Kölling et al. 2015; Mönig et al. 2015). How do these approaches compare to single-modality environments? A parallel question to these is how does the tool relate to pedagogy and curriculum? How can teachers and curriculum developers take advantage of the affordances of tools designed to be intuitive and accessible to novices? These are questions that we and others are actively pursuing.

A final avenue of future work is to look at the impact of modality on different types of students. Do struggling students see more, less, or different benefits from working in a specific modality compared to students who have excelled in academic settings? We have some data to start to answer this question, but given the relatively small sample size of each condition, further divisions of the students into subpopulations leaves us with little statistical power to make the claims we hope to. As such, we hope to address this as we seek to replicate this work on a larger scale.

6 CONCLUSION

This article presents a systematic, classroom-based, comparative study of how programming modality impacts learners. It shows how modality affected students' attitudes, perceptions, and conceptual learning. Thus, it supports the claim that modality has a direct impact on learners' experiences with programming and their early computer science classroom learning outcomes. Understanding the relationship between modality and learning is consequential with respect to deciding what tools to use in classrooms and to inform the design of future introductory programming environments. Given the increasing presence of computer science in K-12 education and the growing ecosystem of educational programming environments and curricula, findings from studies such as this one are essential to ensure we are best serving the current generation of learners. While many open questions remain and there is much work to do, this study helps to fill in one piece of the larger puzzle on how best to introduce today's students to essential computing concepts. As enrollment in computer science learning opportunities grows, it is our hope that research

such as this continues to advance our understanding of the relationship between environment and learner, and that those findings can inform the next generation of tools, curricula, and classroom practice. In taking on this challenge, we can prepare learners for the computational future that awaits them, in the classroom and beyond.

REFERENCES

H. Abelson and A. A. DiSessa. 1986. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press.

M. Armoni and M. Ben-Ari. 2010. *Computer Science Concepts in Scratch*. Rehovot, Israel: Weizmann Institute of Science.

M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. 2015. From scratch to "real" programming. *ACM Trans. Comput. Educ. TOCE* 14, 4 (2015), 25:1–15.

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

D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens. 2015. Pencil code: Block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC'15)*. New York: ACM, 445–448.

D. Bau. 2015. Droplet, a blocks-based editor for text code. *J. Comput. Sci. Coll.* 30, 6 (2015), 138–144.

A. Begel. 1996. *LogoBlocks: A Graphical Programming Language for Interacting with the World*. Cambridge, MA: Electrical Engineering and Computer Science Department, MIT.

A. Begel and E. Klopfer. 2007. Starlogo TNG: An introduction to game development. *J. E-Learn.* (2007), 1–5.

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

P. Bontá, A. Papert, and B. Silverman. 2010. Turtle, art, turtleart. In *Proceedings of Constructionism 2010 Conference*.

N. C. C. Brown, M. Kolling, and A. Altadmri. 2015. Lack of keyboard support cripples block-based programming. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 59–61.

A. Bruckman, M. Biggers, B. Ericson, T. McKlin, J. Dimond, B. DiSalvo, M. Hewner, L. Ni, and S. Yardi. 2009. Georgia computes!: Improving the computing education pipeline. *ACM SIGCSE Bull.* 41, 1 (2009), 86–90.

S. Cooper, W. Dann, and R. Pausch. 2000. Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll.* 15, 5 (2000), 107–116.

W. Dann, S. Cooper, and B. Ericson. 2009. *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Prentice Hall Press.

W. Dann, S. Cooper, and R. Pausch. 2011. *Learning to Program with Alice*. Prentice Hall Press.

W. Dann, D. Cosgrove, D. Slater, D. Culyba, and S. Cooper. 2012. Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, 141–146.

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

A. A. diSessa. 2000. *Changing Minds: Computers, Learning, and Literacy*, Cambridge, MA: MIT Press.

V. Donzeau-Gouge, G. Huet, B. Lang, and G. Kahn. 1984. Programming environments based on structured editors: The MENTOR experience. In D. Barstow, H. E. Shrobe, and E. Sandewall, eds. *Interactive Programming Environments*. McGraw Hill.

L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick. 2013. Designing ScratchJr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*. ACM, 1–10.

D. Franklin, G. Skifstad, R. Rolock, I. Mehrotra, V. Ding, A. Hansen, D. Weintrop, and D Harlow. 2017. Using upper-elementary student performance to understand conceptual sequencing in a blocks-based curriculum. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17)*. New York: ACM, 231–236.

