Children Learning Computer Science Concepts via Alice Game-Programming

Linda Werner
University of California
Santa Cruz, CA
linda@soe.ucsc.edu

Shannon Campe
ETR Associates
Scotts Valley, CA
shannonc@etr.org

Jill Denner
ETR Associates
Scotts Valley, CA
jilld@etr.org

ABSTRACT
Programming environments that incorporate drag-and-drop methods and many pre-defined objects and operations are being widely used in K-12 settings. But can middle school students learn complex computer science concepts by using these programming environments when computer science is not the focus of the course? In this paper, we describe a semester-long game-programming course where 325 middle school students used Alice. We report on our analysis of 231 final games where we measured the frequency of successful execution of programming constructs. Our results show that many games exhibit successful uses of high level computer science concepts such as student-created abstractions, concurrent execution, and event handlers.We discuss the implications of these results for designing effective game programming courses for young students.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education – *Computer science education.* D.3.3 [Programming Languages]: Language Constructs and Features

General Terms
Experimentation, Languages, Measurement

Keywords
Assessment, Game-Programming, Middle School, Pair Programming, Alice

1. INTRODUCTION
Widely used initial programming environments like Alice and Scratch use drag-and-drop methods and provide many pre-defined objects and operations. Researchers and educators at the K-12 level are hopeful that these environments will teach computer science (CS) even when being used in other subject areas such as math, physics, biology, earth science, history, and language arts [12]. Since CS is not (yet) one of the K-12 core topics [4], using these programming environments in the core curriculum holds promise for introducing CS concepts to a broader and more diverse group of students. But there is little research on whether students as young as those in middle school will learn complex computer science concepts from using Alice when CS is not the focus of the course. In this paper, we describe a game-programming course called iGame that involved 325 middle school students using Alice across four semesters (in and after school). iGame was part of a study where we addressed many research questions including the following:

- What are the CS concepts that are accessible with Alice programming constructs?
- What are the Alice programming constructs that are used by middle school students making games without required game specifications?
- Which of these programming constructs are used successfully by middle school students?

To address these research questions, we reviewed a subset (approximately 20) of student-created games to identify the CS concepts that are accessible, and then analyzed all 231 games by counting the frequency of inclusion and successful execution of these programming constructs. In this paper, we first discuss prior related work, followed by a description of the iGame course including participant demographics, procedures of the study, and the game analysis process. Next, we list the results of the analysis of the games and discuss these results. The paper concludes with a discussion of future work.

2. PRIOR WORK
Studies of programming environments for K-12 audiences focus primarily on psychological and educational factors, with less attention to the software engineering perspective [11]. Two edited books from the late 1980s include research on how novices learn to program [13][10]. It has only been in recent years that we have seen studies of what students learn when using programming environments designed for youth such as Alice, Scratch, Kodu, Stagecast Creator, and Agent Sheets. We briefly describe some of the relevant findings from these studies.

Kelleher et al worked with Alice 2.0 and Storytelling Alice, a version of Alice developed to motivate young women to program. She reported that young women using Storytelling Alice and Alice were equally successful at learning basic programming constructs [6]. A large study of high school students in Taiwan found that Alice was more effective at teaching comprehension of fundamental programming concepts than C++[16]. Maloney et al [9] analyzed 536 Scratch projects created by youth aged 8-18 during an 18-month period at an after-school community center. Approximately 20% of the projects had no 'scripts,' meaning no programming. Of the remaining 425 projects, 88% exhibit scripts running in parallel, and more than 50% included user interaction and the use of looping constructs. Another study of Scratch use among 5th grade students found that the most common

SIGCSE'12, February29-March 3, 2012, Raleigh, N. Carolina, USA.

