Approaches for Teaching Computational Thinking Strategies in an Educational Game: A Position Paper

Aaron Bauer, Eric Butler, Zoran Popović
Center for Game Science
Computer Science & Engineering
University of Washington
Seattle, WA 98195
Email:{awb, edbutler, zoran}@cs.washington.edu

***Abstract*—Computer science is expanding into K12 education and numerous educational games and systems have been created to teach programming skills, including many block-based programming environments. Teaching computational thinking has received particular attention, and more research is needed on using educational games to directly teach computational thinking skills. We propose to investigate this using *Dragon Architect*, an educational block-based programming game we are developing. Specifically, we wish to study ways of directly teaching computational thinking strategies such as divide and conquer in an educational game, as well as ways to evaluate our approaches.**

***Keywords*—block-based programming, game-based learning, computational thinking, CS education**

I. Introduction

Teaching *computational thinking* has been a focus of recent efforts to broaden the reach of computer science education, including those using block-based programming environments. In their review of recent literature on teaching computational thinking, Lye and Koh [1] use Brennan and Resnick's definition of computational thinking as consisting of *concepts*, *practices*, and *perspectives* [2]. Concepts are basic programming ideas (such as variables, conditionals, and loops), practices are the problem-solving strategies used while programming (such as "being incremental and iterative" or "using abstraction and modularization"), and perspectives are the relationships with the wider technological world. Lye and Koh conclude that more research is needed on teaching computational thinking *practices* in particular. How to *directly* teach such skills in is an open problem, including in the context of a block-based programming environment. Teaching skills directly involves providing appropriate guidance and scaffolding to help students acquire those skills, rather than just exposing them to environments where those skills are necessary.

In this paper we propose investigating directly teaching computational practices in an educational block-based programming game called *Dragon Architect*[1]. We briefly discuss existing work on teaching computational thinking and the direct teaching of problem-solving strategies. We describe ways this might be attempted in *Dragon Architect*, as well as ways we might evaluate such work.

[1]Available at http://centerforgamescience.org/portfolio/dragon-architect

II. Game Description

Before we discuss approaches for teaching computational thinking, we describe basic information about *Dragon Architect* to provide context for the discussion. In *Dragon Architect*, players write code to control a dragon that builds 3D structures in a cube world. Our game, in development since spring 2014, is played in a web browser. Similar to other programming environments, the user interface is separated into two parts: an area where the player can assemble their code and a visualization of the 3D environment their code affects (see Figure 1). The game uses the block-based programming library Blockly [3] for inputing code.

The player can write programs to move the dragon in three dimensions and have the dragon place and remove cubes of various colors. In addition to blocks that control the dragon directly, players can use definite loops and procedures (see Figure 2). As players progress through the game, they alternate between short sequences of puzzles with a specific goal and a constrained set of available code blocks and an open-ended sandbox. The game begins with puzzles that introduce the idea of assembling and running code, as well as the code blocks for moving the dragon and placing cubes. After that, the player can creatively experiment and build in the sandbox and complete other puzzle sequences to make more code blocks available, switching between sandbox and puzzles at any time. In this way, the language the player uses to write instructions for the dragon gradually expands as the player advances.

The popularity and broad appeal of *Minecraft* [4] motivated our use of a 3D grid world in which the player's programs could place cubes. This choice also makes it natural to extend our game in the future with exploration, more complex interaction with the environment, or players working together in a shared world. Our playtests with *Dragon Architect* have shown the premise of programming a dragon in a *Minecraft*-like world appeals to younger players of all genders. Common sandbox activities have included making the dragon travel very long distances, building big and impressive towers, and spelling one's name out of cubes.

