Expressing Computer Science Concepts Through Kodu Game Lab

Kathryn T. Stolee*
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE, U.S.A.
kstolee@cse.unl.edu

Teale Fristoe*
Expressive Intelligence Studio
University of California, Santa Cruz
Santa Cruz, CA, U.S.A.
teale@soe.ucsc.edu

ABSTRACT

Educational programming environments such as Microsoft Research's Kodu Game Lab are often used to introduce novices to computer science concepts and programming. Unlike many other educational languages that rely on scripting and Java-like syntax, the Kodu language is entirely event-driven and programming takes the form of 'when – do' clauses. Despite this simplistic programing model, many computer science concepts can be expressed using Kodu. We identify and measure the frequency of these concepts in 346 Kodu programs created by users, and find that most programs exhibit sophistication through the use of complex control flow and boolean logic. Through Kodu's non-traditional language, we show that users express and explore fundamental computer science concepts.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Experimentation

Keywords

Educational programming languages, Kodu

1. INTRODUCTION

Since the development of high level programming languages, researchers and educators have embraced domain specific languages designed to introduce children to programming from a young age [10]. While ease of use has always been a concern with these languages, often attractiveness to children is a top priority, with the rationale that students should enjoy their experiences to gain the most from them. For this reason, these languages often target media creation, such as art, animations, or games. However, rarely are the actual educational benefits of the languages the focus of scrutiny, resulting in a stark lack of understanding of the skills these languages provide for their users.

Microsoft Research's Kodu Game Lab [4] is a recent addition to this tradition of educational programming environments. Since its release in the summer of 2009, it has been installed over 50,000 times in over 120 countries. Kodu allows users to create their own video games by designing the world, deciding which characters will appear in it, and programming the characters using an easy-to-understand visual programming language. It takes the unique perspective that children should be both producers and consumers of games. Like many similar programming environments, Kodu has been shown to be enjoyable for users, and clubs have formed worldwide to teach Kodu (e.g., [1, 2, 5]), but its educational benefits have not been explored in any rigorous fashion.

In this work, we begin to identify which computer science concepts users can express through Kodu, and then explore which concepts actual users have utilized in Kodu. We aim to address the following research questions:

RQ1: Which computer science concepts can be expressed through the Kodu Language?

RQ2: How much time do end users spend designing and programming during a Kodu session?

RQ3: How often does each computer science concept appear in the programs created by the Kodu community?

To this end, we look at several language features available in Kodu, connecting them to fundamental computer science concepts. We empirically investigate what users of Kodu are actually learning, first by determining what percentage of their time is spent programming and configuring their program as opposed to playing or exploring, and then by performing a frequency analysis on 346 end user programs, measuring how often each computer science concept appears.

The rest of this paper is structured as follows. Section 2 presents related work in educational programming languages and Section 3 gives some background on Kodu Game Lab and the Kodu language. Section 4 presents the computer science concepts that can be learned in Kodu (*RQ1*), Section 5 shows how users spend their time during a Kodu session (*RQ2*), and Section 6 addresses *RQ3* by exploring how often each concept is present in the games created by the community. Section 7 presents the threats to validity, and Section 8 presents future work and conclusions.

*Work completed while at Microsoft FUSE Labs.

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

Rule	→	Condition Action
Condition	→	Sensor FilterSet
Action	→	Actuator ModifierSet Selector
FilterSet	→	Filter FilterSet | Filter
ModifierSet	→	Modifier ModifierSet | Modifier
Sensor	→	see | bump
Filter	→	apple | red | ϵ
Actuator	→	move | eat
Modifier	→	toward | away | ϵ
Selector	→	it | me | ϵ

Figure 1: Grammar Sketch for a Rule

2. RELATED WORK

Released in 1967, Logo [10] pioneered the development of educational programming environments, and has been very influential on the development of subsequent languages in the domain [3, 6, 13, 15]. While there have been countless educational programming environments since Logo, Alice [6], Greenfoot [3], and Scratch [13] are commonly explored by the research community and have many similarities to Kodu. Like Kodu, all are visual programming languages that allow users to create their own animations, games, and simulations. Scratch also has extensive community features that facilitate sharing of programs and collaboration [13]. Unlike Kodu, the languages of Alice, Greenfoot, and Scratch follow the Java-like syntax of many mainstream programming languages, whereas the Kodu language is entirely event-driven (see Section 3.1 for language details).