programming concepts included Boolean expression, conditions, and loops. Less frequent were variables and events [2]. Stolee and Fristoe [14] analyzed 346 Kodu programs created by Xbox users and shared on the Xbox Live Community and reported on program constructs that represent understanding of CS concepts. The age range of these Kodu programmers' is unknown. Rodger et al [12] analyzed all Alice programs created by approximately 35 middle school students who attended one of a series of one-week summer camps. These researchers measured CS topics used in the programs. They also included a few programming patterns in their list of constructs: timer and score. Unfortunately, they reported no details about whether the constructs used in these programs, which mapped to specific CS topics, were on executable program paths (i.e., "reachable"). Werner et al [17] analyzed 23 Alice games created by middle school aged children enrolled in a summer camp. They reported on programming construct use representative of specific CS concepts including those exhibiting abstraction and various methods of program flow control necessary for algorithmic design. They reported high use of event handlers and parallelism but again did not report if these constructs were found in reachable code. Denner et al [5] analyzed 108 Stagecast Creator games created by primarily low income Latina middle school girls for evidence of characteristics in three categories: programming, organizing and documenting, and designing for usability. They report 82% of the programs exhibited conditional execution, 36% exhibited parallel execution, 41% had no unreachable code, but only 4% of the program used global variables. Koh et al [7] developed an automatic method to semantically analyze AgentSheets programs, another programming environment used in K-12, producing a 'computational thinking pattern graph' for each program. These graphs can be superimposed on one another to allow for visual comparison of the corresponding programs. They have used this method to analyze approximately 2500 student projects looking for patterns such as counting, generating, absorption, collision, etc, but there are no published reports of their analysis of this large set of programs.

The studies reviewed above provide some indication of the kinds of programming constructs and the CS concepts that are accessible with these types of programming environments. The current paper extends this prior work by looking not only at which programming constructs are used, but whether they are used successfully, and whether they were provided by the programming environment or created by the students.

Some of the prior studies of programming environments designed for youth focused on making games, while others used other types of projects. However, there is enough evidence to suggest that programming a game holds particular promise for engaging students in higher order thinking and comprehension [5][3][15]. Therefore, in the current study, we focus on programming games.

3. iGAME

3.1 Participants

The games described in this paper were collected over 2 years as part of a study of how game creation and pair programming can promote computational thinking in middle school students. A total of 325 students with parental consent voluntarily participated in study classes at seven public schools along the central California coast. Here we report on 231 games created individually or as part of a programming pair. In year 1, 90 students participated in after-school classes, and in year 2, 37 participated in after-school and 198 in elective, in-school technology classes.

Of the 325 students, 37% were female; they ranged in age from 10-14 years (mean=12), 45% were white, 37% were Latino/a, 73% spoke English or primarily English at home. Among the 274 for whom we had parent report, 27% had mothers with educational levels of high school or lower, and 38% of mothers had completed a university degree (since mothers are more likely to complete the consent forms, we rely on their educational levels).

3.2 Procedure

Two of the programming environments in the Alice series developed at Carnegie Mellon University were used in our study. In year 1, students used Storytelling Alice (SA), and in year 2 they used Alice 2.2 due to the limited use of PCs (which is necessary to run SA) at our partner schools. In the rest of this paper we will refer to both SA and Alice 2.2 as Alice unless it is necessary to distinguish between the two. Alice allows users to control characters in 3D environments using drag-and-drop programming and using a language that is closely related to Java and many other modern imperative programming languages. Most code is written in the methods of objects that have properties that store state and functions that return values. Each property, method, and function is attached to an object, with World being the global object. The event system in Alice is primarily used to handle user interactions, such as mouse clicks, although it can also handle in-World events, such as a change in a variable's value.

Classes were randomly assigned to a pair programming or solo programming condition. Pairs and individual students engaged with CS concepts in a three-stage acquisition progression called Use-Modify-Create [8] over approximately 20 hours during a semester. In the first half of the semester, pairs and individual students worked through a series of self-paced instructional exercises built to provide scaffolding, which we call "challenges." During the last half of the course, the students freely designed and developed their own games. Figure 1 shows the Alice programming environment and a student-created game titled "Wizard Duel" where wizards throw lightning bolts and fire balls at each other to acquire points to win. Most students completed 8 to 10 challenges, though some completed all 17. In our courses where pair programming was used, two students shared one computer to both work through the challenges and create their game, with one driving (controlling the mouse and keyboard) and the other navigating (checking for bugs, consulting resources, and providing input). Students were asked to reverse roles approximately every 20 minutes. In our courses where pair programming was not used, each student worked individually on the challenges and their game.