III. Teaching Computational Thinking Strategies

Many have studied how to increase the presence and effectiveness of computational thinking in computer science educa-

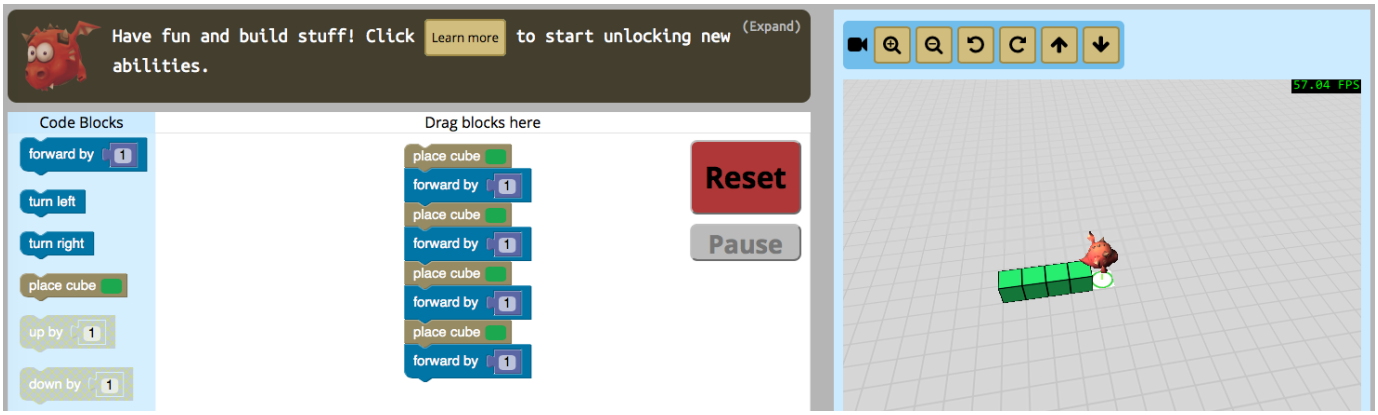

Fig. 1. The player assembles code to control the dragon on the left side, and the dragon and world it inhabits are visualized on the right side. Only a few different code blocks are available to the player initially, and more are unlocked by completing guided puzzles.

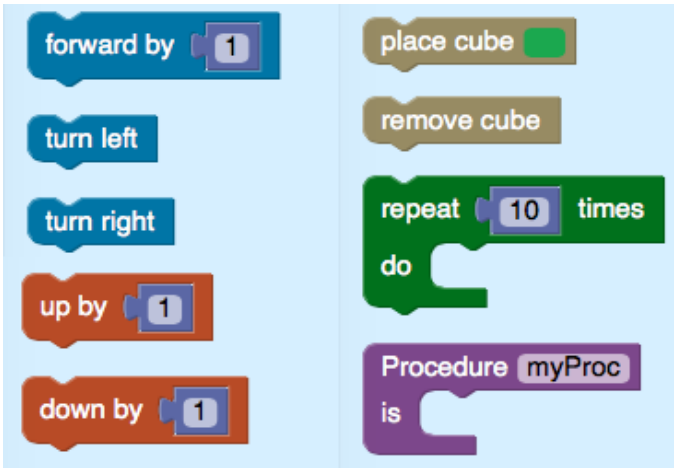

Fig. 2. The programming elements available in *Dragon Architect*, which include moving the dragon, placing blocks, definite loops, and procedure definitions.

tion and education in general (e.g., Barr and Stephenson [5], Grover and Pea [6]). Furthermore, educational games and other systems often have teaching computational thinking as an explicit goal (e.g., Weintrop and Wilensky [7], Kazimoglue et al. [8]) or have a computational thinking framework built around them (e.g., Gouws et al. [9], Computational Thinking with Scratch [10]). A recent review of the literature on teaching computational thinking found that additional empirical research is needed, especially in the case of computational thinking practices [1]. We believe the use of educational games in particular to teach computational thinking skills deserves to be the focus of more empirical work.

Specifically, we propose to investigate directly teaching computational thinking strategies in *Dragon Architect*. Simply playing in a computational environment where these strategies are necessary is unlikely to teach students such complex skills [11]. Instead, we must address how to directly teach computational thinking skills by investigating which guidance is effective and how it is best deployed in an educational game.

