Toward a Responsive Interface to Support Novices in Block-Based Programming

Fernando J. Rodríguez*, Cody R. Smith†, Andy Smith†,
Kristy Elizabeth Boyer*, Eric N. Wiebe†, Bradford W. Mott†, and James C. Lester†
*University of Florida, Gainesville, Florida, USA
†North Carolina State University, Raleigh, North Carolina, USA
Email: fjrodriguez@ufl.edu

***Abstract*—Block-based programming environments are often used for teaching novice students, including at the undergraduate level. Despite the simplification these tools offer, undergraduates still require additional support, often learning programming by themselves or in large section courses with little instructional support. Programming environments that provide adaptive support hold great promise for meeting this need. This paper presents the early design and piloting of PRIME, a learning environment under development that provides scaffolded support for novices in block-based programming. A pilot study with PRIME compared two implementations of the functionality for moving between programming subtasks: one with a static "Next Step" button for advancing through subtasks at any time, and one with a responsive button that only appeared once the current subtask was completed. Analysis of students' code quality showed that students in the responsive condition achieved higher quality code in later programming tasks. The results highlight design considerations and opportunities for adaptively supporting novices in block-based programming.**

I. INTRODUCTION

A central thrust of computer science education research has been improving the recruitment and retention of students into computing-related fields of study [1]. However, many undergraduate students face challenges in traditional programming courses [2], as these have been shown to be particularly difficult for *novice* learners [3]. Additionally, there is an ever-growing population of "conversational" programmers, students that wish to learn essential programming concepts but who are not interested in becoming professional programmers [4]. Block-based programming languages are a promising choice for supporting these novices because they reduce the need to focus on syntax, which may encourage novices to attempt more complex implementations [5]. When compared to text-based programming languages, students using block-based languages have been shown to spend less time idle, or without modifying their code [6]. Additionally, some aspects of block-based code representations, such as the nesting of blocks and the closer match with natural language descriptions, can help students better understand programming concepts [7].

For all of their benefits in teaching programming, block-based programming languages do not remove all of the challenges with learning programming. While block-based programming tools remove hurdles related to low-level syntactic requirements of most text-based programming languages (e.g., semicolons at the end of statements), they still require correct application of core programming concepts such as loops and conditionals. They also require properly conceived and constructed algorithmic abstractions of the problem. Undergraduate students who are attempting to implement new concepts learned in class may still struggle with programs that do not produce the expected results even if there are no compiling issues. This may be particularly true of non-CS majors who have minimal prior experience with programming and CS concepts.

Efforts to expand the capabilities of block-based languages with intelligent support are becoming more common: Snap! introduced first-class data types and procedures to Scratch [8], and additional platforms have also been created to help developers easily extend the functionality of Scratch [9]. Intelligent support is another means of providing help for struggling novice programmers. Intelligent tutoring systems use data from learners to adapt problem difficulty [10], provide hints [11], and give feedback [12], all of which can serve to improve their learning experiences.

The study presented in this paper was conducted within the context of the PRIME programming environment. The main objective of the PRIME project is to develop a learning environment that provides adaptive support to learners as they interact with programming tasks in a block-based environment. The target audience for PRIME is undergraduate non-computer science majors taking an introductory programming course, or independent learners who wish to develop their coding skills outside of a classroom context. As an initial exploration of adaptivity in PRIME, we conducted an experiment to evaluate a form of responsive feedback, the appearance of a "Next Step" button when a subtask within a larger programming task was completed. A group of undergraduate programming novices completed a set of tasks in one of two versions of PRIME: one with a responsive "Next Step" button that appeared only upon completion of a subtask, and one in which the button was always present, providing the student greater freedom over when to move on. Students using the responsive version of PRIME achieved higher code quality than the students with the static version of PRIME in later tasks, based on scoring their solutions according to a correctness rubric. The results of this work can inform the ongoing and future design of systems that support novice students in learning block-based programming.