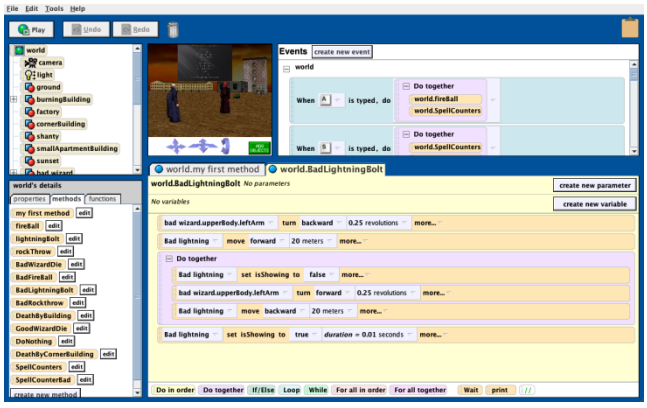

Figure 1: Student-created 'Wizard Duel' game.

3.3 Game analysis process

Five undergraduate CS students were trained to examine the 231 games (66 SA games and 165 Alice 2.2 games) and count programming constructs the middle school students used, and which use resulted in no failures. Our analysis is similar to that done by Rodger et al [12] of Alice programs, Maloney et al [9] of Scratch programs, and Stolee and Fristoe [14] of Kodu programs with one big difference: we look for **reachable** (i.e., on a path that is executable) instances of important programming constructs. Only one of the studies described in Section 2, the study by Denner et al [5] of 108 Stagecast Creator programs created by middle school students, looked at issues of reachability. They found 59% of the Stagecast programs contained unreachable code. From our experience teaching middle school students other programming environments, we found it common that students would include code that is never executed. For this study, if the Alice construct is reachable, then we determine whether execution of that construct causes abnormal program termination. All games were analyzed by at least two of the five undergraduate students and all discrepancies were discussed until unanimous agreement was reached, or the discrepancy was resolved by one of the authors. If we were only looking for the presence of these constructs and not reachability, this analysis could be automated. Due to determining whether the constructs in the abstraction and modeling category were created by the student/pair ('student-created') and testing if the instance of each reachable construct in each category did not cause abnormal program termination when executed ('no failure'), human observation was required.

4. RESULTS AND DISCUSSION

The important programming constructs in our analysis are divided into three categories: abstraction and modeling (see Table 1), control structures (see Table 2), and event handlers (see Table 3).

Table 1. Abstraction and Modeling constructs

Construct	# (%) student-created	# (%) reached	of # reached, #(%) no failure
Methods *student created only*	164 (71%)	138 (60%)	135 (98%)
Parameters *student created only*	1 (0.4%)	1 (0.4%)	1 (100%)
Functions	1 (0.4%)	106 (46%)	100 (94%)
Variables non-list	72 (31%)	62 (27%)	53 (86%)
Variables list	9 (4%)	5 (2%)	4 (80%)

Table 2. Control Structure constructs

Construct	# (%) reached	of # reached, #(%) no failure
If/else using variable or function in Boolean expression	67 (29%)	63 (94%)
If/else using true/false in Boolean expression	14 (6%)	8 (57%)
Nested if/else	17 (7%)	15 (88%)
Loop	32 (14%)	27 (84%)
While	38 (17%)	32 (84%)
Do in order	54 (23%)	50 (93%)
Do together	145 (63%)	136 (94%)
For all in order	2 (1%)	1 (50%)
For all together	4 (2%)	4 (100%)

