Accelerating K-12 Computational Thinking Using Scaffolding, Staging, and Abstraction

David S. Touretzky
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
dst@cs.cmu.edu

Daniela Marghitu
Computer Science and Software Engineering
Auburn University
Auburn, AL 36849
marghda@auburn.edu

Stephanie Ludi
Dept. of Software Engineering
Rochester Institute of Technology
Rochester, NY 14623
salvse@rit.edu

Debra Bernstein
TERC
Cambridge, MA 02140
debra_bernstein@terc.edu

Lijun Ni
Georgia Inst. of Technology
Atlanta, GA 30332-0760
lijun@cc.gatech.edu

ABSTRACT

We describe a three-stage model of computing instruction beginning with a simple, highly scaffolded programming environment (Kodu) and progressing to more challenging frameworks (Alice and Lego NXT-G). In moving between frameworks, students explore the similarities and differences in how concepts such as variables, conditionals, and looping are realized. This can potentially lead to a deeper understanding of programming, bringing students closer to true computational thinking. Some novel strategies for teaching with Kodu are outlined. Finally, we briefly report on our methodology and select preliminary results from a pilot study using this curriculum with students ages 10–17, including several with disabilities.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Eduation**]: Curriculum

General Terms

Design, Experimentation

Keywords

Kodu, Alice, robotics, computational thinking

1. INTRODUCTION

Highly scaffolded programming environments offer novices a smoother path to early success in computing [7], but their limited expressiveness must inevitably lead to their abandonment in favor of more powerful conventional languages.

SIGCSE'13, March 6–9, 2013, Denver, Colorado, USA.

We have developed a three-stage model of programming instruction for K-12 students that embraces these transitions between frameworks as part of the learning process. The model was built upon a three-step ladder model developed by Daniela Marghitu since 2005, in which students with varying abilities and experience levels could learn any of three software frameworks and progress at their own pace from one to another [12]. Our new curriculum specifies the ordering and guides students to make explicit analogies between frameworks to develop a deeper, more abstract understanding of fundamental computing concepts.

We conducted a pilot study using Microsoft Kodu [15], Alice [4], and Lego Mindstorms NXT-G [8] as the three stages of a five-day computer camp for 31 children ages 10-17. In this paper we describe our strategy for teaching programming using Kodu, including several innovative ideas that were developed in the course of the project. We also discuss some CS Unplugged-type enrichment activities [2] employed in the camp. We then describe how students transitioned from Kodu to Alice and NXT-G, and how we mapped concepts from one framework to another.

A key design goal for the curriculum was that it be accessible to a wide range of students, including those with disabilities. Our initial offering was not completely universal (e.g., Alice and Kodu cannot be used by students who are blind), but we were able to include students who were partially sighted or had other disabilities, as described in Section 6. We provided assistive technology and adjusted activities to ensure that students with disabilities had equal access to the tools, activities, and interactions that made up the curriculum [13]. Examples included large-print handouts and assessments, legible, high contrast Kodu and Alice worlds, and extra participant support for students who required more assistance. With these aids, the curriculum described here was used by all of the students in the camp.

2. DEEP & ABSTRACT UNDERSTANDING

If the goal of K-12 computing education is to teach students "computational thinking" [20, 21], how should that be facilitated? We believe mastery of computational thinking implies a deep and abstract understanding of the fundamental concepts of computing:

- **Deep:** the ability to recognize fundamental concepts instantiated in progamming contexts. For example, recognzing that a certain task inherently involves conditional execution, or looping, or parallelism, or persistent state.
- **Abstract:** separating the essence of a mechanism from mere syntactic details. For example, appreciating that WHEN/DO in Kodu, If/Then in Alice, and SWITCH blocks in NXT-G all function as *conditional expressions*, even though they look different.

We can promote this style of thinking by guiding students to draw analogies between different formalisms [3, 5]. We transition them betwen these formalisms in a carefully structured way. The increased complexity of instruction is mitigated by this scaffolding.