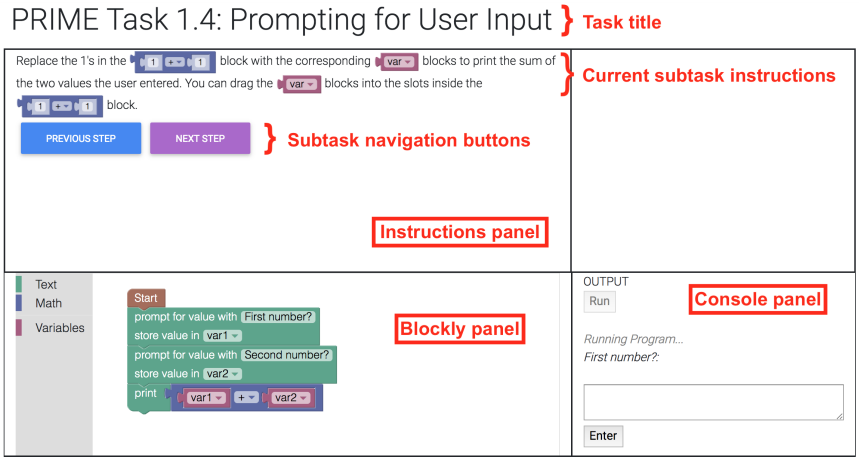

Figure 1. Screenshot of the PRIME programming interface.

II. PRIME OVERVIEW

The PRIME (Personalized Real-time Intelligent Mentoring Environment) project is a multi-year effort to design, implement, and evaluate an intelligent tutoring system to support programming novices in learning computer science concepts through a block-based language. We are designing the PRIME environment to serve both as a tool that faculty can use during the first few weeks of an introductory programming course, and as a standalone environment that learners can use to learn programming concepts independent of any course.

Task Progression Design

The design process for PRIME began by developing the first draft of the set of tasks that PRIME would use as its basis for teaching programming concepts. To do so, we reviewed the syllabi for introductory programming courses from the fifty top-rated undergraduate computer science programs in the US [13]. From this review, we extracted the set of topics that are typically covered in the first five weeks of the courses and the order in which they are covered. This work resulted in a topic sequence split into the following five units: 1) Input/Output, Variables, and Loops; 2) Functions, Parameters, and Return Values; 3) Conditional Execution; 4) String Manipulation and Basic Data Structures; and 5) Search and Sort Algorithms. The work presented in this paper focuses on Unit 1, which covers the topics of displaying output, variables, mathematical expressions, requesting user input, and definite loops.

Interface Design

When selecting a programming language for PRIME, we evaluated different alternatives for block-based programming platforms. We selected Google's Blockly framework [14] due to its ease of customization and ability to translate block programs into text-based (e.g., Python) equivalents. The transition between blocks and text is an important component of the project, but it is beyond the scope of this paper.

Figure 1 shows a screenshot of the PRIME prototype in HTML5. The main user interface includes the *Blockly* panel, the *Console* panel, and the *Instructions* panel. The Blockly panel consists of a visual coding widget with the block-based coding workspace and toolbox. The default workspace has been augmented with a "Start" block, which serves a similar purpose to the "main" function in other programming languages. The toolbox varies for each task, gradually adding more blocks as the students complete tasks and are introduced to new topics. This approach is based on prior research which suggests it can reduce extraneous cognitive load [15] and increase interface usability for learners [16].

The Console panel contains a "Run" button and shows the output generated from running the program. An input window also appears in this panel if a program prompts the user for input. Finally, the Instructions panel contains step-by-step instructions for a given task. This type of instruction format is common for intelligent programming tutors [17]. In addition to navigation buttons, this panel also contains positive feedback and links to the next task when a student has successfully completed the current task.

Preliminary Study and Focus Group

In order to better understand the needs of our target population of undergraduate programming novices, we conducted a preliminary focus group during the Fall 2017 semester. We recruited 22 undergraduate students (16 from computing-related majors) from our partner institution, a Historically Black College. These students were given one hour to write a program for a number guessing game within a block-based interface built using Google's Blockly framework.