Table 3. Event Handler constructs

Construct	# (%) reached	of # reached, #(%) no failure
Method other than 'my first method' at start	37 (16%)	32 (87%)
When a NUMBER is typed	28 (12%)	27 (96%)
When a LETTER is typed	101 (44%)	98 (97%)
When SPACE is typed	39 (17%)	39 (100%)
When ENTER is typed	13 (6%)	13 (100%)
When SINGLE ARROW is typed	34 (15%)	31 (91%)
When ANY KEY is typed	6 (3%)	2 (33%)
When the mouse is clicked on object	102 (44%)	99 (97%)
Let ARROW KEYS move object	100 (44%)	100 (100%)
Let MOUSE move list object	1 (0.4%)	1 (100%)
While something is true	10 (4%)	9 (90%)
When something becomes true	38 (17%)	36 (95%)
When a variable changes	2 (1%)	1 (50%)
Let the MOUSE orient the camera	0 (0%)	0 (0%)
Let the MOUSE move the camera	8 (4%)	8 (100%)

Tables 1, 2, and 3 list the programming constructs for each category, the number of games for which an instance of that programming construct was 'student-created', 'reached', and 'reached with no failure', and the percentages of the total number of programs for each of those conditions. For the abstraction and modeling constructs, we report data for all three categories ('student-created', 'reached, 'reached with no failure). For control structure and event handler constructs, we report data for only two categories, 'reached' and 'reached with no failure.' Only these two categories are important for control structures because students don't 'create' these constructs; they take existing instructions and program them to execute actions in their games. For event handler constructs, Alice provides handlers; students only have to select and use them correctly.

It is important to note that only instances of 'Methods' and 'Parameters' that were student-created (see Table 1) were counted. All students experimented with the set of built-in

methods ('move', 'turn', 'resize', 'say', 'play sound', 'move to', etc.) and common parameters of these methods ('duration', 'style', etc.) that come standard in all gallery objects. We were interested in whether or not a game has a student-created method (at the global level or local to an object) and any student-created parameters. We found that the majority (71%) of games contained a student-created method; in 60% of the games a student-created method was reachable; and in 98% of the games where at least one student-created method was reachable, it executed without failure.

Only one Alice 2.2 game contained a student-created method with a parameter. With student-created parameters, very complex abstractions are possible. This topic was covered, with other topics, in the last challenge; it is not surprising there are limited examples of this construct in the games since many students did not work through a number of the latter challenges. The lack of exposure to latter challenges and limited practice to repeat use of the construct was also evident in the creation and use of functions. Students used built-in functions that were reachable in almost half of the games (46%). Of the games with a reachable built-in function, 94% of them executed without failing. Only one student created their own function using Alice 2.2 but it was not used on an executable program path. Figure 2 shows program code from a student game containing the most commonly found use of a built-in function - to implement collision detection in Alice. There was a large difference in the percentage of games containing the use of functions between SA (used in year 1) and Alice 2.2 (used in year 2). 33% of the SA games used functions but 51% of the Alice 2.2 games used functions. The reason for this difference may be due to the addition of a required challenge that taught the proximity function into the year 2 curriculum. Figure 3 shows another example of the use of the proximity function (to execute a method containing multiple instructions).

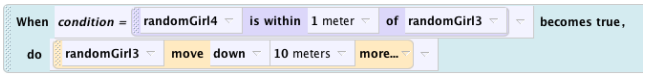

Figure 2: An event handler with a built-in proximity function.

Students often created their own non-list variables (32% of the games had at least one instance) and 27% of the games used either their own or built-in non-list variables in reachable code. Non-list variables were used without failure in 85% of those games in which they were found in reachable code. Students often used non-list variables to implement patterns such as counters and timers in their games. This is another area where there was a large difference between games created using Alice 2.2 and SA due to making a formerly optional challenge a required one in year 2: 32% of the Alice 2.2 games (year 2) contained one or more uses of non-list variables; only 15% of the SA games (year 1) contained one or more uses of non-list variables. Only a few games contained successfully-used student-created or built-in variable lists. More often, students repeated code segments for each individual object rather than building a list of objects and using the special list processing constructs of 'For all together' or 'For all in order'.