In terms of evaluation, educational programming environments have been measured primarily using qualitative and anecdotal evidence [15, 17]. A more recent trend has been toward the inclusion of quantitative data in the evaluation, though sample sizes generally remain small. Kelleher has analyzed user habits by tracking time spent programming versus other activities, such as designing in Alice [9]; we perform a similar analysis with Kodu. Pre- and post-tests have also been used to evaluate what users learn by using Alice and Scratch, and are generally accompanied by qualitative support [11, 12, 14]. Researchers are only beginning to examine artifacts written in Alice and Scratch to determine what the authors of the systems may be learning, a technique we continue in this work [7, 11, 16, 19].

3. ABOUT KODU

At its core, Kodu is an environment that allows users to create and play their own video games. It is available for download for the Xbox and also on the PC. Developed by Microsoft Research, the vision of the tool is to help users learn computer science concepts through game creation.

3.1 Kodu Language

Kodu is a high-level, visual, and interpreted language that can be represented as a context-free grammar [18]. Users can program each character (e.g., a fish, cycle, apple, tree) individually, and the programming defines how it interacts with the world, much like an intelligent agent. The programming takes place on pages, which define different states for the character. A character may contain up to 12 pages of programming, and can maintain its state and control flow by switching between pages. Each page contains a set of rules, where a rule is analogous to a statement in typical programming languages. Each rule is in the form of a condition and an action, which form a *when – do* clause, that is, *when condition, do action*. If the condition is satisfied, then the action is performed. All the rules on a page are evaluated in a single frame, from top to bottom. There are over 500 tiles that can be used to compose rules in the Kodu Language. In Figure 1, we give a grammar sketch for a rule in Kodu, and limit the alphabet to 10 tiles. Each capitalized word (e.g., *Rule, Condition*) represents a non-terminal symbol and appears on the LHS of a production rule. Each lowercased word (e.g., *see, apple*) represents a terminal symbol that maps to one tile in the Kodu language.

Figure 2: Programming Interface in Kodu

3.2 Programming in Kodu

Programming in Kodu involves the placement of programming tiles in a meaningful order to form the condition and action on each rule. We show an example of the programming user interface in Figure 2, where the program shown was generated using the grammar shown in Figure 1. In the example, there are two rules, one per line. The condition of each rule is on the left, and the action on the right, as is defined by the grammar rule, *Rule → Condition Action*.

For the first rule, the condition is, when *see red apple*, and the action is, *move toward*. In the condition, *see* is the sensor and *red* and *apple* are filters. The action defines the behavior of this particular character when it sees a red apple, that is, it *moves toward*. Here, *move* is the actuator, *toward* is the modifier, and *it* is the default selector, since the condition identifies a target object (i.e., the red apple). The second rule has a similar condition with a different sensor, *bump*. The action, *eat*, indicates that the character should eat any red apple it bumps. The programming in Figure 2 applies to the first page in this character's programming, as indicated by the number "1" at the top of the screen.

4. COMPUTER SCIENCE CONCEPTS IN KODU

While the Kodu language is unlike most mainstream programming languages, it allows its users to explore many fundamental concepts in computer science. In this section, we explore each of these concepts and how they can be expressed and used in the Kodu language (*RQ1*).

4.1 Variables and Scope

Simply, a variable is a symbol that holds a value. In Kodu as in other languages, variables can have a local or global scope. Here, we describe how global, local, and random variables are expressed and used in Kodu.

