From Blocks to Text and Back: Programming Patterns in a Dual-modality Environment

David Weintrop
University of Chicago
UChicago STEM Education
Chicago, IL, USA 60637
dweintrop@uchicago.edu

Nathan Holbert
Teacher's College, Columbia University
Department of Mathematics, Science, and Technology
New York City, NY, USA 10027
holbert@tc.columbia.edu

ABSTRACT
Blocks-based, graphical programming environments are increasingly becoming the way that novices are being introduced to the practice of programming and the field of computer science more broadly. An open question surrounding the use of such tools is how well they prepare learners for using more conventional text-based programming languages. In an effort to address this transition, new programming environments are providing support for both blocks-based and text-based programming. In this paper, we present findings from a study investigating how learners use a dual-modality environment where they can choose to work in either a blocks-based or text-based interface, moving between them as they choose. Our analysis investigates what modality learners choose to work in, and if and why they move from one representation to the other within a single project. We conclude with a discussion of design implications and future directions for this work. This work contributes to our understanding of the affordances of blocks-based programming environments and advances our knowledge on how best to utilize them.

CCS Concepts
Human-centered computing→Visualization • Social and professional topics→Computer science education

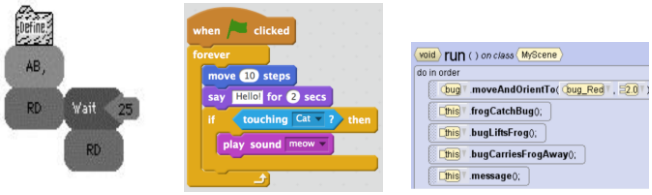
General Terms
Design, Human Factors, Languages

Keywords
Computer Science Education; Blocks-based Programming

1. INTRODUCTION
The ability to program is becoming an increasingly valuable skill in our highly technological world. In response to the growing interest in learning to program, many introductory programming tools are being designed to be 'low-threshold'—meaning they are intuitive, welcoming, and appeal to diverse audiences. One such approach that has become widely adopted in the design of introductory tools is blocks-based programming (Figure 1), which provides syntactic information through the visual shape of commands and allows users to author programs by dragging-and-dropping block-shaped commands together. As more, and younger, learners are introduced to programming, the blocks-based approach is becoming the de facto standard for introductory programming environments and for early exposure to computer science (CS) more broadly.

Despite widespread use, open questions remain about the blocks-based modality and its fit in conventional CS education. More specifically, it is unclear how well such tools prepare students for future CS learning opportunities or how best to transition learners from blocks-based introductory tools to more conventional text-based languages [19]. One proposed solution involves the creation of dual-modality interfaces that allow learners to seamlessly shift back-and-forth between blocks-based and textual representations [3, 7, 12, 16]. In addition to allowing the user to decide what modality to work in, such tools also provide an opportunity for learners to see each representation of code "side-by-side," which can highlight structural similarities as well as syntactic differences [22]. While recent work has offered insight into perceived supports offered by blocks-based environments, and in the ways learners transition from blocks to text, less is known about the particular conceptual resources mobilized by each representation. In other words, when novices have a choice between blocks and text, which modality do they choose? Why? And how does this process change as experience grows?

In this paper, we use Pencil Code [3], a programming environment that allows learners to switch between blocks-based and text-based representations of code, to investigate these questions. The paper begins with a review of blocks-based programming and its rise in formal educational contexts. We then present data on the pattern of modality choice in two distinct populations of novice programmers and provide an analysis exploring *why* and *when* learners move from one modality to the other. We conclude with a discussion of the implications of these findings with respect to the use of currently available blocks-based programming environments as well as the next generation of "low-threshold" tools. This paper contributes to our knowledge of how novices make use of blocks-based programming tools and advances our knowledge of the affordances of the modality.



LogoBlocks Scratch Alice

Figure 1. Three examples of blocks-based programming tools.

SIGCSE '17, March 8–11, 2017, Seattle, WA, USA.

