The Challenges of Studying Blocks-based Programming Environments

David Weintrop
Learning Sciences
Northwestern University
Evanston, IL
dweintrop@u.northwestern.edu

Uri Wilensky
Learning Sciences & Computer Science
Northwestern University
Evanston, IL
uri@northwestern.edu

***Abstract*—In this paper we discuss some of the central challenges to rigorously studying blocks-based programming. We categorize these challenges into four groups: design, context, evaluation, and trajectory. For each category, we breakdown specific issues we confront as researchers in trying to isolate the blocks-based modality and outline our own strategies for addressing them. Throughout, we draw on our own experience as designers, educators, and researchers, along with past and current work, to provide guidance on ways to address these challenges and share insights our approach has yielded.**

Keywords—blocks-based programming; computer science education; design; methods

I. Introduction

Blocks-based programming is increasingly becoming the way that learners are being introduced to programming. Led by the popularity of tools like Scratch, Alice, and the suite of environments offered as part of Code.org's Hour of Code, blocks-based programming has become a central approach used in the design of introductory programming environments. Following this trend, a growing number of curricula are utilizing blocks-based programming tools, including the CS Principles project, the Exploring Computer Science program, and the materials being developed and disseminated by Code.org. Given the growing prominence of this approach, it is critical that we fully understand the consequences of the decision to rely on this modality in formal educational settings. Despite its growing prevalence, many open questions remain surrounding the effects of blocks-based programming on learning. This is due, in part, to the challenges of studying blocks-based programming independent of other factors that often accompany the activity of learning to program. In this paper, we discuss various challenges we face in trying to study blocks-based programming, grouping these challenges into four categories: Design, Context, Evaluation, and Trajectory. Drawing on our own experience conducting classroom-based research on blocks-based programming, we discuss specific challenges to studying programming modality and then present our own strategies to overcoming some of these challenges.

II. Challenges of Studying Blocks-based Programming

We break the challenges associated with studying blocks-based programming down into four general categories. The first category, Design, focuses on features of blocks-based programming tools and discusses challenges inherent to the representation, and various ways the blocks-based modality can be conflated with other features of introductory environments. The second category is Evaluation, which looks at challenges associated with evaluating learning in blocks-based environments. The next category, Context, outlines challenges associated with different factors external to the programming environment that affect learning and engagement. Finally, we explore challenges faced in studying learner trajectories that blocks-based programming tools support, specifically focusing on the question of studying transfer from blocks-based to more conventional text-based programming languages.

A. Design

The first set of challenges we discuss are those related to the goal of trying to study the blocks-based modality and features of its visual display and interaction dynamics. We use the term modality here to specifically refer to the visual, representational characteristics of blocks-based programming, which is distinct from the language (i.e. set of primitives provided), and the larger environment in which the modality is situated (i.e. the other elements of the interface like that palette that holds sets of blocks and the stage, where the results of programs are displayed). In highlighting the distinction between modality and language, we encounter the first challenge: the potential of conflating the two. While it is not possible to completely disentangle these two characteristics of blocks-based programming, it is important to recognize where language and modality are bound together, and where they can be separated, as well as the consequences of doing so.

An example of the challenge in conflating language and modality can be seen in Lewis' [1] comparison of Scratch and Logo. As part of their post test, students were asked to answer a question on conditional expressions, Figure 1 shows the two ways an if statement was presented in the question.

```
if greater? Mousex 0
  [setpencolor 2]
```

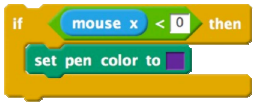

Fig. 1. A portion of a conditional logic question from [1].

So, when Lewis found that students did better on the Scratch version of the question, how do we explain the result? Is it because students find the infix `<` easier to interpret than the prefix `greater?` operator? Or possibly because the colors, shapes, and nesting of the blocks make that form easier to parse? These two potential explanations highlight the conflation of language and modality, in one explanation the difference is a feature of the language (`<` vs. `greater?`) while the other attributes the difference to the modality (text vs. blocks display). In our own work, we have encountered this language/modality conflation in a number of places, the most clear-cut example being in our analysis of student understanding of definite loops. In questions that had students compare blocks-based `repeat` commands with text-based `for` loops, students did significantly better on the blocks-based questions. However, when we compared blocks-based `for` loops to text-based `for` loops, we saw that performance gap disappear [2]. This suggests that the gains we found in the `repeat` vs. `for` questions were more likely due to language than modality, a finding that is consistent with Lewis' [1].