Global Scope

Global variables in Kodu can be read from or written to by any character. In Kodu, the scores, which hold integer

values, act as global variables. There are 37 global scores that can be maintained by any program, one for each letter of the English alphabet and color supported by Kodu.

Local Scope

Characters have four local properties that are analogous to local variables. These are color, glow, expression, and health. Color, glow, and expression each hold an enumerated value (e.g., orange, happy), and health holds an integer value. A character can read and write all properties for itself and some of the properties of other characters. For example, *kodu*[1] can be programmed to *change* the health property of a *cycle*, but it cannot *check* the health of the *cycle*.

Random

A common way to introduce nondeterminism in programming is through random variables, and Kodu has adopted this practice. There are three random variable tiles that can be used, one for time, one for score, and one for color. These can be used as filters or modifiers to trigger behavior or set a property or value at random.

4.2 Boolean Logic

The Kodu language allows users to express logical negation, conjunction, and disjunction in their programs, hence exposing the user to these fundamental concepts.

Negation

The concept of negation is perhaps the most obvious among the boolean logic constructs that can be explored through the Kodu language. A specific tile, called the *not* tile, can be applied to any condition in any rule. Consider the following block of code (where A is a condition and B is an action):

```
(1) When not A do B
```

Here, when condition A is met, then nothing happens. However, if A is not met, then B executes.

A byproduct of the logical negation is the ability to express *if – then – else* statements. All rules in Kodu represent conditional statements of the form *if – then*, but the *not* title can introduce an *else* clause. Consider the following code:

```
(1) When A do B
(2) When not A do C
```

Here, when condition A is met, then action B executes. However, if A is not met, then C executes. In other words, we have *if A, then B, else C*.

Conjunction

The ability to indent rules in the Kodu language creates a logical conjunction in the program structure. Indented rules will be evaluated when the condition on the parent rule is met. Consider the following block of code:

```
(1) When A do B
(2)     When C do D
(3)     When always do E
```

If condition A is not met, then lines (2) and (3) will never be evaluated. However, when condition A, then the conditions on line (2) and (3) are also checked (the *always* condition in line (3) evaluates to true, always). This logical structure forms an implication, where $A \Rightarrow B$, $A \wedge C \Rightarrow D$, and $A \wedge 1 \Rightarrow E$. And so, we have a logical conjunction of actions $B \wedge E$ when $A = true$ and a logical conjunction of conditions $A \wedge C$ to perform action D.

[1]The *kodu* is a character that can be programmed, like the *cycle* or the *fish*. It is typically the main character in a Kodu game.

```
-------------------------------------------------
|         Page 1          |         Page 2          |
| (1) When A do Switch 2 | (1) When C do Switch 3 |
| (2) When B do Switch 4 |                         |
-------------------------------------------------
|         Page 3          |         Page 4          |
| (1) When D do Switch 2 | (1) When F do Switch 3 |
| (2) When E do Switch 4 |                         |
-------------------------------------------------
```

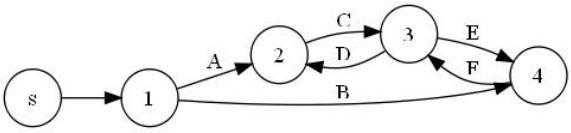

Figure 3: Example Object Control Flow

Disjunction

Kodu users can achieve logical disjunction by using two or more rules with different conditions and the same action. Consider the following block of code:

```
(1) When A do B
(2) When C do B
```

Here, we have two implications, $A \Rightarrow B$ and $C \Rightarrow B$, which can be simplified into $A \vee C \Rightarrow B$, representing a logical disjunction in the program. One side effect of the disjunction here is that B will get executed twice in the case that A and C both evaluate to true.

4.3 Objects

In object-oriented programming, objects are data structures that encapsulate fields and methods in a single package. Each character in Kodu is an object, and each has locally scoped variables, discussed previously. When programming, a user can *create* new characters, *delete* characters, and *clone* characters, and characters can also be created and removed during runtime. In most cases, a clone performs a deep copy including programming and properties. However, if the character has been declared to be *creatable*, then a clone will create a new object that references the original and mimics its behavior, forming a rudimentary class system. When a creatable character is modified, the changes to its programming impact all clones.