ACM 978-1-4503-4698-6/17/03…$15.00.
DOI: http://dx.doi.org/10.1145/3017680.3017707

2. PRIOR WORK

Blocks-based programming environments are designed to ensure novice programmers have early successes. These environments use a programming-command-as-puzzle-piece metaphor, providing visual cues as to how and where commands can be used. To author programs in these environments the user drags-and-drops prefabricated commands onto a canvas, where they snap together if placed in a valid sequence. Lead by the popularity of tools like Scratch [21], Alice [6], and Blockly [10], blocks-based tools are becoming the standard approach for the design of programming tools for younger learners. Early blocks-based programming environments were inspired by the Logo language [18], and sought to provide an accessible way to allow learners to control the Lego/Logo programmable brick, a pre-cursor to the Lego Mindstorms kits [4]. Over time, a growing number of environments and libraries have been developed that incorporate visual, blocks-based programming techniques. A recent review of coding environments for children included 19 drag-and-drop tools among the 24 environments discussed for learners under the age of eight, and 28 drag-and-drop environments out of the 47 total reviewed environments [9].

The popularity of blocks-based programming tools has led to their incorporation into formal CS educational settings. Large scale curricular efforts, such as Exploring Computer Science [11] and all five courses currently listed on the AP CS Principles course's website [2] are using blocks-based programming environments as the primary mode of programming instruction. The use of blocks-based programming tools in formal educational context has seen mixed results, with some studies reporting successes [1, 7], while others questioning the suitability of such environments in preparing learners for future CS learning opportunities [17, 19]. Further, studies exploring learning in each modality suggest differences with respect to programming comprehension [14, 24], program generation [20] as well as with perceptions of the power and authenticity of each modality [23]. Understanding the affordances of blocks-based versus text-based environments remains an active area of research, with consensus on how best to utilize blocks-based tools in formal educational contexts yet to emerge.

One way forward in the blocks versus text debate is to create programming environments that support both modalities. For example, Pencil Code [3], allows learners to choose which modality they would like to use, including a button learners can click to convert their textual program to a blocks-based form or vice-versa (Figure 2). In this way, the choice of which modality to use is up to the learner. Matsuzawa et al. [16] created a tool that allowed learners to move between blocks-based and text-based versions of Java programs and proceeded to teach a semester long introductory programming course at the university level, tracking which modality students chose to use. They found that over the duration of the course, students were more likely to use blocks-based tools earlier in the year, and saw a steady shift of learners moving from the graphical, blocks-based interface to the text-based form of Java. In the work we present below, we replicate and expand on this work in two directions. First, our study includes two distinct populations, giving us insight into the universality of this trend. Second, we investigate novice programming patterns, specifically focusing on when and why learners shift modalities to better understand the supports provided by the different modalities. This second question can provide insights into the thought process associated with moving between modalities, further illuminating the affordances of each.

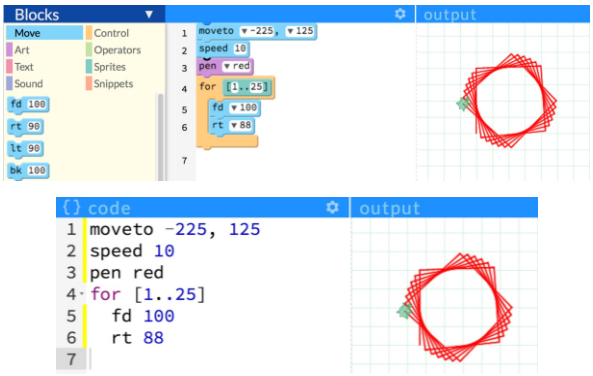


Figure 2. The two interfaces of Pencil Code, the text interface (top) and the blocks interface (bottom).

3. METHODS

In this paper, we explore when and why learners of varying experience switch between blocks-based and text-based modalities in a programming environment that supports both. Participants were drawn from two populations. One population, consisting of 13 girls, was recruited as part of a high school class designed to introduce students to computational thinking [5]. The girls spent three 100-minute classes working through a series of Pencil Code activities designed to introduce them to the basics of computer programming (including concepts like loops and variables). This condition, which we will refer to as the high school condition (HSC), consisted of eight high school freshmen, two 8th grade students, two sophomores, and a single junior. The school population is 72% African American, 25% Hispanic, and less that 2% each white and Asian, a distribution that is reflected in the class. Sixty-seven percent of students in the school are from low-income families. The three lessons culminated in students creating an interactive website to promote the class they were taking. As part of the assignment students authored programs to draw images on the screen, respond to user inputs, and programmatically incorporate images from the Internet.

