PseudoBlocks: A Block-Based Gameboard and Interactive Sandbox for Constructing Pseudocode

Liu Jiang, Jared Wolens, Harry Gamble
Stanford University
450 Serra Mall
Stanford, CA
{liujiang, jwolens, hgamble}@stanford.edu

ABSTRACT
Metacognition is a key tenet of self-regulated learning. We introduce Pseudoblocks, a block-based gameboard accompanied by an interactive sandbox. Learners organize pre-fabricated physical blocks representing set actions to construct individual lines of pseudocode that form an outline to the game Brick Breaker. By mapping high-level block-based visual code to a more detailed text-based representation, Pseudoblocks helps learners transition from beginning to intermediate programming. Our approach prompts a process of reflection whereby students engage and disengage with their code to consider the holistic construction of their programming projects.

ACM Classification Keywords
H.5.m Information interfaces and presentation (e.g., HCI): Miscellaneous

Author Keywords
Metacognition; pseudocode; block-based programming; tangible user interface; algorithmic planning; design; learning.

1.INTRODUCTION

We present Pseudoblocks, a block-based gameboard coupled with an interactive sandbox that aims to solve three challenges that novice programmers face. Firstly, novice programmers have difficulty transitioning from visual programming languages like Scratch to more powerful text-based programming languages like Java. Pseudoblocks addresses this by mapping between high-level blocked-based code and a more detailed text-based representation. Secondly, novice programmers often do not participate in the first two stages of writing a program, analyzing the problem and designing a solution plan), thereby resulting in bugs in their code. Thus, Pseudoblocks focuses on the planning process that precedes programming, specifically the construction of pseudocode, as success or failure in this stage affects the subsequent steps of implementing, testing, and debugging an algorithm [5]. Thirdly, existing block-based programming tools for novices are often pure software and do not incorporate features that inspire reflection. In contrast, Pseudoblocks has both digital and physical elements and leverages tactility as a means of fostering learners' engagement with metacognition.

Through Pseudoblocks, learners build a pseudocode outline to the game Brick Breaker. Pseudocode methods are constructed by arranging pre-fabricated physical blocks, each representing a control structure, variable, or action, in three templates on the gameboard. The three templates increase in difficulty such that the learner forms basic individual functions in the first template, builds upon those functions and incorporates control structures in the second, and completes the whole pseudocode program in the third. As each block is placed on the gameboard, a more detailed text-based representation of the block's pseudocode label is generated in real-time in the interactive sandbox. Pseudoblocks encourages iteration by alerting learners when their block arrangements contain gross structural errors, thus minimizing debugging during the later stages of programming implementation. Once satisfied with their pseudocode, learners can then move to the provided starter code and utilize the pseudocode outline as a guide to programming their Brick Breaker project.

The three learning goals of Pseudoblocks are to transition learners from visual to text-based programming languages, make pseudocode an integral part of programming, and concretize the problem analysis and algorithmic planning process that precedes programming. By emphasizing the metacognitive process of solving computing problems rather than their solutions, Pseudoblocks encourages active experimentation, reflective observation, and abstract conceptualization [17].

2. BACKGROUND

Metacognition, a key tenet of self-regulated learning, increases content understanding of problem representation and facilitates greater transfer of problem solving skills [9, 13]. Previous studies have shown that whereas experts demonstrate metacognitive knowledge of the program task and possess ideal working strategies, novice programmers practice little advance planning and have opportunistic styles

CHI'14, April 26–May 1, 2014, Toronto, Canada.

DOI string from ACM form confirmation

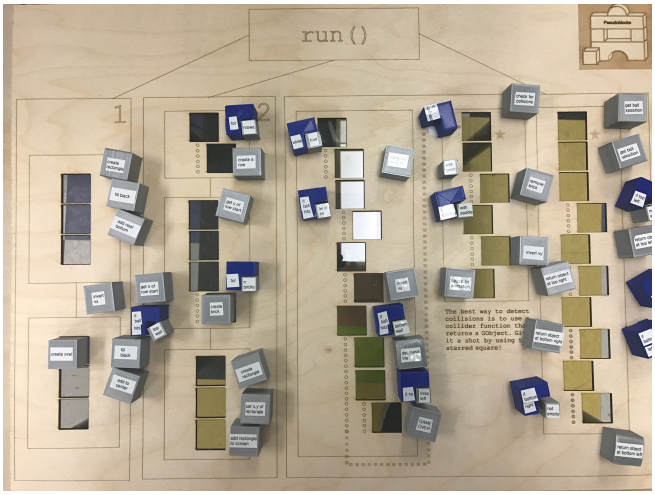

Figure 1. An overview of the PseudoBlocks gameboard. Gamepieces are pictured neighboring their slots for insertion.