4.4 Control Flow

Kodu users can introduce non-linear program flow through the use of multiple pages of code and transitions between the pages. As each page represents a state for a character, pages map to nodes in a control flow graph, where each graph's start node is labeled *s*. The edges between nodes are labeled with the condition on which page switch is made, with the exception of the outgoing edge from the start state, which is always taken. Figure 3 shows a program fragment with four pages, which represents the behavior of a character from a program gathered for our study (selection criteria described in Section 6), and the generated control flow graph.

While several analyses are possible on the control flow graph, we can determine certain characteristics of the character behavior that are related to computer science concepts by detecting cycles and calculating the maximum *fan-in* and *fan-out*. A control flow graph that contains a cycle, like $2 \rightarrow 3 \rightarrow 2$ in our example, represents the introduction of iteration, as the character can cycle through those states. Nodes that have a high fan-out value (e.g., nodes 1 and 3 have fan-out = 2) are indicative of conditional control flow, where the same state can transition to one of many different states. A high fan-in value (e.g., nodes 2 and 4 have fan-in = 2) is an indication of lightweight code reuse, since that code can be called from many different locations.

5. INSTRUMENTATION

We have shown that many computer science concepts can be expressed through the Kodu language, but unless users are taking advantage of their ability to configure, modify, and program their world, these benefits of Kodu are lost. To address *RQ2*, we have instrumented Kodu so it logs the amount of time spent in each area of the tool. The instrumentation was deployed with Kodu v1.0.65 released on July 1, 2010 for the PC Kodu users only. We collected instrumentation data for 38 days from July 1, 2010 - August 8, 2010. This resulted in data for 4,229 sessions originating from 1,580 installations, for an average of 2.7 sessions per install. (We use installations as a proxy for users. Since Kodu does not require a login ID, we do not know the number of actual users covered by this data.)

We instrumented Kodu using timers that measure the amount of time spent in each of several areas of the user interface. Table 1 shows the average (*avg*), minimum (*min*), 25th percentile (*q1*), median (*med*), 75th percentile (*q3*), and maximum (*max*) percentages of time spent per session in each of five areas of the Kodu Game Lab: *Menu* includes time spent on the main menu, home menu, save menu, and searching for games to load; *Prog.* includes time spent in the programming UI (shown in Figure 2); *Add, Scan* refers to time spent adding characters and exploring the world without changes; *Edit* refers to time spent editing the terrain, editing character properties (e.g., creatables, speed), and editing world properties (e.g., lighting, score visibility); *Play* refers to time spent actually running the game. The final column, *Time*, shows the time spent per session in minutes. The average active time per session was over 25 minutes, with a median of over 14 minutes.

Table 1 shows that an average of 14.6% of each session's time was spent in the programming UI, and 15.6% of each session's time was spent editing the characters and world. In total, this means that an average of 30.2%, or nearly one-third, of the time spent per session involved programming or tweaking parameters in the world (compared to an average of 22.9% of the time spent playing the game). Further, while we did not segment time spent adding characters versus exploring the world, it can be argued that part of the *Add, Scan* time involves program design, as this is when characters are added to the world. Additionally, among the 4,229 sessions, 2,578 (60.9%) entered the programming UI (the interface shown in Figure 2), and 3,308 (78.2%) used one of the edit tools for the world, characters, or terrain. Among those sessions that entered the programming UI, users viewed or edited their code an average of 12.7 times per session and played their game an average of 13.7 times, illustrating that users likely swap between programming and playing as they tweak and test their programs. This rapid switching between developing and testing has been shown to be a successful problem solving technique for students learning how to program [8].