The second population that participated in this study includes four girls and six boys enrolled in a graduate level course on the design of educational learning environments (mean age of 29) taught by one of the authors. Participants in the graduate condition (GC) completed a brief survey reporting prior programming experience as well as the Computer Attitude Survey [8] during the first session of the course. Students in the GC used Pencil Code as part of a "create a quilt" assignment. In this activity, students broke up into groups of four with each group member being assigned to create a program to visually represent themselves in a "patch" or section of the program's output area. To design his or her own patch, learners use programming commands to direct a "turtle" to move, draw, and color on a white screen. For example, to draw a blue square, learners might provide the commands:

```
pen blue, 10
for [1..4]
  fd 100
  rt 90
```

Individual programs from each group member were then "stitched" together to form a larger "quilt." Students in the GC began the assignment on the second session of class. Students completed the entire assignment outside of class sessions over one week

In both the HSC and GC conditions, all learners used the Pencil Code programming environment to complete the programming assignments, meaning programs could be constructed completely by dragging and snapping together blocks with a mouse, by typing text commands, or a mixture of both. To answer our research questions about why and when students choose to use a given modality or switch between modalities, all actions performed in the Pencil Code environment were logged. These logs provide a unique identifier for the user, a timestamp, the interface mode (blocks vs. text), the action performed, and the complete program. Using these logs, we can reconstruct the process of program construction, as well as identify when and where the user switched modalities during the course of this construction.

4. RESULTS

We begin our results section by looking at trends in student modality preference. We then correlate these trends with self-reported experience and confidences levels, before looking more closely at the log data to understand why students use one modality over the other, and what causes them to switch between modalities.

4.1 Student Modality Choice

Documenting what modality students choose to work in is the first step towards understanding the roles that the interface plays in novices learning to program. During the course of the studies, students overwhelming used the blocks-based modality for the programming assignments. Participants in the HSC used the block modality 92% of the time, while the GC participants used the block modality 91% of the time. This suggests, at least at a high level, that the blocks-based modality is not more developmentally appropriate for one age over the other. However, the distribution of time spent in the two modalities was not uniform across the student population. Instead, some students worked almost exclusively in blocks, while other preferred text, and a third group moved between the two. In other word, the choice of modality is not driven by age, but instead, by some other factor.

In looking at student modality choice over time, Matsuzawa et al. [16] designed a dual-modality Java programming environment and found that university CS students in an introductory programming class shifted modality preference over the course of the semester; starting with the blocks-based interface, and moving to text-based over time. They also found the students' choice of modality, and the amount of time spent using the blocks-based interface correlated with their self-reported confidence in programming. In an effort to explore the generalizability of this finding, we applied Matsuzawa et al.'s Block Editing Rate metric (*Rb*), which is the proportion of time spent in the blocks-based interface compared to total time spent on task, to actions logged by learners in the HSC and GC. Figure 3 shows student modality choice over time, with each row representing a single student. To calculate *Rb*, we used our log data, looking specifically at captured *Run* events. A *Run* event is logged every time a student runs their program, which is akin to complication in other languages. For each student, we took their full set of *Run* events, sorted them by time, then broke them down into 20 segments (corresponding to the 20 columns in Figure 3). For each segment, we then calculated the student *Rb* by counting the number of *Run* events called from the blocks-based mode and dividing it by the total number of runs in the segment. After calculating *Rb* for every segment and every student, we compiled them into a grid representation and used color intensity to represent time spent in each modality—the darker the square, the more time spent in the text interface. We then sorted the student rows by overall preference, with the students who spent the most time in blocks at the top, and student who preferred the text interface as the bottom.