After the programming activity, we split the students into groups of 2 or 3 for focus group sessions. These sessions were conducted by graduate student volunteers and consisted of questions on their perceptions of computer science and what kinds of things they would like to learn within a computer science course. Students from computing majors commonly associated computer science with "coding," and most students described it as "difficult" and "frustrating." Students from non-computing majors were asked what they valued most from

their programming course, with half of the students mentioning "a new way of thinking" and "web programming" as the most valuable takeaways. When asked, "If given the opportunity to spend some time learning computer science, what would you want to learn?," a common theme among student responses was the desire to learn something "I can put on a resume," such as web programming, data processing, encryption, and game development. The balance between facilitating computer science learning and providing an authentic programming experience is central to the design of PRIME, and adaptive features could help achieve that balance.

III. Data Collection Study and Analysis

When designing the subtask navigation options for the PRIME environment, we considered two functional designs for the "Next Step" button. One design had this button always visible, allowing users to page through all subtasks in a given task at any point. The other design only displayed the "Next Step" button when the current subtask was completed, serving as a form of positive feedback on the user's progress toward the task's final solution. The goal of the pilot study reported here was to compare how students performed in these functional designs of PRIME: the version with the button always visible (i.e., *Static*) and the version with the button that appears when the subtask is completed (i.e., *Responsive*).

Participants for the pilot study were recruited from an introductory programming course at the same college from the preliminary pilot study, but approximately halfway through the Spring 2018 semester. The study took place during one of their lab meetings. A total of 28 students were enrolled in the course. Of these, two did not consent to participate in the study, one left the lab meeting early, and one "clicked through" the survey responses. The data analyzed here therefore includes 24 study participants (8 female, 16 male; 20 Black, 3 white, 1 did not report race; ages 18-24, one age 27, and one age 28). Students were randomly assigned to each condition, with 13 in the Responsive condition (5 female, 8 male; 10 Black, 2 white, 1 did not report race; ages 18 - 24) and 11 in the Static condition (3 female, 8 male; 10 Black, 1 white; ages 18 - 24, one age 27, and one age 28).

Participants were given one hour to interact with PRIME and complete as many of the following seven tasks as possible:

1) Displaying an output message
2) Storing a numeric value in a variable
3) Storing a mathematical expression in a variable
4) Prompting for user input
5) Adding five user-entered numbers
6) Using a definite loop
7) Counting down from a number entered by the user

With the exception of tasks 6 and 7, all task instructions were divided into subtasks to help students get familiar with the interface and new concepts. Each subtask builds upon the subtasks before it, so they are typically completed in a predetermined order. The students could page through the task instructions in different ways: students in the Static condition would always see the "Next Step" button regardless of their progress, while students in the Responsive condition would only see the "Next Step" button when a given subtask within the main task had been completed. PRIME determines when a subtask is completed by comparing the current state of the program with a predefined set of conditions based on the presence of specific blocks. Upon successfully completing the given task, students in all conditions would be notified and given the link to the next task.

TABLE I
TASK 4 CODE QUALITY SCORING RUBRIC

Item	Points
Variables are defined	1
Program includes "prompt" blocks	1
Program includes "+" block	1
Program includes "print" block	1
Program includes variable blocks	1
Program prompts for two values	1
"prompt" blocks store values in separate variables	1
Program adds the variables together with the "+" block	1
Program prints the result of the addition operation	1
Program runs with no errors	1
Total	**10**

PRIME generates task logs when students run a program or advance to the next task. The logs include interface events and the current student program. We extracted these programs and manually scored them using itemized rubrics for each task. These rubrics were handmade, based on the presence of specific blocks and how they connected to each other. Table I shows the rubric for Task 4 as an example, which has a maximum score of 10.

IV. Results

We hypothesized that students in the Responsive (R) condition would achieve higher code quality scores than students in the Static (S) condition, since the Responsive condition provides a form of implicit feedback on subtask correctness. In the following section, we compare the code quality scores between the two conditions to determine if the average score of the students in the Responsive condition is indeed higher than the average score of the Static condition students.

Of the seven tasks, all 24 students completed tasks 1 through 3 ($n_R = 13$, $n_S = 11$). For task 4, 23 students completed ($n_R = 13$, $n_S = 10$), while 13 students completed task 5 ($n_R = 8$, $n_S = 5$). Continuing on, only 8 students ($n_R = 4$, $n_S = 4$) completed task 6, and 4 students completed task 7 ($n_R = 1$, $n_S = 3$). Due to the low number of students that completed tasks 6 and 7, we only consider tasks 1 through 5. There was no significant difference in the number of tasks completed between the Responsive and Static conditions.