Table 1: Percent of Time Spent in Kodu (Time column shown in minutes)

	Menu	Prog.	Add, Scan	Edit	Play	Time
avg	16.1%	14.6%	30.9%	15.6%	22.9%	25.36
min	0.1%	0.0%	0.2%	0.0%	0.1%	0.16
q1	4.7%	0.0%	18.9%	0.1%	10.0%	5.31
med	10.7%	9.2%	28.4%	9.6%	18.7%	14.30
q3	22.1%	25.8%	40.5%	24.5%	30.9%	36.32
max	98.0%	94.8%	97.1%	92.1%	96.2%	335.32

6. ANALYSIS OF KODU PROGRAMS

Section 4 describes many basic yet important computer science concepts that can be expressed using the Kodu language, and Section 5 shows that end users are regularly using the programming interface while interacting with Kodu. Here, we explore how often each computer science concept is employed in Kodu programs created in the wild (*RQ3*).

To achieve this goal, we obtained 346 Kodu programs created by Xbox users and shared through the Xbox Live community between June 20, 2009 and July 30, 2010. In the Xbox community, users can upload their programs to any of three locations; we collected all the programs from one of those repositories (often users upload to all three). We do not have any demographic information about these users.

6.1 Variables

Table 2 shows the frequency within the population of programs of each type of variable defined in Section 4.1, *Global Variables*, *Local Variables*, and *Random Variables*. The *CS Concept* column indicates the computer science concept being evaluated, *# Games* indicates how often the concept appears in the population, *% Games* shows the percentage of the population, and *% Rules* indicates the average percentage of rules per program that exhibit the concept.

Over half of the programs use global variables in some capacity, and the average number of global variables used per program is 4.1. Within those programs, an average of 23.9% of the total rules interact with a global variable in some capacity. Approximately two-thirds (67%) of these programs perform more reads than writes on the global variables; 29% perform more writes than reads, and 4% perform equal numbers of writes and reads per variable. The average number of reads per variable is 4.0, and the average number of writes is 2.3. This indicates that in the general case, the scores are used to check and maintain program state, as they are read nearly twice as often as they are written, on average.

We found that 81.2% of the games have characters that read the local variables of other characters, whereas only 64.5% perform a write on the local variables. Additionally, among the games that read local variables, 31.8% of the rules perform a read; among the games that write to the local variables, only 17.9% are involved with a write. These numbers indicate that local variables are used to trigger other actions more often than they are modified.

Random variables are present in over one-third of the games, and impact nearly 10% of the rules in those games.

Table 2: Variables and Boolean Logic in Population

CS Concept	# Games	% Games	% Rules
Global Variables	196	56.6%	23.9%
Local Variables – Read	281	81.2%	31.8%
Local Variables – Write	223	64.5%	17.9%
Random Variables	126	36.3%	9.7%
Logical *Not*	61	17.6%	4.4%
If-Then-Else Statements	29	8.4%	8.1%
Logical *And* Condition[2]	17	20.9%	18.5%
Logical *And* Action[2]	13	16.0%	15.6%
Logical *Or*	208	59.9%	16.6%

In general, users put random variables in the condition (62%) rather than the action (38%). A random variable in the condition indicates that the user desires a controlled action to occur at some non-deterministic condition.

6.2 Boolean Logic

The last five rows of Table 2 show the frequency of games and rules within the population that utilize the boolean logic constructs described in Section 4.2. The logical *not* is only present in 17.6% of the games we studied, and among those games, 47.5% of them use the *not* tile to form an *if - then - else* statement. The creation of these predicates maps very naturally to the use of conditionals in traditional programming languages.

The presence of the logical *and* is dependent on the use of indentation in the programming rules[2]. Among the games for which this feature was available, only 22.2% used this feature, but among those games, 28.7% of the rules were indented. And so, even a smaller percentage of the games utilize logical *and* in the condition and in the action. Among the indented rules, we found that 63.6% used the logical *and* in the condition, and 36.4% used the logical *and* in the action. The *and* in the condition implies that a complex state is needed in order to trigger the action, while the *and* in the action binds multiple actions to the same condition.