A few things stand out about Figure 3. First, for both the HSC and the GC, there is an increasing trend toward text over time observable by the increased color intensity for both conditions as you move from the top left of the grid representation to the bottom right. At the outset of working in Pencil Code, students rarely used the text mode (as can be seen by the very light shading of the left-most column), whereas by the end of the observed activity, we see greater frequency of darker segments indicating heavy text-modality use. A second thing to note is the lack of a continuous transition for many students, this is especially pronounced in the HSC where the darkest patches came three-quarters of the way through the curriculum. This suggests that the transition from blocks-to-text is not a one directional shift, but instead students move back and forth between blocks and text

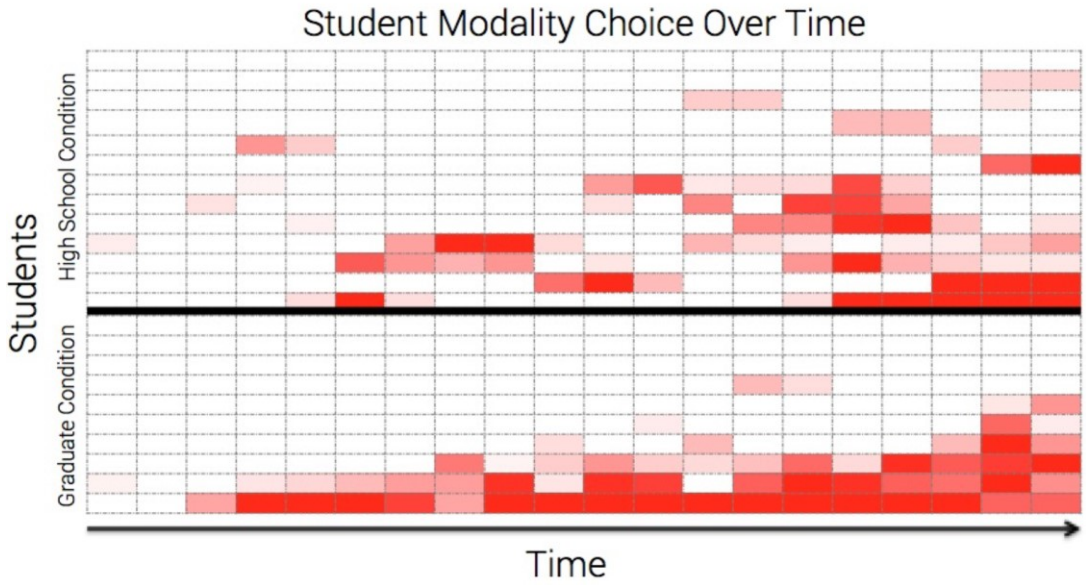


Figure 3. Student modality choice over time – the darker the square, the more time spent in the text interface.

over time. Finally, this representation highlights Pencil Code's ability to support students who want to *only* work in the blocks mode (as can be seen in the top three rows of the GC) as well as students who were either not interested in the blocks interface or felt they did not need the additional supports that blocks might provide (as can be seen in the bottom row of the GC condition). Together these findings show that Pencil Code, and the dual-modality design approach more generally, can meet the "low-threshold, high-ceiling" design goal desired in introductory programming learning environments.

To examine the likelihood of a correlation between self-reported confidence and interface choice, as reported by Matsuzawa et al. [16], students in the GC condition were asked to complete both an experience survey and the Computer Attitude Survey (CAS) [8]. On a scale of 1-5, students in the GC reported a mean experience of 3.5, meaning they had some level of computing experience, but do not self-identify as experts. On the CAS, a survey that measures the level to which survey takers have the same attitudes towards technology and programming as professional computer scientists, GC students had a mean score of 61%. CAS scores were highly correlated with self-reported experience (Spearman's coefficient 0.711, $p < 0.001$). Though on average, students of all levels of programming experience agreed with experts on the need for a flexible mindset towards programming (0.70) and on the real world value of programming (0.93), but scored low on the ability to see connections between various problem solving solutions (0.25).

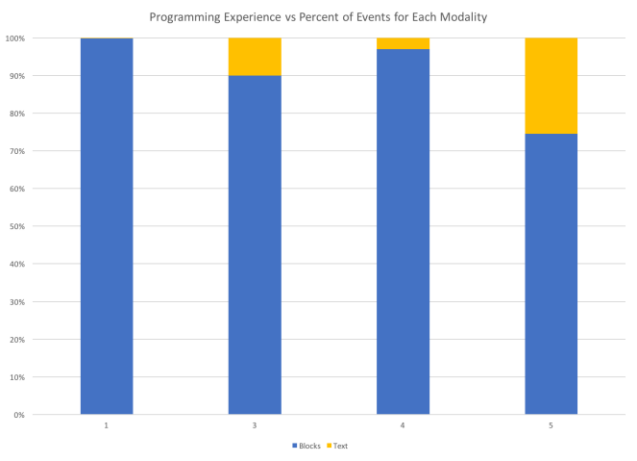


Figure 4. Percentage of programming events in the blocks or text modalities organized by self-reported programming experience.