One computatinal thinking skill of interest is the identification and application of problem-solving strategies. A great deal of recent education research suggests that "curricula can model such strategies for students" and that appropriate guidance can "enable students to learn to use these strategies independently" [12]. Mayer and Wittrock call attention to the substantial evidence in the education literature for teaching what they call *domain-specific thinking skills* and *metacognitive skills* [13]. The former would include the ability to use a strategy like divide and conquer, and the latter would include knowing when and where to employ that strategy. In both cases, Mayer and Wittrock describe studies (for non-computer science domains) that have shown teaching these skills directly can improve learning and performance. It is an open question whether this can be applied to teaching computational thinking in a game.

One computational thinking strategy we intend to focus on in *Dragon Architect* is *divide and conquer*. One potential approach is to lead the player though a top-down deconstruction of building a castle in order to model iteratively subdividing a large problem into more manageable subproblems. The player is presented with a single code block that builds an entire castle, but discovers the construction has a number of flaws. The next several puzzles each decompose some part of the flawed program in order to give the player a chance to repair it. For example, to enable the player to give the castle the correct number of walls and towers, the castle code block is split into a tower block and a wall block that the player uses to write a corrected castle procedure, as shown Figure 3.

A companion approach is a gradual bottom-up progression modeling combining the blocks currently available to the player into more sophisticated constructs. For example, the player is tasked with writing a program to place a line of cubes. When this is completed, the player is awarded a new kind of block that by itself places a line of cubes (i.e., a block encapsulating the player's previous program). Subsequent puzzles ask the player to use the line block to construct other, more complicated structures, each time granting the player a single, encapsulating block.

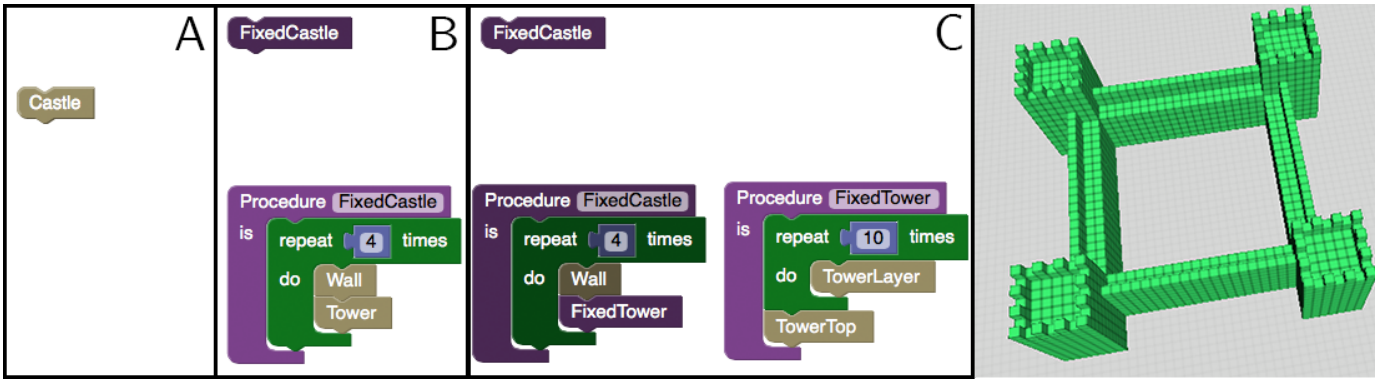

Fig. 3. The code required by a progression of levels demonstrating the strategy of divide and conquer. In A, the player uses a single code block to build an entire castle. Then, in B, the player is given an empty `FixedCastle` procedure, which they must fill with the appropriate number of wall and tower blocks. Finally, in C, the player is given a completed `FixedCastle` procedure and must fill in the `FixedTower` procedure as shown. The final completed castle is shown on the right.