Duplicating the condition among multiple rules on a page with disparate actions is another way to bind multiple actions to the same condition. Among the population, this was observed in 286 (82.4%) of the games, and within those games, 35.0% of the rules had a condition identical to another condition on its page. If we control for the *when always* condition, then duplicate conditions exist in 207 (59.7%) of the games, and within those games, 17.8% of the rules had a condition identical to another on the page. Among the population for which the indentation was available, 76.5% of the games contain duplicated conditions, and this number only drops to 65.4% if we control for *when always*. These instances can be seen as missed opportunities for usage of the indention feature, which is a more explicit representation of the logical *and*. Further, indentation can remove some code duplication that is exhibited through duplicate conditions and promote reuse.

The logical *or* is the most common among the boolean logic structure, and represents the same action tied to disparate conditions. The high frequency may be the result of multiple similar conditions in which the user does not know which actually triggers the event, but further study is needed to validate that conjecture.

[2]The indentation feature needed for the logical *and* was introduced on March 19, 2010. These values consider only the 81 (23.4%) games published after that date.

Table 3: Object and Control Flow in Population

CS Concept	# Games	% Games	% Objects
Programmed Characters	338	97.7%	74.7%
2+ States	202	58.4%	40.0%
Cycles	166	47.9%	29.3%
Fan-in > 1	174	50.3%	47.5%
Fan-out > 1	123	35.5%	52.0%

6.3 Objects

Within the population of programs, 342 (98.8%) of the games contain characters (e.g., kodus, cycles, rocks), with an average of 28.3 characters per game. While some characters have no behavior, nearly three-fourths of the characters per game contain some programming, as shown in the *Programmed Characters* row of Table 3. The *CS Concept*, *# Games*, and *% Games* columns are similar to Table 2, and *% Objects* indicates the average percentage of characters per game that utilize the concept. Overall, 97.7% of the games contain programmed characters. Games also contain an average of 5.7 distinct character types (median is 5), indicating that if the author included one character of a particular type (e.g., tree, kodu, rock), they included many of that type.

On average, most of the characters (60.0%) included in a world have programming of some kind. For any world, an average of 25.2% of the characters contain no programming. Conversely, the remaining 74.8% of of the characters have programmed behavior, and the average number of rules per character is 5.14, which indicates that users are not only taking time to give characters behavior, but that the programs likely contain complex logic. In terms of creatables, 175 (50.6%) of the games contain programming to generate a creatable character during runtime.

6.4 Control Flow

In the control flow analysis, we analyze the structure of the control flow graphs that represent characters behavior, generated as described in Section 4.4. Table 3 shows the frequency of games that contain at least one character with a certain property, where *2+ States* indicates that an character uses multiple pages, *Cycles* indicates that an character's control flow graph contains a cycle, and *Fan-in > 1* and *Fan-out > 1* indicate that at least one node in at least one character's control flow graph has a fan-in or fan-out value greater than one, respectively.

Most programs (58.4%) have used the page system and have at least one character with two or more states. Additionally, 166 of the programs (47.9%) have an character with a cycle, and in fact 29.3% of the characters in those programs contain cycles. Among the characters with two or more states, 45.8% contain a cycle. About half of the programs have a node with a fan-in greater than one, with the average max fan-in among all programs being 2.8. Additionally, over one-third of the programs have a fan-out greater than one, with the average max fan-out among all programs being 1.7. The higher average max fan-in value may indicate that the characters typically have a "hub" node, or

page, that contains default behavior, but further analysis is needed to validate that conjecture.

7. THREATS TO VALIDITY

There are two threats to validity that warrant discussion in this paper. The first is an internal threat concerning the connection between expressing computer science concepts and learning those same concepts, and the second is an external threat concerning artifact selection.