Students frequently used If/Else conditional statements using variables or functions in the Boolean expression (29% of the games with at least one of these statements in reachable code, 94% of those successfully used). The If/Else conditional statements using true or false in the Boolean expression were not frequently used. If an instance of this was found in a student's program and there were no instructions to execute in the body of the If/Else statement, it was counted as an unsuccessful use. 43% of the uses of this construct were unsuccessful. Our hypothesis is that the student tried to use this construct, couldn't figure it out or changed their mind around its use, but didn't remove it from their program because the presence of this construct did not affect execution.

More uses of looping constructs were found in year 2 Alice 2.2 games (17% for the loop construct, 21% for the while construct) than in year 1 SA games (6% for the loop construct, 6% for the while construct). This is likely a result of students programming more 'timers' in their games which typically include the use of a while loop and a non-list variable. This difference could have been due to moving the challenge that taught timers to earlier in the curriculum for year 2.

At least one instance of the 'Do together' parallel computing construct was found in most games: 63% of the games had at least one reachable instance and 94% of those games have a use that is successful. This is suggestive of the ease of use of the parallel programming construct, 'Do together'. Students appear to understand its use and want to use it to produce desired game effects. Less commonly used was the 'Do in order' sequential processing construct. Sequential instruction execution is the default mode of execution. The 'Do in order' construct is required in those instances when only one instruction is allowed, such as an instruction in the body of an If/Else instruction or an instruction in an event handler. Students could create a method and place a call to the new method in those instances.

Most of the students used some kind of event handler in their games with successful execution (85%). The most commonly used events were "when a letter is typed" (present in 44% of the games, 97% of the games where it is present execute the construct instance without failure), "when a mouse is clicked on an object" (44%, 97%) and "let arrow keys move object" (44%, 100%). Figure 3 shows the student game "Bunny Run," with the use of a 'while something is true' event. In this example, the student wants the game to be over when any of the hawks fly close to the rabbit. We did not see many uses of this type of event handler; only nine games used this successfully. This is an example of a very engaging game, that does not include the use of explicit conditional statements (ie., If/Else) or variables of which we may expect to see in games. Some of the games (16%) also contained code that calls a different method when their game starts (method other than 'My first method' when program starts). None of our instructional materials demonstrate these ideas. These uses may be representative of how some students are intrepid users of computer technology and discover new and interesting ways to control their games in different ways.

One result that holds for almost all Alice programming constructs: if a student used a programming construct, it was generally used correctly. This result is important because "it will save other researchers … from doing this type of laborious analysis when automated analysis could suffice.[1]"

Rodger et al [12] reported that 100% of their students used built-in methods in their programs but not quite 50% of their students created methods. Our numbers reflect a higher percentage of student-created method use. Rodger's summer camps were only one week long which may have contributed to the differences found in students creating methods. However, this can't account

[1] These quoted words are thanks to an anonymous reviewer.

for the differences in student-created function use or list variable use. Approximately 60% of their students used built-in functions and almost 15% of their students created their own functions. Forty-five percent of their students created lists and 40% used instructions for list processing. These topics were covered in Rodger's summer camps. Our students did not create and use their own functions and did not create and use list variables. Perhaps these differences are because these two topics were covered in latter challenges that most students did not complete. Further study needs to be done to confirm this hypothesis. We are in the process of determining the number of challenges completed by each student so we can know whether the number and specific constructs within the challenges affects the programming constructs a student used in their game.