3. KODU INSTRUCTION STRATEGIES

Microsoft Kodu [15] is an icon-based programming framework designed to allow young children to construct their own computer games. Kodu employs many clever scaffolding mechanisms to keep novice programmers on track [10]. For example, students compose their programs, organized as sequences of WHEN/DO rules, by selecting icons from a context-sensitive menu, making syntax errors impossible. Rather than using named variables, Kodu uses *scores*, such as the "red score" and the "blue score," which children intuitively understand from playing computer games. (Students can actually program in Kodu using an Xbox game controller instead of a keyboard.) Assigning a value to a score automatically causes it to be displayed on the screen, eliminating the need for explicit output statements.

In the course of developing our curriculum, we had some insights into how to use Kodu more effectively to teach programming. We summarize these ideas here.

1. Start with programs in a pre-made world. Some Kodu tutorials begin by teaching the terrain editor and having students construct their own virtual world. But this activity takes significant time and can divert students from the central task of learning programming. We found it preferable to supply students with small worlds with prepositioned characters and objects. They did, however, work with the terrain editor during free exploration periods, and this contributed to their overall enjoyment of Kodu and provided an additional opportunity to practice new programming skills.

2. Show programs in textual form. Kodu programs are sequences of WHEN/DO rules. A rule is a string of tiles, each bearing a word and an icon, as in Figure 1.

Figure 1: A Kodu rule.

The first example code we show to students uses this iconic notation, but we immediately introduce equivalent textual notation, and quickly abandon the iconic version. The textual version is more compact, allowing more to fit on a page. But more importantly, it introduces students to the notion that the same idea can be expressed in multiple forms: a first step toward abstraction. They see sample code on the page in textual form, but when they enter it into the computer they see it in its iconic form.

3. Use parenthesized keywords to guide menu navigation. While Kodu's context-sensitive menus are helpful, the menu structure is deep enough that students may not know where to find a tile. For example, "move" is a top-level menu item, but "glow" is found in the "actions" submenu, while "vanish" is found in the "combat" submenu. The iconic representation of a tile does not indicate how to find it in the menu tree. We addressed this by including the submenus as parenthesized keywords in the textual notation, so the rule in Figure 1 would be written as:

`WHEN see apple close DO` *(actions)* `glow` *(colors)* `red`

4. Use arrows to help students interpret indentation. A Kodu rule can be linked to a preceding rule by indenting it. The linked rule will be considered for execution only if the parent rule's WHEN clause is satisfied. If the linked rule has a non-empty WHEN clause, this mechanism creates a conjunctive test. If the parent rule has a non-empty DO clause, this mechanism creates a compound action. But as Jill Denner of ETR Associates has observed (personal communication), indentation has no inherent significance to young readers and does not imply any sort of attachment relation. Indentation has also been found to be problematic for beginning Python programmers [9]. This is an instance where Kodu's graphical notation fails to scaffold student understanding.

This deficiency could easily be remedied by tweaking Kodu's graphical user interface. In our textual version, we solved the problem by introducing an arrow icon that links an indented rule to its parent, as shown in Figure 2.

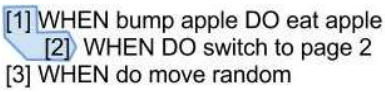

Figure 2: Graphic aid for illustrating how indentation links a rule to its parent rule above.

5. Teach state machines; explicitly identify states with pages. Kodu programs are organized as pages of rules, with all the rules on a page applying simultaneously. Thus, the most general way for a character to change its behavior is to switch to another page. We introduce students to the concept of state machines as a way of reasoning about program behavior, with each page corresponding to a state node [17]. We support this with a CS Unplugged-type activity described in Section 4. State machines are particularly helpful for teaching students about loops.

The Kodu editor displays the page number in a unique color: blue for page 1, green for page 2. etc. We use these same colors for nodes in our state machine diagrams, reinforcing the correspondence between nodes and pages. Instructions to switch pages form the links between nodes.

6. Have the kodu change color to indicate what page it's on. When creating complex behaviors that loop between pages (states), it's helpful to know a character's current page in order to understand its behavior. We established a convention that the first rule on every page would set the kodu's color to the page's color. We used this conven-

tion to frame questions about program state such as "Why did the kodu turn red when it bumped the tree?"