First, we have mapped the Kodu language onto computer science concepts and shown how these concepts can be expressed through Kodu. However, we cannot guarantee that just because a concept is expressed that it is necessarily learned. An empirical user study with pre- and post-tests would be necessary to evaluate the actual user learning.

The second threat is that the programs we analyzed come from those self-selected by the users in the Xbox community as being good enough to share, so they may represent an upper-bound in complexity. Additionally, since we cannot determine the provenance of any single Kodu program, we do not know how much of the program complexity is the product of the author versus how much resulted from an existing program that was modified by the author. However, since these programs come from the Xbox community and were created in an unsupervised environment, and since they were self-selected for sharing, we conjecture that the programming is mostly original.

8. CONCLUSIONS

In this work, we have shown that the Kodu language can be used to represent computer science concepts by pairing language constructs with fundamental concepts in computer science, but further analysis is needed to verify that users are actually grasping these concepts through Kodu. We found that users spend more time programming and configuring their programs than they do playing them, which indicates that the Kodu has reached its goal of making programming accessible to all users. Additionally, through an evaluation of 346 Kodu programs created by the user community, we found the programs to be large, complex, and exhibit many fundamental computer science concepts, which indicates that Kodu may be useful not just as a beginning programming language, but as a launching point from which users can smoothly transition to more mainstream programming languages.

Acknowledgments

We would like to give special thanks to the Kodu team and the FUSE Labs, especially Stephen Coy, Will Portnoy, Matt MacLaurin, Eric Anderson, Brad Gibson, and Rachel Schiff, for their valuable feedback and support of this work.

9. REFERENCES

[1] The impact of web 2.0 technologies in the classroom. State of Victoria: Department of Education and Early Childhood Development, December 2009.

[2] Explorer kodu club. http://koduclub.org/default.aspx, Retrieved August 1, 2010.

[3] Greenfoot. http://www.greenfoot.org/, Retrieved September 1, 2010.

[4] Kodu game lab. http://research.microsoft.com/en-us/projects/kodu/, Retrieved August 1, 2010.

[5] Kodudes – the write buzz. http://sites.google.com/site/koduxperts/, Retrieved August 1, 2010.

[6] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. In *Northeastern conference on The journal of computing in small colleges*, pages 107–116, 2000.

[7] A. Dahotre, Y. Zhang, and C. Scaffidi. A qualitative study of animation programming in the wild. In *ESEM '08: Symposium on Empirical software engineering and measurement*, 2010. to appear.

[8] B. Hanks and M. Brandt. Successful and unsuccessful problem solving approaches of novice programmers. In *Symposium on Computer science education*, pages 24–28, New York, NY, USA, 2009. ACM.

[9] C. Kelleher. *Motivating programming: using storytelling to make computer programming attractive to middle school girls*. PhD thesis, Pittsburgh, PA, USA, 2006. Adviser-Pausch, Randy.

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

[11] C. Kelleher, R. Pausch, and S. Kiesler. Storytelling alice motivates middle school girls to learn computer programming. In *Conference on Human factors in computing systems*, pages 1455–1464, 2007.

[12] C. M. Lewis. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Symposium on Computer science education*, pages 346–350, New York, NY, USA, 2010. ACM.

[13] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A sneak preview. In *C5 '04: Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, 2004.

[14] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. Learning computer science concepts with scratch. In *ICER '10:Workshop on Computing education research*, pages 69–76, 2010.

[15] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[16] D. Parsons and P. Haden. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Conference of the National Advisory Committee on Computing Qualifications*. Citeseer, 2007.

[17] M. Resnick. *Turtles, termites, and traffic jams: explorations in massively parallel microworlds*. MIT Press, Cambridge, MA, USA, 1994.

[18] K. T. Stolee. Kodu language and grammar specification. Microsoft Research whitepaper, Retrieved September 1, 2010.

[19] L. Werner, J. Denner, M. Bliesner, and P. Rex. Can middle-schoolers use storytelling alice to make games?: results of a pilot study. In *Conference on Foundations of Digital Games*, pages 207–214, 2009.