Comparing students' self-reported experience with the log data on modality reveals a weak correlation between the two overall (Spearman coefficient 0.223 $p < 0.001$). Looking at students that reported the highest experience, we do find a greater propensity for the text modality (Figure 4). These two analyses replicate Matsuzawa's [16] previously reported findings on both student modality choice over time and the correlation between modality and confidence. Our work extends these previous findings by looking at university students outside of programming courses and younger, high school aged learners. We now turn to our second our research question, trying to understand when and why students shift between modalities.

4.2 Motivations for Shifting Modality

One finding from the literature on blocks-based programming is that students perceive blocks-based programming to be easier than writing programs in more conventional text-based languages [23]. Given our interest in learning and the design of new tools, understanding the reasons for this perception and if and how it bears out in practice are of particular importance. Dual-modality environments provide an opportunity to investigate and understand these aspects of blocks-based interfaces by looking at when shifts between modalities occur and what happens immediately after the shift, we can gain insight into the learner's motivation for switching modality. In other words, by investigating likely intentions when students who have become comfortable with the text modality move to blocks, we deepen our understanding of how the blocks-based modality supports novices.

To more easily determine what the learner is trying to accomplish when switching modalities, we focused on log events that were captured when students toggle the interface from text to blocks. For this analysis, the two student populations are grouped together in an attempt to understand the full breadth of motivations for moving between modalities, independent of assignment or experience. The logs contained 217 instances of this transition, each containing a snapshot of the program when the user was in the text modality. We then looked to the next log event in the system to see what the learner did after transitioning into the blocks modality, which also contains a program snapshot. By comparing the two program snapshots, we are able to determine the specific programmatic changes made after switching into the blocks modality. We then coded the specific changes using a coding scheme that identified whether the change was adding, moving, or removing code, and, in the case where code was added, which block type (based on the block "bins" predefined by Pencil Code, like `Motion` and `Control`) was selected.

After transitioning from text-to-blocks, there are a number of next steps learners could take, including adding a new command, deleting some portion of the program, moving blocks within the program, or simply returning to the text modality. While learners shifted to the block representation for a variety of reasons, our analysis indicates that 65.4% of these events were to add commands to their program. Two-thirds (67.1%) of these code block additions involved adding a block-type that had not been previously used in that program. The high frequency of the addition of previously unused block after a text-to-blocks transition indicates the block representation supported learners in adding new, never-before-used commands to their programs. On the one hand, this suggest learners may be using the "drawer" present in the blocks-based modality to browse the available set of commands. Alternatively, users may be relying on the block representation to avoid accidental syntax errors [15]. Finally, it is possible that some students may simply prefer dragging-and-dropping commands into their programs over the act of typing, which suggests that ease-of-composition motivated the transition.

Given that students frequently transition from text to blocks in order to add a new type of block to their program, we can gain insight into what commands are challenging or have difficult syntax by analyzing the types of new blocks that were added. Looking at the block type added after a text-to-blocks transition, the most frequently added blocks were from the *movement* (30.5%), *art* (21.6%), and *control* (18.0%) categories. As the assignments asked students to write code to move a turtle to create visually interesting patterns, it's perhaps not surprising that these blocks were frequently used. This is even after the user had begun

working in the text modality. When the addition of these blocks is analyzed for frequency of use, we found that 86.7% of the time a control block was added; it was added for the first time. First-add move blocks and art blocks were 62.8% and 63.9% respectively. The reliance on the block representation to add control blocks, which include commands like `repeat`, `if/else`, and `while`, further indicates the value of blocks in overcoming syntax challenges. However, as these commands also involve complex non-linear processes, the block representation may also be providing a conceptual support as learners attempt to incorporate these complex ideas into their programs.