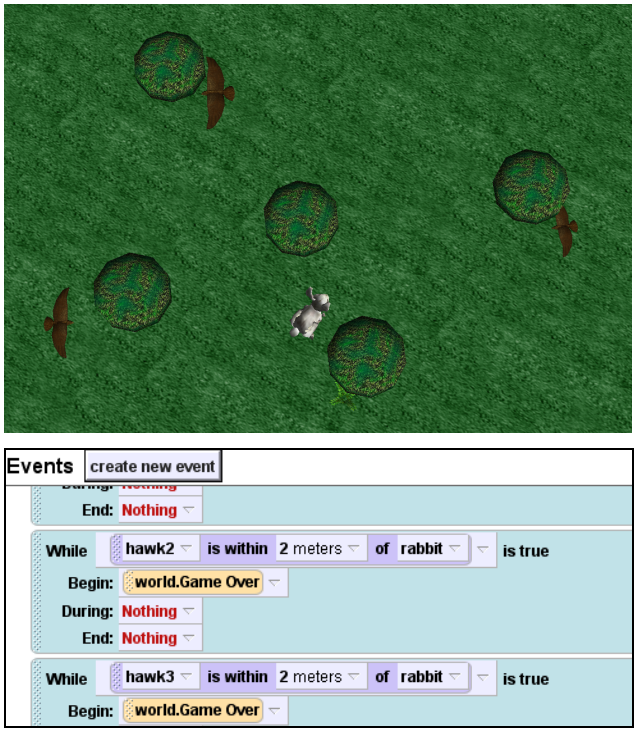

Figure 3. 'Bunny Run' student game with 'while something is true' event.

5. CONCLUSIONS AND FUTURE WORK

Elective technology computer game design courses for middle school students are a promising strategy to introduce CS concepts to a broad population. The findings show that students demonstrated an understanding of a range of computer science concepts that include abstraction and modeling, control structures, and event handlers, while using Alice to make games. The most common constructs were methods, functions, and events. Surprisingly, there were few differences between the use and successful use of constructs, suggesting that if something was in the program, it was generally used correctly.

Based on prior research [13], we expected that making games in Alice would engage students in key features of programming, such as extensive control structures for sequential, conditional, and parallel execution; both global and local variables; abstractions via the use of methods and functions; and event handling for mouse clicks and key input. And indeed, many of the games incorporated elements that represent these complex programming concepts. Almost one-third of the programs used variables, almost one half used functions, but only one of these programs contained a student-created function. Creating functions and using them, as well as creating methods with parameters, are complex concepts that were included in challenges along with the instructions for using other programming constructs. The findings suggest that some of the examples or the specific focus we placed on certain constructs in some of the challenges were not clear to the students, or did not help them program their game ideas. We are in the process of analyzing the game mechanics students chose. Even if not fully operational, those choices will help us learn more about what to include in the curriculum and how to teach this age group

Our results build on prior studies that show middle school students are able to use programming constructs that exhibit complex CS concepts such as abstraction, but struggle with variable initialization, looping, conditionals, pointers, and recursion. This study extends prior research on the use of programming concepts in computer game design courses because we analyzed the creation, use in reachable code, and execution without failure, of programming constructs. Almost all of the prior research focused on syntactic analysis but not on whether the programming constructs were executable, or executable without failure. Future studies that compare students working with different programming environments are needed to determine whether some types of programming constructs are used more frequently because they are the easiest to learn in Alice, because they satisfied conditions for a 'good game' (user input, clear instructions, engaging narrative), or because they were used more frequently in the sample games and challenges.

The simple counts of programming constructs reported in this paper is only a first step in our research on what students are learning while programming a game in Alice. Our broader focus is on the definition and assessment of computational thinking in middle school, in which the use and understanding of programming constructs is one small part [1]. To this end, we are also looking at the students' games for evidence of programming patterns (i.e., combinations of programming constructs) and game mechanics. We have identified a number of non-contiguous sequences of programming constructs that when present in an Alice program indicate a higher abstraction, similar to the work by Koh et al [7]. A few examples of these Alice patterns are counters, timers, and collision detection. We have also identified a number of game mechanics in the students' games such as navigation, collection, and timed challenge. It is possible that the mechanic might inform what students learn for their games and, therefore, determines how computationally "complex" (i.e., constructs/patterns used) the game is. The ideas a student has for their 'dream' game are very motivational for a student's learning. If we can determine what students want to design, then we can identify programming constructs necessary to implement the game mechanics and patterns for those 'dream' games. This process should drive curriculum development.