7. Distinguish implicit from explicit looping. The Kodu interpreter repeatedly executes all applicable rules on the current page. This can result in *implicit looping* behavior with no use of control structure primitives. For example, the following program loops forever:

```
[1] WHEN DO score red 1 point
```

Kodu's movement actions, such as *move toward,* are incremental; the programmer relies on implicit looping to repeat the actions until a goal is met. For example, to move the kodu toward a tree and then go do something else when the tree is reached, one would write an implicit loop with a termination test that exits the page:

```
[1] WHEN see tree DO move toward
[2] WHEN bump tree DO switch to page 2
```

Implicit looping cannot be used when the loop body contains actions requiring complex sequencing or nested iteration. In this case one must use what we call *explicit looping*, where the program explicitly switches back and forth between the loop's control page and its body pages, which may contain their own implicit or explicit loops.

We introduced the concept of implicit looping to make *explicit* to students the way that Kodu interprets their programs. Students also learn to override implicit looping using the *once* tile, e.g., the following program does not loop:

```
[1] WHEN DO score red 1 point once
```

Drawing a distinction between implicit and explicit looping helps students develop a more abstract understanding of the loop construct by seeing it implemented in two different forms.

8. Provide graphical notation to highlight implicit and explicit looping. Explicit looping is easy to identify in a state machine diagram because the relevant state nodes form a loop in the graph [17]. We go further and label state nodes with their function in the loop: initialization, loop logic (including termination test), loop body, and wrap-up. The latter refers to actions taken after exiting the loop. Not every loop has separate pages for all four of these functions. But students are taught that explicit loops by definition involve page switching, and hence, loops in the state machine diagram.

Because implicit loops confine the body and loop logic to a single page, we introduced an extension to the standard state machine notation to distinguish those pages: a semicircular arrow appearing inside a state node, as in Figure 3.

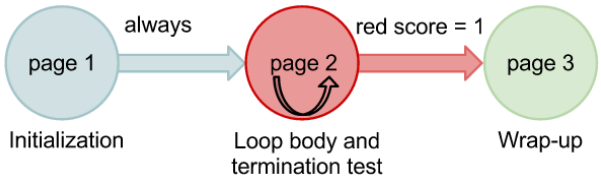

Figure 3: State machine diagram with added arrow indicating implicit looping within a page (state).

There are several conditions under which a page does not implicitly loop. For example, if every rule contains a *once* tile, or if execution of the page always includes a *switch to page* action. In one exercise, students were asked to go through a complex Kodu program and mark on their state machine diagram the pages that do contain implicit loops.

9. Teach bugs explicitly by giving them names and showing sample code. A *program structure* bug could result from incorrect indentation, which changes the dependency relationship between rules. An *order of operations* bug could result from swapping two rules, because in the event of conflicting actions, Kodu gives priority to the lower numbered rule. *Loop initialization* and *loop termination* bugs are straightforward. In the Alice component we introduced the *fencepost error* bug and the *divide by zero* bug.

A series of pins with cartoon bugs illustrated these six bug types. Students earned bug pins by successfully completing bug-related exercises, such as deliberately introducing a bug of the specified type, demonstrating its effects to the instructor, then fixing it.

4. CS UNPLUGGED-TYPE ACTIVITIES

Computer Science Unplugged is a series of activities developed by Bell et al. in which students use their bodies or simple props to simulate algorithms illustrating important computer science concepts [2]. Hence, no computers are involved. This type of kinesthetic learning is fun for students across a broad age range, and has proven effective for reaching students with disabilities [14]. We used three such activities in our summer camp. Two were of our own design; the third came directly from the CS Unplugged book [2].

In the first activity, students hand-simulated the execution of a Kodu program containing two characters: a kodu and an exploding mine. We created a large poster of the state machine for the program that was placed on the floor, and used chess pieces (a white pawn and a black pawn) to track the states of the two characters. We also laid out a map of the Kodu world on the floor, and placed hand-made kodu and mine characters on the map, moving them as the simulation progressed. The kodu character included a rotating mechanism that could be used to change its color. The mine could glow various colors, and we used little flags on stick-pins to simulate that. Students shared the tasks of deciding which rules were able to execute on the current cycle, moving the kodu's pawn when a state change occurred, and moving the kodu and the mine characters around on the map. The exercise reinforced students' understanding of the Kodu rule execution mechanism, including such fairly advanced topics as implicit looping and priority arbitration.