of debugging [6, 23]. Pseudoblocks aims to bridge this gap by enabling novices to physically engage with metacognition and reflect on the holistic construction of their programs.

Pseudoblocks marries the theories of Constructionism and Constructivism. The model of learning is consistent with Constructionism as the learner consciously constructs externalized arrangements of blocks that give insight to her understanding of structuring pseudocode [1, 16]. In line with Constructivism, the cognizing learner actively reevaluates her schemata of how individual functions operate and how the program executes as a cohesive whole when she organizes and reorganizes the blocks [1, 22]. The learner's knowledge of functions, control structures, and decomposition are not just grounded in the specific context of Brick Breaker and can be extrapolated as more generalized principles that can be applied to any programming project.

Technologies that foster metacognition in novice programmers have existed since the 1980's. One of the earliest technologies was "Intelligent Tutoring Systems" (ITS), which were shown to improve student performance [3]. ProPL, an ITS focused on teaching programming, used dialogue to help novice programmers solve decomposition problems [11]. While ITS systems prompt planning and reflection, their tutoring-first tactics risk over-scaffolding the process of acquiring problem solving skills and may prevent novice programmers from independently generating solutions.

Another class of technologies which encourage metacognition are debuggers, which fall into one of two categories: static or dynamic [7]. While a static debugger analyzes code for syntax or logic errors, dynamic debuggers enable developers to control the execution of code and stop it at chosen breakpoints to examine highly localized behaviors [7]. However, static debuggers (e.g. Findbugs) are often analogous to spellcheck and provide the user with few learning opportunities to reflect whereas dynamic debuggers (e.g. gdb) may engage a user's critical reasoning skills but are inaccessible to novice programmers [7]. Pseudoblocks fills the gap between ITS and debuggers by affording novice programmers the physical experience of planning their code via the medium of pseudocode blocks.

Planning algorithmic design is an essential precursor to programming. Literature that investigates learners' performance under different programming preparation environments often compare graphical representations with pseudocode. In contrast to graphical representations like flowcharts, pseudocode accommodates the creation of complex programs by standardizing and simplifying the process of defining a complicated problem and determining the sequence of steps needed to solve it [11]. Graphical notations also lack the syntactical elements required to accurately represent and encapsulate the four computational concepts of sequence, iteration, selection, and recursion [18]. Because the ultimate desired end product of programming is a literate program, the textual nature of pseudocode makes it more appropriate than graphical representations for Pseudoblocks' target population of learners, who are transitioning from beginning to intermediate programming.

At the heart of Pseudoblocks's design is pseudocode, which serves as a high-level algorithmic description of a program. Pseudoblocks assists not only the design of algorithms but also the process of debugging, especially for programs that are large or complex. Because writing pseudocode results in better code, Pseudoblocks guides learners in building a pseudocode outline, thereby ensuring that they metacognitively engage in designing an algorithmic structure for their code [15]. In omitting programming languages' strict syntax, which beginner programmers often find challenging to master, Pseudoblocks provides an opportunity for learners to focus exclusively on the algorithmic logic of their pseudocode without having to acquire complex symbolic notation systems [3, 19, 24].

Pseudoblocks consists of real-world blocks that the learner organizes and structures into pseudocode. The benefits of physical programming environments to student learning of abstract concepts have been well-explored. As a specific example, Kwon (2012) discovered that controlling physical robots with tangible systems had a greater influence on learners' algorithmic thinking than their virtual counterparts [9]. One of the limitations of previous learning tools utilizing pseudocode is that they are purely virtual [4, 15, 18, 19]. We posit that the concrete existence of hardware blocks, as compared to software blocks, garners greater attentiveness from the learner. By connecting physical blocks, learners can better visualize how functions interact with each other and within the broader program.