N. Fraser. 2015. Ten things we've learned from Blockly. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 49–50.

D. Garcia, B. Harvey, and T. Barnes. 2015. The beauty and joy of computing. *ACM Inroads* 6, 4 (November 2015), 71–79.

J. Goode, G. Chapman, and J. Margolis. 2012. Beyond curriculum: The exploring computer science program. *ACM Inroads* 3, 2 (2012), 47–53.

S. Grover and S. Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, In *Variables, and Boolean Logic*. ACM Press, 267–272.

S. Grover, R. Pea, and S. Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Comput. Sci. Educ.* 25, 2 (April 2015), 199–237.

S. Grover, R. Pea, and S. Cooper. 2016. Factors influencing computer science learning in middle school. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 552–557.

D. Harlow, H. Dwyer, A. Hansen, A. Iveland, and D. Franklin. Accepted. Ecological design based research in computer science education: Affordances and effectivities for elementary school students. *Cogn. Instr.* (Accepted).

B. Harvey. 1997. *Computer Science Logo Style: Beyond Programming*. MIT Press.
B. Harvey and J. Mönig. 2010. Bringing "no ceiling" to Scratch: Can one language serve kids and computer scientists? In J. Clayson & I. Kalas, eds. *Proceedings of Constructionism 2010 Conference*. 1–10.
C. Hill, H. Dwyer, T. Martinez, D. Harlow, and D. Franklin. 2015. Floors and flexibility: Designing a programming environment for 4th-6th grade classrooms. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 546–551.
M. Homer and J. Noble. 2014. Combining tiled and textual views of code. In *IEEE Working Conference on Software Visualisation (VISSOFT'14)*. IEEE, 1–10.
M. S. Horn and U. Wilensky. 2012. NetTango: A mash-up of netlogo and tern. In *When Systems Collide: Challenges and Opportunities in Learning Technology Mashups*.
K. Johnsgard and J. McDonald. 2008. Using Alice in overview courses to improve success rates in programming I. In *IEEE 21st Conference on Software Engineering Education and Training, 2008 (CSEET'08)*. 129–136.
J. Kaput, R. Noss, and C. Hoyles. 2002. Developing new notations for a learnable mathematics in the computational era. *Handb. Int. Res. Math. Educ.* (2002), 51–75.
M. Kölling, N. C. C. Brown, and A. Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE'15)*. New York: ACM, 29–38.
M. Kölling and F. McKay. 2016. Heuristic evaluation for novice programming systems. *Trans. Comput. Educ.* 16, 3 (June 2016), 12:1–12:30.
C. M. Lewis. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. 346–350.
D. J. Malan and H. H. Leitner. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bull.* 39, 1 (2007), 223–227.
J. H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bull.* 40, 1 (2008), 367–371.
J. H. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. 2010. The Scratch programming language and environment. *ACM Trans. Comput. Educ. TOCE* 10, 4 (2010), 16.
Y. Matsuzawa, T. Ohata, M. Sugiura, and S. Sakai. 2015. Language migration in non-CS introductory programming through mutual language translation environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM Press, 185–190.
O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE'11)*. ACM, 168–172.
O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. 2010. Learning computer science concepts with Scratch. In *Proceedings of the 6th International Workshop on Computing Education Research*. 69–76.
J. Mönig, Y. Ohshima, and J. Maloney. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 51–53.
B. Moskal, D. Lurie, and S. Cooper. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. 75–79.
P. Mullins, D. Whitfield, and M. Conlon. 2009. Using Alice 2.0 as a first language. *J. Comput. Sci. Coll.* 24, 3 (2009), 136–143.
S. Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.
T. W. Price and T. Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the 11th Annual International Conference on International Computing 944 Education Research (ICER'15)*. New York. ACM Press, 91–99.
A. Repenning. 1993. Agentsheets: a tool for building domain-oriented visual programming environments. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. ACM, 142–143.
M. Resnick, B. Silverman, Y. Kafai, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, and J. Silver. 2009. Scratch: Programming for all. *Commun. ACM* 52, 11 (November 2009), 60.
A. Robins, J. Rountree, and N. Rountree. 2003. Learning and teaching programming: A review and discussion. *Comput. Sci. Educ.* 13, 2 (2003), 137–172.
R. V. Roque. 2007. *OpenBlocks: An Extendable Framework for Graphical Block Programming Systems*. Master's Thesis. Massachusetts Institute of Technology.
R. Benjamin Shapiro and M. Ahrens. 2016. Beyond blocks: Syntax and semantics. *Commun. ACM* 59, 5 (April 2016), 39–41.
B. L. Sherin. 2001. A comparison of programming languages and algebraic notation as expressive languages for physics. *Int. J. Comput. Math. Learn.* 6, 1 (2001), 1–61.
W. Slany. 2014. Tinkering with pocket code, a scratch-like programming app for your smartphone. In *Proceedings of Constructionism 2014*.
N. Smith, C. Sutcliffe, and L. Sandvik. 2014. Code club: Bringing programming to UK primary schools through scratch. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE'14)*. New York: ACM, 517–522.