One way to address this issue is to use environments in which the blocks-based and text-based modalities share language features. One nice example of this is Pencil Code [3], where the blocks-based interface is essentially a visual overlay rendered on top of the text-based form of a program. Thus, the actual text of the program is identical across the two modalities. Figure 2 shows Pencil Code text and blocks versions of the question from Figure 1 using JavaScript as the base language.

```
if (mouse.x > 0) {
  pen(red);
}
```

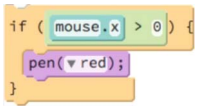

Fig. 2. Pencil Code's text and block-based rendering of the code in Figure 1.

Part of the reason it is not possible to completely disentangle modality from language is due to the blocks-based visualization's ability to clearly depict the constituent elements of a single command. Having a clearly delimited boundary to a command makes it visually apparent how to parse a program and thus makes it possible for a single command to be made up of compound phrases, like: `say Hello! for 2 seconds`. This enables a programming language to have a more readable, natural language feel; a feature that learners identify as contributing to the easy-of-use of blocks-based tools [4].

A second conflation the emerges in studying blocks-based programming tools is linking the blocks-based modality with the other features of the programming environment that surround and give meaning to the programming activity. For example, Scratch includes not just the blocks-based programming interface, but also the Logo-like ability to introduce sprites and have them visually enact programs on the screen. By embedding programming in an environment that makes it easy to create games and interactive stories that can easily be personalized and shared, Scratch has been very successful at getting learners engaged and excited about programming [5]. As we are interested in understanding the affordances of blocks-based programming, it is important to try and understand the contribution of the blocks-based modality to creating this sucessful and engaging programming environment. Our solution to this issue has been to build on the successes of rich, engaging programming contexts by bringing text-based programming into these interactive, graphical programming contexts. In Snappier!, a modification of Snap! [6], we made it possible for learners to create custom blocks that are controlled by short text-based programs. In this way, text-based programs could interact with all the elements of the blocks-based world and be incorporated into larger blocks-based programs [7].

B. Evaluation

A second type of challenge we face when trying to understand the strengths and drawbacks of blocks-based programming stems from a dearth of good, validated assessments that will give us comparative insights between blocks-based and text-based modalities. While there are a number of assessments that use blocks-based programming (e.g., [8], [9]), and others built around programming in blocks-based environments (e.g., [10]), there are few assessments well suited to the research pursuit of understanding the affordances and drawbacks of blocks-based programming relative to isomorphic text-based alternatives. To fill this gap, we created the Commutative Assessment [2]. Each multiple-choice question on the assessment includes a short program that can be displayed in either a blocks-based or text-based form. The set of potential answers for each question includes the correct answer along with choices informed by prior research on novice programming misconceptions. We used the Commutative Assessment in a 10 week study that had students take the assessment three times, alternating the modality shown for each question across the different administrations, thus giving us with-in student performance, as well as time-series data, to try and link conceptual understanding with modality. For a longer description of the assessment, as well as findings from its use, see [2].

The previously discussed assessments focus more on program comprehension than program generation. To date, most of the work looking at programming authoring has relied on qualitative analysis of program authoring, which is very useful for understanding how learners rely on features of the modality (e.g. [11], [12]). Another methodology that is growing in popularity that can be useful for assessing blocks-based programming tools is the use of educational data mining and learning analytics techniques. The collection and analysis of log data from students as they develop, run, and revise their programs can provide insights into the practice of program composition. While there is some innovative work happening in this space (e.g., [13], [14]), we have yet to see this approach applied as a methodology for evaluating programming modality. This is an analysis we intend on doing in the future and think it could be very insightful for understanding the effect that modality has on practices that can complement the findings from the quantitative content evaluations.

C. Context

There are many differences between formal classroom settings and the informal contexts in which much of the blocks-based learning research has occurred. The differences

include the layout of the physical space, the peer culture, the autonomy learner have, the time dedicated to each activity, and the presence (or absence) of experts in the form of teachers, councilors, or other experienced programmers. Given all these differences, it is not safe to assume that what is true in an informal, open-ended context would necessarily hold in formal classrooms. Trying to comparatively study the role of modality in supporting (or hindering) a student's emerging programming understanding needs to account for these larger contextual differences. This challenge is best addressed through study design. In a recent study, we used a quasi-experimental design to hold constant many of these external factors. We followed three introductory classrooms in a single school; all three classes worked through the same curriculum and were taught by the same teacher in the same classroom. Each class used a different version of the same programming tool, with the only difference between them being the programming modality, either blocks-based, text-based, or a hybrid blocks/text tool. This design allowed us to focus specifically on questions of representation and modality that is at the heart of our research agenda.