Along with adding new blocks, other actions students took after toggling include deleting existing blocks (15.6% of the time), moving existing blocks to new locations in the program (13.3% of the time), or other events like toggling back to text or temporarily removing commands from the program (a combined 5.7% of post-toggle actions). While these actions were less frequent, and potentially less revealing than the patterns we found for adding new blocks, patterns within these actions do point to further affordances of the block representation. For example, when students transitioned from text-to-blocks and then proceeded to move code in their program, 60.7% of the time the move included shifting the scope of the moved code. In the blocks interface, shifting scope means moving the block into, or out of a block that has a nesting shape, such as a conditional or iterative block. Here the visual depiction of scope afforded by the blocks may provide a conceptual resource for learners as they attempt to leverage the non-linearity of code. A second interpretation could be that mechanics of moving blocks between different scopes is more easily accomplished in the blocks-based modality compared to copy-and-pasting or manually adding and removing whitespace to change scope. Both explanations point towards affordances of the blocks-based representation in helping novice programmers navigate issues of scope.

5. DISCUSSION

Our analysis of student use of the Pencil Code environment, which allows users to seamlessly switch between text and block representations of code, indicates blocks serve both an introductory role in a new programming environment as well as a conceptual support for those that have become accustomed to the tool. In this section, we review the central findings from our data and discuss potential implications of these findings.

Two different groups of programmers were evaluated using the Pencil Code environment to complete an open-ended drawing assignment. Despite one group being composed of high school students and the other graduate students, both groups overwhelmingly used the block representation when coding. As Figure 3 indicates, while blocks were used throughout the assignment, there was a gradual shift towards the text modality as the assignment progressed. As the environment (Pencil Code) and language (CoffeeScript) were new for all students regardless of condition, it is not surprising that they might use the blocks-based modality in the early stages of the assignment. These findings replicate those reported by Matsuzawa et al. [16] that found when students work in dual-modality programming environments, the frequency of using the text-based modality grows in parallel with experience in the environment. This paper extends these prior findings by broadening the population of learners this trend holds for to include non-CS majors as well as high school aged learners. However, it is interesting that there was not a quicker or larger shift towards text-based coding for students in the GC condition that reported higher levels of programming experience. This may suggest that students' self-reported experiences were not accurate, that the language experience these students had did not support them in using the text-based language in Pencil Code, or that the blocks representation was robust enough (and the environment user-friendly enough) that these experienced users never felt the *need* to code exclusively in text.

To understand how the blocks-based modality supported novice programmers as they authored their programs, we analyzed the contents and changes of program snapshots that occurred when users shifted from the text modality to blocks. This analysis revealed that when students that coded in the text modality returned to the block representation, they did so mostly to add new code. When student made this move, they were usually adding commands that they had not yet used in their program. The fact that 65.4% of the blocks added were being added for the first time suggests that the block modality supported users in finding new commands for use in their program. Furthermore, of the command types added, complex control blocks (such as for loops and if statements) were often added (86.7%) for the first time during this text-to-blocks shift. This reliance upon the blocks-based modality may indicate that users were either unable or hesitant to add commands that may introduce syntax errors into their programs, which can be particularly tricky for control commands. An alternative explanation is that the blocks-based modality may serve a conceptual function as learners attempt to incorporate non-linear commands into their programs.

While it is often claimed that blocks-based programming environments offer the advantage of reducing syntax errors [4, 15], our findings suggest that blocks also offer information about what is possible in the space and provide a low-stakes means of exploring unfamiliar code [23]. By organizing and displaying possible programming commands alongside the programs being authored, the user is exposed to possibilities they may not have known were available throughout their design process. This matches findings in human-computer interaction on the ease of recognition over recall. Because blocks allow these previously unknown commands to be easily added in the middle of an in-development program without fear of syntax errors or structural issues (and easily moved and/or removed thereafter), users may be more likely to experiment with these unfamiliar commands in their code. This particular affordance of the block representation suggests that even text-only environments might benefit from the presence of a "drawer" of possible/useful code that can easily be added to in-process programs.