F. Swetz. 1989. *Capitalism and Arithmetic: The New Math of the 15th Century*, La Salle, IL: Open Court.
B. Tangney, E. Oldham, C. Conneely, S. Barrett, and J. Lawlor. 2010. Pedagogy and processes for a computer programming outreach workshop—The bridge to college model. *IEEE Trans. Educ.* 53, 1 (2010), 53–60.
M. Tempel. 2013. Blocks programming. *CSTA Voice* 9, 1 (2013), 3–4.
The Playful Invention Company. 2008. *PicoBlocks*. Playful Invention Company.
A. Wagh and U. Wilensky. 2012. Evolution in blocks: Building models of evolution using blocks. In C. Kynigos, J. Clayson, and N. Yiannoutsou, eds. *Proceedings of the Constructionism 2012 Conference*.
D. Weintrop. 2016. *Modality Matters: Understanding the Effects of Programming Language Representation in High School Computer Science Classrooms*. PhD Dissertation. Evanston, IL: Northwestern University.
D. Weintrop and U. Wilensky. 2016. Bringing blocks-based programming into high school computer science classrooms. *Paper presented at the Annual Meeting of the American Educational Research Association (AERA)*.
D. Weintrop and U. Wilensky. 2012. RoboBuilder: A program-to-play constructionist video game. In C. Kynigos, J. Clayson, and N. Yiannoutsou, eds. *Proceedings of the Constructionism 2012 Conference*.
D. Weintrop and U. Wilensky. 2015a. The challenges of studying blocks-based programming environments. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 5–7.
D. Weintrop and U. Wilensky. 2015b. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC'15)*. New York: ACM, 199–208.
D. Weintrop and U. Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the 11th Annual International Conference on International Computing Education Research (ICER'15)*. New York: ACM, 101–110.
U. Wilensky. 2001. Modeling nature's emergent patterns with multi-agent languages. In *Proceedings of EuroLogo*. 1–6.
U. Wilensky. 2006. Complex systems and restructuration of scientific disciplines: Implications for learning, analysis of social systems, and educational policy. *Paper presented at the Annual Meeting of the American Educational Research Association*.
U. Wilensky, C. E. Brady, and M. S. Horn. 2014. Fostering computational literacy in science classrooms. *Commun. ACM* 57, 8 (August 2014), 24–28.
U. Wilensky and S. Papert. 2010. Restructurations: Reformulating knowledge disciplines through new representational forms. In J. Clayson & I. Kallas, eds. *Proceedings of the Constructionism 2010 Conference*.
U. Wilensky and W. Rand. 2014. *Introduction to Agent-Based Modeling*, Cambridge, MA: MIT Press.
M. H. Wilkerson-Jerde and U. Wilensky. 2010. Restructuring change, interpreting changes: The deltatick modeling and analysis toolkit. In J. Clayson AND I. Kalas, eds. *Proceedings of the Constructionism 2010 Conference*.
A. Wilson and D. C. Moffat. 2010. Evaluating scratch to introduce younger schoolchildren to programming. In *Proc. 22nd Annu. Psychol. Program. Interest Group Universidad Carlos III de Madrid, Leganés, Spain* (2010).
D. Wolber, H. Abelson, E. Spertus, and L. Looney. 2014. *App Inventor 2: Create Your Own Android Apps*. 2nd ed. Beijing: O'Reilly Media.
D. Yaroslavski. 2014. *Lightbot*. Armor Games.

Received December 2016; revised April 2017; accepted May 2017