Another feature that Pseudoblocks leverages is block-based programming, which has been shown to both increase novice students' interest in programming and maintain the motivation of more experienced students [13]. Despite the existence of block-based visual programming tools

Figure 2. Interactive sandbox that generates real-time pseudocode embedded with pre-fabricated template functions

like Scratch and App Inventor, such tools do not facilitate the creation of pseudocode or the planning that precedes programming, nor do they include reflective or monitoring features like annotations. Via fiducials, Pseudoblocks creates a direct one-to-one mapping between block-based visual code and text-based pseudocode, a gap identified in observations of Scratch [7]. Because the learner's goal is to assemble the blocks into the correct ordering of pseudocode, our tool is both a means of reflection for the learner as well as an embedding of the learner's reflection.

3. DESIGN

3.1. Features and Affordances

Pseudoblocks possesses a number of design features and affordances. Firstly, Pseudoblocks gamifies the act of creating pseudocode by adopting the cultural forms of the gameboard as well as the MatrixâĂŹs black-and-green coding scheme, which is associated with the advent of the digital age. The pre-marked spaces on the gameboard imply that a skillful strategy is required to determine the correct arrangement of blocks. Secondly, Psudoblocks leverages a block-based design that compiles linearly, similar to the manner in which code executes. Specifically, Pseudoblocks creates a direct mapping between higher level visual pseudocode and more detailed text-based pseudocode, which other block-based programming tools like Scratch do not afford. Furthermore, Pseudoblocks bridges the virtual with the physical world by effectively converting hardware pseudocode into a software representation. Furthermore, the tactile physicality of Pseudoblocks enables learners to better visualize how functions interact within the broader program while encouraging learners to more attentively arrange their blocks. The three types of blocks - control structures (if/for/while), variables, and functions - cluster sets of actions into categorical definitions. Likewise, in blackbloxing the syntax of Java, Pseudoblocks enables learners to focus on algorithmic design and not have to worry about symbolic notation.

The real-time generation of textual pseudocode coupled with the live feedback further the learning goal of metacognitive reflection. Pseudoblocks alerts learners when their pseudocode deviates heavily from the correct pseudocode

Figure 3. The three types of PseudoBlocks that help users chunk and cluster discrete actions.

outline, thus surfacing algorithmic design problems during this planning phase and minimizing debugging in later stages of programming. Rather than simply generating a static file of textual pseudocode when the learner has finished arranging all blocks, feedback is given as each block is placed. The learner can thus step back and challenge her schemata of how an individual function operates or how the program executes as a whole before placing the next block. As a result, the learner is able to engage in consistent reflection and avoid potentially overwhelming and deflating moments of signficiant error. In leaving room for learners to make minor errors, Pseudoblocks, in the spirit of critical pedagogy, acts less as a teacher that disseminates knowledge and analyzes performance and more as a collaborative, dialogic partner in the learning process.

Pseudoblocks has a low floor, wide walls, and a high ceiling. The walls are wide because learners can explore roughly 41! possible pathways to the solution. While there is one correct arrangement of the blocks, a high ceiling exists due to the provided Java environment and starter code, which enable learners to work on more sophisticated projects with near limitless potential. The scaffolding of Pseudoblocks creates a low floor and provides novice programmers with easy ways to get started. The board is structured like a lesson plan in that difficulty of pseudocode construction increases from the left template to the right, and successful completion of each template prepares learners for the one that subsequently follows. Similar to the step-by-step manner in which programs are constructed, the learner sequentially builds upon her pseudocode methods. The first template promotes the concept of creating standalone functions, the second requires merging functional blocks with control structures, and the final template explores decomposition and the construction of the pseudocode outline in its entirety.