Along with illuminating patterns in how novices learn to program in dual-modality programming environments, this work also has potential design implications for the creation of low-threshold/high-ceiling programming environments. On the low-threshold end of the spectrum, the finding that students often transition to the blocks-based modality to add a new command or to change the scope for existing commands, suggests that including similar features into text-environments could be useful for novices just becoming comfortable with text-based programming. One implementation of this can be seen in Greenfoot's "frame-based" editing approach to text environments [13]. As for ensuring a high-threshold for more advanced programmers, our findings suggest a dual-modality design means at the least, blocks don't restrict the text-based programmer, and at the best, blocks provide new opportunities for exploration and experimentation to enhance or extend text-based programs. By allowing learners to choose which modality they want to work in, novices who need additional support can leverage the various

scaffolds designed into blocks-based tools, while students with more experience or who are particularly eager to learn text-based coding can do so. Further, the dual modality approach gives learners control of their own learning experience, deciding for themselves about what scaffolds they want or when they might need more support.

6. CONCLUSION

As interest in learning to program continues to grow, new interfaces and programming languages are being designed to make the practice engaging and accessible. In this paper, we explore how novice programmers use Pencil Code, a tool that provides text and blocks-based representations of code, to understand how access to both modalities impacts programming practices. By studying how and when student move back and forth between blocks-based and text-based interfaces, we advance our understanding of the affordances of the tools for helping beginning programmers as they are starting out. Our findings indicate both high school and university-aged novice programmers productively used the dual-modality feature of Pencil Code throughout their programming experience. All students started their time with Pencil Code in the blocks-based modality, with some students quickly moving to text while other staying in the graphical interface. Regardless of the modality chosen, all students were able to fully participate in the course and complete the programming activities, showing the effectiveness of the dual modality approach for welcoming and supporting novices, while also keeping more experienced programmers engaged. While much of the discussion around the design of introductory programming has been focused on debating which is better for learners – blocks or text, this paper shows the answer may be: why not both?

7. REFERENCES

[1] Armoni, M. et al. 2015. From Scratch to "Real" Programming. *ACM Transactions on Computing Education (TOCE)*. 14, 4, 25:1-15.

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

[3] Bau, D. et al. 2015. Pencil Code: Block Code for a Text World. *Proceedings of the 14th International Conference on Interaction Design and Children* (New York, NY, US), 445–448.

[4] Begel, A. 1996. *LogoBlocks: A graphical programming language for interacting with the world.* Electrical Engineering and Computer Science Department. MIT.

[5] Brady, C. et al. 2017. All Roads Lead to Computing: Making, Participatory Simulations, and Social Computing as pathways to Computer Science. IEEE Transaction on Education. *IEEE Transactions on Education*. 60, 1, 1–8.

[6] Cooper, S. et al. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*. 15, 5, 107–116.

[7] Dann, W. et al. 2012. Mediated transfer: Alice 3 to Java. *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, 141–146.

[8] Dorn, B. and Elliott Tew, A. 2015. Empirical validation and application of the computing attitudes survey. *Computer Science Education*. 25, 1, 1–36.

[9] Duncan, C. et al. 2014. Should Your 8-year-old Learn Coding? *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (New York, NY, USA), 60–69.

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

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

[12] Homer, M. and Noble, J. 2014. Combining Tiled and Textual Views of Code. *IEEE Working Conference on Software Visualisation (VISSOFT)* (Victoria, BC), 1–10.

[13] Kölling, M. et al. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. *Proceedings of the Workshop in Primary and Secondary Computing Education* (New York, NY, USA), 29–38.

[14] Lewis, C.M. 2010. How programming environment shapes perception, learning and goals: Logo vs. Scratch. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (New York, NY), 346–350.

[15] Maloney, J.H. et al. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*. 10, 4, 16.

[16] Matsuzawa, Y. et al. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 185–190.

[17] Meerbaum-Salant, O. et al. 2011. Habits of programming in Scratch. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education* (Darmstadt, Germany), 168–172.

[18] Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic books.

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

[20] Price, T.W. and Barnes, T. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment., 91–99.

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

[22] Weintrop, D. et al. 2015. Teaching Text-based Programming in a Blocks-based World. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (New York, NY, USA), 678–678.

[23] Weintrop, D. and Wilensky, U. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. *Proceedings of the 14th International Conference on Interaction Design and Children* (New York, NY, USA), 199–208.

[24] Weintrop, D. and Wilensky, U. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA), 101–110.