In the second activity, students simulated a bubble sort as an introduction to nested loops and a hint at arrays. (There are no arrays in Kodu, and we did not cover Alice arrays due to time limitations.) Ten students formed a line and each held a card containing a number. Another group acted as sorters. With square dance music playing and the audience encouraged to clap along, the instructor acted as "caller". On each pass, a sorter would start at the left edge of the line, examine the first pair of numbers, and give one of two instructions: "bow to your partner" if the numbers were in the correct order, or "switch with your partner" if the two should switch places. The sorter then advanced to the next pair. As students became proficient at sorting, we sped up the pace, which increased the entertainment value.

The final activity, which came during the Alice portion of the camp, was the network sorting activity from the CS Unplugged book. We laid out a six-element sorting network on the floor, and students moved synchronously through the stages of the network, arriving at the output stage in sorted order. This allowed us to make some interesting comparisons between the two sorting algorithms, e.g., the sorting network requires fewer comparisons than a bubble sort would use.

5. CONCEPTUAL MAPPING

The first two days of the camp were devoted to Kodu, the next two days to Alice (occasionally looking back at Kodu for comparison), and the final day to NXT-G. In this section we discuss the conceptual mapping between Kodu, Alice, and NXT-G that we guided students to develop through direct instruction, group discussion, and exercises.

5.1 Objects

Both Kodu and Alice have named objects ("characters" in Kodu), but Alice objects can have nested components, such as body parts. Also, Alice objects are organized in a tree for convenient inspection, while Kodu objects must be accessed at their location in the world. Alice objects have named user-defined methods that take optional parameters, while Kodu only provides numbered pages. Kodu's simplicity is a deliberate attempt to protect beginners from being overwhelmed by too many details. As students become more comfortable with programming they can learn new constructs such as parameterized methods.

5.2 Variables

Kodu uses predefined *scores* as variables, while Alice has conventional named and type variables. This is one of the areas where trading simplicity for power is evident. Using variables in Kodu is effortless. In Alice it takes some work, since one must first explicitly create the variable and assign it a name and initial value, and take additional steps to display its value. But the benefit of descriptive names soon becomes evident as programs with multiple variables are introduced. VARIABLE blocks in NXT-G are similar to Alice's named and typed variables.

5.3 Conditionals

Since the only Kodu statement is the WHEN/DO rule, every line of a Kodu program is a conditional. Alice also has WHEN/DO rules, mainly used to respond to events such as mouse clicks on objects or single-character keyboard input. We start students out using these events very early, and the analogy with Kodu is immediately apparent. Alice WHEN/DO rules can also include arbitrary expressions as conditions, but since they run asynchronously and in a global context (no access to local variables), they cannot replace the conventional If/Then, which Alice also provides.

We introduce If/Then statements later in the curriculum, after methods, functions, predicates, and parallelism have all been covered. We also introduce If/Then/Else, and explore with the students how this logic can be implemented in Kodu using a pair of WHEN/DO rules with opposite conditions. In NXT-G, the SWITCH block implements If/Then/Else.

5.4 Parallelism

Kodu uses a unique form of parallel execution. All the rules on a page evaluate their conditions simultaneously. Then actions take effect sequentially, and in the event of conflicting actions, the lower numbered rule has priority. So, for example, if one applicable rule moves the kodu north but a later rule that also has a true condition moves it south, the kodu will move north. This can be useful for creating default behaviors whose action will be taken only if no earlier rule is eligible to fire, e.g.:

```
[1] WHEN see apple DO move towards
[2] WHEN DO move random
```

Because all rule evaluations are completed first in Kodu instead of being interleaved with the actions, a rule's action cannot affect the ability of later rules to fire on the same cycle. This can lead to unexpected behavior if the programmer has an incorrect model of the execution cycle. (One of the authors actually experienced this bug.) For example, given that scores are initialized to zero, the following code fragment sets the red score to 5 but also switches to page 2 even though the second rule appears to guard against that:

```
[1] WHEN DO score red 5 points
[2] WHEN scored red 0 points DO switch to page 2
```

In Alice, sequential execution is the norm; parallelism must be explicitly requested via a Do Together block, and there is no conflict resolution mechanism. We examine this difference with the students, comparing Kodu code with Alice code for the same task. NXT-G also provides for explicit parallelism via a "sequence beam", but we did not cover this due to lack of time.