Table II shows the average code quality scores for tasks 1 through 5. There were no significant differences in the average code quality scores between the Responsive and Static conditions for tasks 1 through 3. However, the average code

TABLE II
AVERAGE CODE QUALITY SCORES (AND STANDARD DEVIATION) BETWEEN CONDITIONS FOR EACH TASK (* INDICATES A SIGNIFICANT DIFFERENCE WITH P < 0.05)

Task	Responsive	Static	Max
1) Displaying output	3.92 (0.28)	3.72 (0.65)	4
2) Defining variables	7.62 (0.65)	7.09 (1.22)	8
3) Math expressions	7.23 (1.48)	7.27 (1.35)	8
4) Reading input*	**8 (3.63)**	**7 (2.49)**	**10**
5) Accumulator*	**6.38 (2.72)**	**3.8 (0.45)**	**10**

quality scores for the Responsive condition were significantly higher than the average code quality scores for the Static condition in both tasks 4 and 5 (one-tailed Wilcoxon signed-rank test: $p_4 = 0.0112$, $p_5 = 0.0448$).

V. DISCUSSION

Our goal in this study was to compare how students performed in programming tasks using one of two functional designs of PRIME's "Next Step" subtask navigation button: the Static version and the Responsive version. There were no significant differences in code quality scores for tasks 1 through 3, most likely due to these tasks having a more tutorial nature, explaining how the interface works and more explicitly stating what needed to be added to the code. Code quality scores for tasks 4 and 5 were higher for students in the Responsive condition than students in the Static condition. Task 4 introduces the concept of reading user input and asks students to add two numbers entered by the user. Task 5 involves implementing an accumulator program to add five values together, serving as a precursor for the need of a looping construct. We hypothesized that students in the Responsive condition benefited from the implicit feedback provided by the button, supporting their task completion.

To gain insight into this finding, we conducted focus groups at the end of the students' interactions with PRIME to understand how students used and interpreted the "Next Step" button within the different conditions. Students were divided into five focus groups: 3 for the Static condition (two groups of 4 students, one group of 3 students) and 2 for the Responsive condition (one group of 6 students, one group of 7 students).

Regarding the "Next Step" button, we asked students, "What helped you decide you were ready to move on to the next subtask?" Students in the Responsive condition reported using the appearance of the button as an indicator of when they were ready to move on. One student stated, "You had to get the instruction correct for it to even pop up," illustrating how students were aware of the purpose of the responsive button and using it to their advantage. Students in the Static condition mentioned adding blocks from the instructions and proceeding with the next subtask, going back to previous subtasks if they got stuck. This kind of behavior indicates that these students were evaluating their own progress, as evidenced by one student, "Once you get to the end of the instruction, you press Next Step and go to the next step and do all of that." Here, "end of the instruction" refers to the student's own perception of the subtask's completion. Students in the Static condition focus groups also stated task 5 was particularly challenging.

These results inform refinements to future iterations of the PRIME environment. We have observed from these students that even implicit feedback provided by the responsive "Next Step" button can maximize the benefit they receive from their interactions with PRIME. Future versions of PRIME will provide feedback not only on whether students' code is correct, but also on errors within the subtasks. One of the most requested forms of feedback by these students was real-time feedback on the code's status (e.g., "Am I doing this right?"). Regarding the Static condition students, self-reflection on their code is a skill we definitely want them to develop. However, our findings suggest that the scaffolding provided by the responsive button may still be valuable, especially for Unit 1's tasks. We have also continued to revise the instruction text with the goal of balancing between providing enough information for the students to complete the task and providing enough of a challenge for them to develop their coding skills.

Limitations

The introductory nature of the tasks, combined with the subtask scaffolding, do not allow for a wide variety of possible solutions. Additionally, the common language effect sizes for both significant results was 0.2, which indicates the difference in code quality scores between conditions was trivial. Verifying if these results generalize to more open-ended programming tasks is an essential component of PRIME's design for the later units. Additionally, students interacted with an early prototype of PRIME, and several students experienced network lagging and browser crashes during the study due to the high volume of data logging occurring in the background. It is possible that these bugs affected students' performance within the environment.