D. Trajectory

A major open question facing the computer science education community is what comes after blocks-based programming. As more curricula incorporate blocks-based tools, especially at the high school and undergraduate levels, educators face the question of how best to transition learners from the blocks-based modality to more conventional text-based programming languages. Part of the challenge of answering this question convincingly is its longitudinal nature, as this transition often happens between courses. In such cases, claims can be made about successful transitions, but it is hard to attribute the successes to specific features of the blocks-based modality due to the amount of time that has passed, leaving the researchers to make only general claims (e.g., [15]). Additionally, such studies lack a control condition, making it difficult to attribute improvements to the modality (or features of the modality), and instead have to take the environment as a whole, making such studies susceptible to the issues of conflation previously discussed. We are studying the blocks-to-text transition by having shift modalities during a single course, holding constant classroom, peers, and teachers. We follow students for the first five weeks of an introductory programming class while they use a blocks-based environment, then continue to observe, interview, and collect program log-data on those students for an additional ten weeks as they transition to Java. In following students across the transition and having students initially work in differing modalities, we have designed a study that will be able to give us data to more closely attribute successes and failures of the transition to features of the introductory programming tools.

III. Conclusion

Programming is now recognized as a core 21st century skill. School districts, responding to this trend, are rapidly introducing new computer science courses and integrating computational thinking into traditionally non-computational coursework to accommodate this shifting computational landscape. Given this trend, it is critical that we as computer science education researchers and designers understand the tools and curricula that we are advocating for schools to use, as the environments and courses that are adopted today will lay the foundation upon which computer science, and computational thinking more broadly, will be taught for years to come. In this paper we have laid out what we see as the central challenges to studying the blocks-based programming approach. Our hope with this paper is to raise awareness of the challenges we face and demonstrate possible ways to address these research challenges and pave the way for rigorous, careful research that will yield insights into the tools students are using today and guide the way for the tools that students will use tomorrow.

References

[1] C. M. Lewis, "How programming environment shapes perception, learning and goals: Logo vs. Scratch," in *Proc. of the 41st ACM Technical Symposium on CS Ed*, New York, NY, 2010, pp. 346–350.

[2] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," in *Proc. of the 12th Annual Int. Conf. on International Computing Education Research*, Omaha, NE, 2015.

[3] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil Code: Block Code for a Text World," in *Proc. of the 14th Int. Conf. on Interaction Design and Children*, New York, NY, USA, 2015, pp. 445–448.

[4] D. Weintrop and U. Wilensky, "To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming," in *Proc. of the 14th Int. Conf. on Interaction Design and Children*, New York, NY, USA, 2015, pp. 199–208.

[5] M. Resnick et al., "Scratch: Programming for all," *Commun. ACM*, vol. 52, no. 11, p. 60, Nov. 2009.

[6] B. Harvey and J. Mönig, "Bringing 'no ceiling' to Scratch: Can one language serve kids and computer scientists?," in *Proc. of Constructionism 2010*, Paris, France, 2010, pp. 1–10.

[7] D. Weintrop, U. Wilensky, J. Roscoe, and D. Law, "Teaching Text-based Programming in a Blocks-based World," in *Proc. of the 46th ACM Technical Symposium on CS Ed*, New York, NY, USA, 2015, p. 678.

[8] C. M. Lewis, "Is pair programming more effective than other forms of collaboration for young students?," *Comput. Sci. Educ.*, vol. 21, no. 2, pp. 105–134, Jun. 2011.

[9] S. Grover, S. Cooper, and R. Pea, "Assessing computational learning in K-12," 2014, pp. 57–62.

[10] L. Werner, J. Denner, S. Campe, and D. C. Kawamoto, "The fairy performance assessment: measuring computational thinking in middle school," in *Proc. of the 43rd ACM technical symposium on CS Ed*, 2012, pp. 215–220.

[11] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari, "Habits of programming in Scratch," in *Proc. of the 16th Annual Joint Conference on ITiCSE,* Darmstadt, Germany, 2011, pp. 168–172.

[12] D. Weintrop and U. Wilensky, "Supporting computational expression: How novices use programming primitives in achieving a computational goal," presented at AERA 2013, San Francisco, CA, USA, 2013.

[13] M. Berland, T. Martin, T. Benton, C. Petrick Smith, and D. Davis, "Using Learning Analytics to Understand the Learning Pathways of Novice Programmers," *J. Learn. Sci.*, vol. 22, no. 4, pp. 564–599, 2013.

[14] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, "Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming," *J. Learn. Sci.*, vol. 23, no. 4, pp. 561–599, 2014.

[15] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari, "From Scratch to 'Real' Programming," *ACM TOCE*, vol. 14, no. 4, pp. 25:1–15, 2015.