Evaluating learning outcomes and other effects of these or similar approaches is a tremendous challenge. Computational Thinking with Scratch proposes three kinds of assessment: (1) artifact-based interviews, (2) design scenarios, and (3) learner documentation [10]. The interviews are intended to engage the learner in a conversation about the artifacts they have created and the practices they used. Design scenarios are a sequence of projects that challenge the learner to critique, extend, debug, and remix existing code. Finally, learner documentation focuses on engaging learners in reflection about their learning, and examples include keeping a journal, commenting code, and creating a visual walk-through of a project using screen capture software. We believe these approaches to be promising, and worthy of further study.

In addition to design scenarios, we propose that more general kinds of in-game assessments could be useful. Like Scratch, *Dragon Architect* provides users a place for unstructured creative exploration. The effectiveness of an attempt to teach computational thinking strategies could be assessed by comparing the programs written by those who completed the relevant puzzles to those who did not (after controlling for time played and differences in programs prior to completing the puzzles). A variety of other in-game metrics could contribute to an assessment including a player's solutions to specific challenges, time taken to complete puzzles, etc.

On-paper assessments (given as pre-test and post-test) could also serve as an evaluation. Computational thinking skills might be assessed through language-independent assessments of computer science knowledge [14] or general problem-solving assessments such as those developed by the Program for International Student Assessment [15]. Though neither of these assessments are explicitely targeted at computational thinking, they both contain items involving computational thinking skills. Finally, Grover and Pea suggest "academic talk" (i.e., student development and use of computational language) could be leveraged as an additional assessment of computational thinking [6].

IV. Acknowledgments

This work was supported by the Office of Naval Research grant N00014-12-C-0158, the Bill and Melinda Gates Foundation grant OPP1031488, the Hewlett Foundation grant 2012-8161, Adobe, and Microsoft.

References

[1] S. Y. Lye and J. H. L. Koh, "Review on teaching and learning of computational thinking through programming: What is next for k-12?" *Computers in Human Behavior*, vol. 41, pp. 51–61, 2014.

[2] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the American Educational Research Association*, 2012.

[3] "Blockly," https://developers.google.com/blockly/, accessed: 2015-02-18.

[4] Mojang AB, "Minecraft," 2011.

[5] V. Barr and C. Stephenson, "Bringing computational thinking to k-12: what is involved and what is the role of the computer science education community?" *ACM Inroads*, vol. 2, no. 1, pp. 48–54, 2011.

[6] S. Grover and R. Pea, "Computational thinking in k–12 a review of the state of the field," *Educational Researcher*, vol. 42, no. 1, pp. 38–43, 2013.

[7] D. Weintrop and U. Wilensky, "Robobuilder: a computational thinking game," in *SIGCSE*, 2013, p. 736.

[8] C. Kazimoglu, M. Kiernan, L. Bacon, and L. Mackinnon, "A serious game for developing computational thinking and learning introductory computer programming," *Procedia-Social and Behavioral Sciences*, vol. 47, pp. 1991–1999, 2012.

[9] L. A. Gouws, K. Bradshaw, and P. Wentworth, "Computational thinking in educational activities: An evaluation of the educational game lightbot," in *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '13. ACM, 2013, pp. 10–15.

[10] "Computational thinking with scratch," http://scratched.gse.harvard.edu/ct/index.html, accessed: 2015-07-23.

[11] R. E. Mayer, "Should there be a three-strikes rule against pure discovery learning?" *American Psychologist*, vol. 59, no. 1, p. 14, 2004.

[12] National Research Council, *Report of a Workshop on the Scope and Nature of Computational Thinking*. National Academies Press, 2010.

[13] R. E. Mayer and M. C. Wittrock, "Problem solving transfer," in *Handbook of educational psychology*, D. C. Berliner and R. C. Calfee, Eds. Routledge, 1996.

[14] A. E. Tew and M. Guzdial, "The fcs1: a language independent assessment of cs1 knowledge," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 111–116.

[15] "Program for international student assessment," http://www.oecd.org/pisa/, accessed: 2015-07-23.