VI. CONCLUSION

Block-based programming languages offer significant promise for helping novices learn programming concepts. However, some students struggle even with block-based interfaces. The objective of PRIME is to improve the learning experience of these students by providing adaptive support as students engage with block-based programming. In an evaluation of a responsive feature within PRIME, we found that students with a responsive support button achieved higher code quality than students with a static button. Focus group discussions confirmed that students in the Responsive condition used the appearance of the button as a form of positive feedback on their progress, while students in the Static condition had to determine for themselves if their solutions were correct or not. The results presented here suggest that even modest positive feedback can have an impact on a student's performance in a learning task. Investigating more sophisticated forms of feedback to support these learners holds great potential for providing the most effective experience possible.

ACKNOWLEDGMENTS

We would like to thank the LearnDialogue Group and the Intellimedia Group for their helpful input. This research was supported by the National Science Foundation under Grants DUE-1626235 and DUE-1625908. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] T. Beaubouef and J. Mason, "Why the high attrition rate for computer science students: some thoughts and observations," *ACM SIGCSE Bulletin*, vol. 37, no. 2, pp. 103–106, 2005.

[2] C. Watson and F. W. Li, "Failure Rates in Introductory Programming Revisited," in *Proceedings of the 19th Conference on Innovation & Technology in Computer Science Education (ITiSCE '14)*, 2014, pp. 39–44.

[3] M. T. H. Chi, "Commonsense Conceptions of Emergent Processes: Why Some Misconceptions Are Robust," *The Journal of the Learning Sciences*, vol. 14, no. 2, pp. 161–199, 2005.

[4] P. K. Chilana, C. Alcock, S. Dembla, A. Ho, A. Hurst, B. Armstrong, and P. J. Guo, "Perceptions of Non-CS Majors in Intro Programming: The Rise of the Conversational Programmer," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 251–259.

[5] B. Xie and H. Abelson, "Skill Progression in MIT App Inventor," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2016, pp. 213–217.

[6] T. W. Price and T. Barnes, "Comparing Textual and Block Interfaces in a Novice Programming Environment," in *Proceedings of the 11th International Conference on International Computing Education Research (ICER '15)*, 2015, pp. 91–99.

[7] D. Weintrop and U. Wilensky, "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)*, 2015, pp. 101–110.

[8] B. Harvey and J. Mönig, "Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists?" in *Constructionism*, 2010, pp. 1–10.

[9] S. Dasgupta, S. M. Clements, A. Y. Idlbi, C. Willis-Ford, and M. Resnick, "Extending Scratch: New Pathways into Programming," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 165–169.

[10] N. Guin and M. Lefevre, "From a Customizable ITS to an Adaptive ITS," in *International Conference on Artificial Intelligence in Education*, 2013, pp. 141–150.

[11] T. W. Price, Y. Dong, and D. Lipovac, "iSnap: Towards Intelligent Tutoring in Novice Programming Environments," in *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2017, pp. 483–488.

[12] H. Kazi, P. Haddawy, and S. Suebnukarn, "Leveraging a Domain Ontology to Increase the Quality of Feedback in an Intelligent Tutoring System ," in *International Conference on Intelligent Tutoring Systems (ITS)*, 2010, pp. 75–84.

[13] M. Stanger and E. Martin, "The 50 best computer-science and engineering schools in America," http://www.businessinsider.com/best-computer-science-engineering-schools-in-america-2015-7/, 2015.

[14] N. Fraser, "Google Blockly: A Visual Programming Editor," http://code.google.com/p/blockly/, 2013.

[15] A. Renkl and R. K. Atkinson, "Structuring the Transition From Example Study to Problem Solving in Cognitive Skill Acquisition: A Cognitive Load Perspective," *Educational Psychologist*, vol. 38, no. 1, pp. 15–22, 2010.

[16] F. J. Rodríguez, K. M. Price, J. Isaac Jr., K. E. Boyer, and C. Gardner-McCune, "How Block Categories Affect Learner Satisfaction with a Block-Based Programming Interface," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 201–205.

[17] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent Tutoring Systems for Programming Education: A Systematic Review," in *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*, 2018, pp. 53–62.