3.2. Technical Construction

Pseudoblocks consists of an instructional guide, a Brick Breaker starter code file, a gameboard, blocks, a compiler, and an interactive sandbox for textual pseudocode generation. The instructional guide introduces learners to the technical affordances of Pseudoblocks and the high-level methods of Brick Breaker that they are expected to construct pseudocode for. Made of 3mm plywood, the lasercut gameboard consists of three templates and rests on top of tempered glass in a wooden frame approximately 2.5ft off the ground. The silver functional and variable blocks and the blue conditional blocks were modeled in SketchUp and 3D printed out of polylactic acid (PLA) using Cura software. Each block is tagged with

a reacTIVision fiducial and labeled with a line of high-level pseudocode that is associated with a distinct chunk of more detailed pseudocode. The blocks are placed by the learner in 1.05" x 1.05" slots in the three templates in order to form pseudocode methods. Conditionals and control structures are specifically constructed by merging L-shaped control statement blocks with smaller variable blocks; Pseudoblocks reads this combined structure as one action sequence. Processing, via the Logitech C922 webcam, uses a linear compilation structure that maps between the position of each block on the gameboard and the corresponding coordinate within the interactive sandbox. When a block is placed on the gameboard, TUIO detects its fiducial ID and generates a more detailed textual representation of the blockâĂŹs pseudocode label in real time in the sandbox. Because each fiducial has a set of fiducials that explicitly cannot follow, the compiler is able to detect sets of âĂIJimpossibleâĂİ block arrangements and alerts the user when errors exist. Once the learner structures all blocks in a template, she can manually âĂIJrunâĂİ her pseudocode, at which point each fiducial set is loaded onto a stack and popped off one-by-one. After achieving the correct block arrangement, the learner can move to the provided Eclipse starter code file and use the text-based pseudocode outline as a guide to programming Brick Breaker.

4. EVALUATION

We produced the current version of Pseudoblocks after several rounds of iterative development and user testing. We collected a variety of data from the user studies, including field notes, photos, and video and audio recordings.

4.1. Design Iterations

Between design iterations, we conducted user tests with novice programmers, Stanford d.school students, and Computer Science professors. We then made the following changes to our design:

Reorganized the gameboard.

Users noted that the gameboard felt "cluttered" and that the templates seemed "unrelated to each other" and involved the completion of "disparate tasks." To address this feedback, we then restructured the gameboard such that the templates cohesively build upon each other and that the difficulty of pseudocode construction increases from left to right.

Increased the block size.

Previously, the size of our blocks was 0.5" by 0.5" but several users commented that they wanted to be able to "feel the blocks in [their] hands." We therefore increased the block size to 1" x 1" in order emphasize the tactile physicality of Pseudblocks. With the larger block size, learners can better visualize how functions interact within the broader program and are encouraged to arrange the blocks with more attentiveness.

Simplified the compiler to a linear design.

The compiler previously utilized a recursive tree structure and attempted to create a new grammar. Users noted that the compiler felt "convoluted." We realized that a tree structure was unnecessary and nonadditive to our learning goals. After much thought, we opted for a sequential design, which is more simple and mirrors the linear style in which code executes.

Figure 4. User tester completing the end of Task 3 arranging the final configuration of blocks.

Generated textual pseudocode in real-time.

In an earlier version of Pseudoblocks, the only feedback came in the form of a static pdf file of the textual pseudocode, which was created once the learner completed her block arrangements. Users who received files with numerous mistakes highlighted commented that they felt "deflated" and "overwhelmed." They also noted that did not know "where to start debugging" and could not determine "which errors were more high priority than others." We thus strove to make Pseudoblocks seem less like a summative assessment that tests for minute errors and more like a collaborative partner throughout the planning process. We accomplished this by generating pseudocode in real-time as each block is placed. Thus, learners can constantly observe and reflect on the effects of changing block placements on the textual pseudocode itself. This live render feature can also challenge learners' schemata of how individual functions operate and how a program executes as a whole.

5. FINAL DESIGN FINDINGS

6. CONCLUSION & FUTURE WORK

Preliminary results point to the success of Pseudoblocks in helping novice programmers engage in the planning process of creating pseudocode, reflect metacognitively, and transition from visual to text-based programming languages. However, there are notable opportunities for future work. The next iteration can enable students to construct their own pseudocode labels. Furthermore, the live feedback feature of the interactive sandbox can be extended such that future Pseudoblocks learners can visualize the actions of functions as they create them. Pseudoblocks currently targets novice programmers who program individually, but user testing can be conducted with more diverse populations and separate versions can be created for intermediate or advanced programmers, pair programmers, and classroom or lecture environments. Ultimately, Pseudoblocks can expand beyond Brick Breaker and be made project-agnostic.