5.5 Looping

Looping in Kodu is both simpler and more difficult than in Alice. Implicit loops occur automatically in Kodu unless blocked by *once* tiles or a page switch. But explicit looping requires at least two page switches, which are the Kodu equivalent of the dreaded "go to". There is also no notion of an associated index variable, so variable updating and testing must be done explicitly. In contrast, Alice's "Loop n times" statement handles all those details invisibly, but also allows the index variable to be exposed if desired. Alice also provides a conventional While loop.

When teaching explicit looping in Kodu we identify four loop components: initialization, loop logic (including termination test), body, and wrap-up. Each of these goes on a separate page, although it is sometimes possible to fit two components on the same page if the required actions are simple enough. When teaching Alice's Loop construct, we expose the loop details and point out how the first three loop components, which make up the loop proper, are realized by this single Loop statement. We also show students Kodu code to implement a specific Alice loop they've studied, to reinforce the mapping between the two formalisms.

NXT-G offers a LOOP block with a variety of options that can cause it to function like either a For or a While loop. As in Alice (but not Kodu), the body of the loop is physically nested inside the block.

5.6 States

We've identified states with pages in Kodu, glossing over the parallelism and implicit looping within. The sequential execution model in Alice suggests that each statement could be regarded as a state, but since Alice statements can nest,

the mapping is not quite that straightforward. Due to lack of time, we did not introduce a new graphical notation in the Alice segment, but were we to do so, we would use flowchart notation. The major difference with state machines is that in a flowchart, the boolean expression component of a conditional or loop is depicted as a separate node.

NXT-G has semantics similar to Alice (sequential execution by default, and nesting components), but its graphical layout more closely resembles a state machine. However, NXT-G uses "wires" between "ports" to transmit parameter values between blocks. These extra links distinguish the graph from a pure state machine graph.

6. THE PILOT STUDY

We conducted a five day CS4All summer camp at Auburn University in July 2012. Students were recruited from schools in Alabama and Georgia, and ranged in age from 10 to 17, with a mean age of 13.1. A total of 31 students participated; 13 were female. Seven of the students self-identified as having one or more of the following disabilities: visual impairment (VI), Asperger's syndrome (Asp), cerebral palsy (CP), or dyslexia. The distribution of ages and genders was: 10 (M, Asp), 12 (F, VI), 13 (M, VI), 14 (M, dyslexia), 15 (M, Asp), 16 (F, CP with VI), 17 (F, VI).

Students each had their own computer, but were organized in groups of 3-5 overseen by one or two instructors. Instructors were mainly CS graduate students, plus two CS/ECE undergraduates, several of the authors, and two undergraduate instructional aides from the Auburn University Special Education and Rehabilitation Department. Students worked for seven hours each day, including time for unplugged activities and free exploration.

7. EVALUATION

We conducted an evaluation of the pilot summer camp, focusing on participant outcomes in two primary areas: knowledge development and attitudes. Knowledge development was measured using two short surveys and one programming task. The first Kodu assessment focused on participants' understanding of state machines and the relationship between program state and program output. In the second Kodu assessment, we asked students to program loops. Our analysis of their programs focuses on inclusion of critical loop components (e.g., loop initialization, loop body, and termination test). The Alice assessment focused primarily on students' understanding of relationships within Alice, understanding of core concepts, and ability to draw comparisons between Kodu and Alice. Participant attitudes were assessed using an attitude survey developed by Heersink and Moskal [6]. The attitude survey was administered on days 1 (pre) and 5 (post) of the camp. Participants' overall reactions to the pilot curriculum were collected via written survey at the end of the camp.