6. ACKNOWLEDGMENTS

Our thanks go to the teachers and administrators at our seven schools, specifically Anne Guerrero, Shelly Laschkewitsch, Don Jacobs, Sue Seibolt, Karen Snedeker, Susan Rivas, and Katie Ziparo. Thanks also to teaching assistants, Will Park and Joanne Sanchez; and to Eloy Ortiz and Pat Rex, whose vital support was necessary to run this program. Thanks to all of the students who participated. Finally, thanks to our undergraduate researchers Dominic Arcamone, Melanie Dickinson, Steven Butkus, Anthony Lim, and Kimberly Shannon who analyzed the students' games.

This research is partially funded by a grant from NSF 0909733 "The Development of Computational Thinking among Middle School Students Creating Computer Games." Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Barr, V. and Stephenson, C., 2011. Bringing computational thinking to K-12: what is Involved and what is the role of the computer science education community? ACM Inroads 2(1), 48-54.

[2] Baytak, A. & Land, S.M., 2011. Advancing elementary-school girls' programming through game design. International Journal of Gender, Science, and Technology, 3(1).

[3] Carbonero, M., Szafron, D., Cutumisu, M., & Schaeffer, J., 2010. Computer-game construction A gender-neutral attractor to computing science. Computesr & Education, 55, 1098-1111.

[4] Cooper, S., Pérez, L.C.and Rainey, D., 2010. K--12 computational learning. Communications of the ACM, 53(11), 27-29.

[5] Denner, J., Werner, L. and Ortiz, E., 2012. Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? Computers & Education, 58(1), 240-249.

[6] Kelleher, C., Pausch, R. and Kiesler, S., 2007. Storytelling Alice motivates middle school girls to learn computer programming. In Proceedings of the SIGCHI conference on Human factors in computing systems. ACM, New York, NY, USA, 1455-1464.

[7] Koh, K.H., Basawapatna, A., Bennett, V., Repenning, A., 2010. Towards the automatic recognition of computational thinking for adaptive visual language learning. IEEE Symposium on Visual Languages and Human Centric Computing, pp. 59-66.

[8] Lee, L., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L., 2011. Computational thinking for youth in practice. ACM Inroads 2(1), 32-37.

[9] Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M. and Rusk, N., 2008. Programming by choice: urban youth learning programming with Scratch. In Proceedings of the 39th SIGCSE technical symposium on Computer science education. ACM, New York, NY, USA, 367-371.

[10] Mayer, R.E., 1988. Teaching and learning computer programming: Multiple research perspectives. Psychology Press.

[11] Robins, A., Rountree, J., & Rountree, N., 2003. Learning and teaching programming: A review. Computer Science Education, 13(2), 137-172.

[12] Rodger, S., Hayes, J., Lezin, G., Qin, H., Nelson, D., Tucker, R., Lopez, M., Cooper, S., Dann, W. and Slater, D., 2009. Engaging middle school teachers and students with Alice in a diverse set of subjects. In Proceedings of the 40th ACM technical symposium on Computer science education. ACM, New York, NY, USA, 271-275.

[13] Soloway, E. and Spohrer, J., 1989. Studying the Novice Programmer. Mahwah, NJ: Erlbaum.

[14] Stolee, K. and Fristoe, T., 2011. Expressing computer science concepts through Kodu game lab. In Proceedings of the 42nd ACM technical symposium on Computer science education. ACM, New York, NY, USA, 99-104.

[15] Vos, N., van der Meijden, H., & Denessen, E., 2011. Effects of constructing versus playing an educational game on student motivation and deep learning strategy use. Computers & Education, 56(1), 127-137.

[16] Wang, T., Mei, W., Lin, S., Chiu, S., & Lin, J., 2009. Teaching programming concepts to high school students with Alice. 39th ASEE/IEEE Frontiers in Education Conference, San Antonio, TX.

[17] Werner, L., Denner, J., Bliesner, M.and Rex, P., 2009. Can middle-schoolers use Storytelling Alice to make games?: results of a pilot study. In Proceedings of the 4th International Conference on Foundations of Digital Games. ACM, New York, NY, USA, 207-214.