7. ACKNOWLEDGMENTS

We thank all the Beyond Bits and Atoms staff, and Paulo Blikstein, for supporting and aiding our research. Thank you to all participants of our user research and those that provided valuable feedback towards our design iteration.

REFERENCES

[1] Ackermann, E. (2001). Piaget's constructivism, Papert's constructionism: What's the difference. Future of learning group publication, 5(3), 438.

[2] Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. The Journal of the Learning Sciences, 4 (2), 167-207.

[3] Brusilovsky, P., Calabrese,E., Hvorecky,J., Kouchnirenko, A. and Miller, P. (1997), Mini-languages: a way to learn programming principles, International Journal of Education and Information Technologies, 2 (1), pp. 65-83.

[4] Crews, T. and Ziegler, U. (1998), The flowchart interpreter for introductory programming courses, Proceedings of FIE '98, pp. 307-312.

[5] Eteläpelto, A. (1993). Metacognition and the expertise of computer program comprehension. Scandinavian Journal of Educational Research, 37(3), 243-254.

[6] Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., and Zander, C. (2010). Debugging From the Student Perspective, IEEE Transactions on Education, 53(3), 390-396.

[7] Grover, S., Pea, R., & Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. Computer Science Education, 25(2), 199-237

[8] Howard, B. C., McGee, S., Shia, R., & Hong, N. S. (2001). Computer-based science inquiry: How components of metacognitive self-regulation affect problem-solving

[9] Kwon, D., Kim, H., Shim, J., & Lee, W. (2012). Algorithmic bricks: A tangible robot programming tool for elementary school students. IEEE Transactions on Education, 55(4), 474-479.

[10] Lane, C. H., & VanLehn, K. (2007). Teaching the tacit knowledge of programming to novices with natural language tutoring. Computer Science Education, 15(3), 183-201

[11] Larson, J. (1986). Problem solving with generic algorithms and computers.

[12] Lehrer, R., Lee, M., & Jeong, A. (1999). Reflective Teaching of Logo. The Journal of the Learning Sciences, 8(2), 245-289.

[13] Mihci, C., & Ozdener, N. (2014). Programming education with a blocks-based visual language for mobile application development. International Association for the Development of the Information Society.

[14] Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., & Pugh, W. (2008). "Using Static Analysis to Find Bugs, IEEE Software, 25(5), 22-29.

[15] Olsen, A. L. (2005). Using pseudocode to teach problem solving. Journal of Computing Sciences in Colleges, 21(2), 231-236.

[16] Papert, S. (1980). Mindstorms: Children, computers, and powerful ideas. New York: Basic books.

[17] Parham, J. R. A cognitive model for problem solving in computer science.

[18] Roy, G. G. (2006). Designing and explaining programs with a literate pseudocode. Journal on Educational Resources in Computing, 6(1).

[19] Scanlan, D., & Clark, L. (1989). An empirical investigation of flowchart preference. Journal of Computers in Mathematics and Science Teaching, 8(2), 56-64.

[20] Shum, S., & Cook, C. (1994). Using literate programming to teach good programming practices. ACM SIGCSE Bulletin, 26(1), 66-70.

[21] Siozou, S., Tselios, N., & Komis, V. (2008). Effect of algorithms' multiple representations in the context of programming education. Interactive Technology and Smart Education, 5(4), 230-243.

[22] Von Glasersfeld, E. (1989). Constructivism in Education. The International Encyclopedia of Education, 162-163.

[23] Webb, N., Ender, P., & Lewis, S. (1986). Problem-Solving Strategies and Group Processes in Small Groups Learning Computer Programming. American Educational Research Journal, 23(2), 243-261.

[24] Wyeth, P. (2008). How Young Children Learn to Program with Sensor, Action, and Logic Blocks. The Journal of the Learning Sciences, 17(4), 517-550.