This paper presents preliminary findings from the post-camp survey. Analysis of the knowledge surveys, programming task, and attitude assessments are underway.

Participant reactions to camp activities were collected via an 18-question written survey administered on the last day of the camp. This survey included closed- and open-ended questions about participants' enjoyment of camp activities, their intentions to continue using the programming environments introduced during the camp, and whether they had taken steps to make those environments available at home.

Participants' reactions to the summer camp were generally positive, with the majority reporting that they enjoyed the camp, learned a lot, and would recommend it to their friends. See Table 1. (Participants responded to a 5-point scale. However, the end points have been collapsed for ease of presentation.)

Table 1: Participants' reactions.

	n	Disagree; Strongly Disagree	Neut.	Agree; Strongly Agree
I enjoyed participating in this camp.	28	14%	7%	79%
I would recomend this camp to my friends.	29	10%	34%	55%
I learned a lot about computer programming at this camp.	29	17%	10%	72%

When asked whether they would continue to use the programming environments introduced during the camp, nearly half of the participants said they would "probably" or "absolutely" use Kodu, with an additional 25% saying they "might" do so. Fewer reported that they would use Alice after the camp ended. The majority of participants (71%) said they would "probably" or "absolutely" want to continue programming robots if they had access to a robotics kit like Lego Mindstorms (see Table 2).

Table 2: Participants' intent to continue.

Response	Kodu	Alice	Robotics
I absolutely will not	11%	25%	0%
I probably will not	18%	14%	11%
I might	25%	29%	18%
I probably will	21%	25%	25%
I absolutely will	25%	7%	46%

More than half of the participants planned to install Kodu (61%) or Alice (50%) on computers at home. 36% reported that they currently had access to a Lego NXT kit either at home or at school, and an additional 29% reported that they planned to get one.

8. LESSONS LEARNED

The 2012 summer camp was an experiment to test the idea that explicitly transitioning students from more scaffolded to less scaffolded frameworks was feasible and beneficial. An extensive amount of data was collected and is presently being analyzed to determine what students actually learned. However, we can make a few preliminary observations here.

Beginning students sometimes find the Alice interface intimidating due to the multiple panes and tabs. There is a fair amount of typing (e.g., method names, variable names, and numeric values), vs. no typing at all in Kodu except to make a character "speak". And Alice's drag-and-drop editing interface is not entirely intuitive. Based on our experience in previous camps, we expected students to take to

Kodu quickly because of its simpler interface. As predicted, once students mastered the ideas of programming characters in a simulated world and switching between editing and execution modes, the transition to Alice went smoothly.

NXT-G has semantics sufficiently so close to Alice, and its graphical interface so closely resembles a state machine diagram, that students took to it immediately. Working with physical robots is exciting for students [1, 16], and their enthusiasm was reflected in the survey responses.

But NXT-G is not the ideal choice for the third stage of the framework precisely because it adds little beyond what students have encountered in Alice. Introducing students to robot programming is still a good idea, but at least with older students, it could be better done using more sophisticated robots and a more powerful programming formalism. During the camp we gave students a chance to teleoperate a Calliope5KP robot developed at Carnegie Mellon [19] that features a five degree-of-freedom arm with gripper and a Kinect camera on a pan/tilt mount. Students were fascinated by the robot and eager to learn more about it.

9. FUTURE WORK

Our pilot curriculum was compressed and covered a limited set of topics due to time constraints. To apply our ideas in a year-long or multi-year computer science course, and ensure that students fully master the concepts, will require considerable additional work. As we move forward we will be assessing the frameworks used in the pilot study. Scratch [11] might be an attractive alternative to Alice.

We would also like to develop an alternative robotics curriculum for high school students using the Tekkotsu software framework [18] and robots similar to the Calliope series. Tekkotsu is based on C++ and includes modules for computer vision, landmark-based navigation, and manipulation. This would make the third stage of our model a significant advance toward college-level programming. For younger students, NXT-G's limitations aside, it at least allows them to see familiar programming constructs in a novel form, facilitating a more abstract understanding.

10. ACKNOWLEDGMENTS

This work was funded by National Science Foundation awards CNS-1151542, CNS-1151713, CNS-1151980, and CNS-1152003. We thank our camp instructors T. Ben Brahim, J., Weaver, Y. Rawajfih, E. Banu, S. Kulkarni, C. Stephens, Y. Tiang, C. Cira, A. Jain, A. Marshall, and G. Leung.

11. REFERENCES

[1] Barker, B. S., and Ansorge, J. (2007) Robotics as a means to increase achievement scores in an informal learning environment. Journal of Research on Technology in Education, 39(3):229–243.

[2] Bell, T., Witten, I. H., and Fellows, M. (2010) Computer Sciene Unplugged. Available at <http://csunplugged.org/books>.

[3] Bransford, J.D., Brown, A. L., and Cocking, R.R. eds. (2000) How People Learn: Brain, Mind, Experience, and School. Washington, D.C.: National Academy Press

[4] Dann, W. P., Cooper, S., and Pausch, R. (2008) Learning to Program with Alice (2nd Edition). Prentice-Hall.

[5] Gentner, D., Loewenstein, J., and Thompson, L. (2003) Learning and transfer: A general role for analogical encoding. Journal of Educational Psychology, 95(2):393–408.

[6] Heersink, D., and Moskal, B. M. (2010) Measuring high school students' attitudes toward computing. *Proc. SIGCSE'10*, 446–450.

[7] Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys, 37(2), 83–137.

[8] Kelly, J. (2010) LEGO Mindstorms NXT-G Programming Guide. New York, NY: APress.

[9] Konidari, E. and Louridas, P. (2010) When students are not programmers. ACM Inroads 1(1):55–60.

[10] McLaurin, M. (2011) The design of Kodu: A tiny visual programming language for children on the Xbox 360. Proceedings of POPL-11.

[11] Maloney, J., Resnick, M., Rusk, N., Silverman, B., Eastmond, E. (2010) The Scratch programming language and environment. ACM Transactions on Computing Education, November 2010.

[12] Marghitu, D., Ben Brahim, T., Rawajfih, Y., Weaver, J., and Shanahan, J. (2010) Auburn University K-12 inclusive STEM outreach programs. SDPS 2010 Conference, June 6–9, 2010, Dallas, TX USA.

[13] Marghitu, D. (2012) Auburn University K-12 Inclusive STEM Outreach Programs. SIGCSE 2012 workshop slides available at <https://fp.auburn.edu/comp2000/SIGCSE2012_AccessComputingWorkshop.htm>.

[14] Marghitu, D., Bell, T., et al. (2009) Using virtual worlds to engage typical and special needs students in kinesthetic computer science activities: A Computer Science Unplugged case study. AACE EdMedia 2009 Conference, June 22–26, 2009, Honolulu, HI, USA.

[15] Microsoft Kodu Game Lab. Available at <http://www.kodugamelab.com/About>.

[16] Nugent, G., Barker, B., Grandgenett, N., and Adamchuk, V. I. (2010) Impact of robotics and geospatial technology interventions on youth STEM learning and attitudes. Journal of Research on Technology in Education, 42(4), 391–408.

[17] Stolee, K. T., and Fristoe, T. (2011) Expressing computer science concepts through Kodu Game Lab. *Proc. SIGCSE'11*, pp. 99–104.

[18] Touretzky, D. S. (2010) Preparing computer science students for the robotics revolution. Communications of the ACM, 53(8):27–29.

[19] Touretzky, D. S., Watson, O., Allen, C. S., and Russell, R. J. (2010) Calliope: Mobile manipulation from commodity components. In A. Thomaz and M. D. Anderson (Eds.), Papers from the 2010 AAAI Robot Workshop. Technical report WS-10-09. Menlo Park, CA: AAAI Press.

[20] Wilson, et al. (2010). Running On Empty: The Failure to Teach K–12 Computer Science in the Digital Age. Association for Computing Machinery, 2010. Available at <http://www.acm.org/runningonempty/>.

[21] Wing, J. M. (2006) Computational thinking. Communications of the ACM 49(3):33–35.