Scratch for Budding Computer Scientists

David J. Malan
Division of Engineering and Applied Sciences
Harvard University
Cambridge, Massachusetts, USA
malan@post.harvard.edu

Henry H. Leitner
Division of Continuing Education
Harvard University
Cambridge, Massachusetts, USA
leitner@harvard.edu

ABSTRACT

Scratch is a "media-rich programming environment" recently developed by MIT's Media Lab that "lets you create your own animations, games, and interactive art." Although Scratch is intended to "enhance the development of technological fluency [among youths] at after-school centers in economically disadvantaged communities," we find remarkable potential in this programming environment for higher education as well.

We propose Scratch as a first language for first-time programmers in introductory courses, for majors and non-majors alike. Scratch allows students to program with a mouse: programmatic constructs are represented as puzzle pieces that only fit together if "syntactically" appropriate. We argue that this environment allows students not only to master programmatic constructs before syntax but also to focus on problems of logic before syntax. We view Scratch as a gateway to languages like Java.

To validate our proposal, we recently deployed Scratch for the first time in higher education via Harvard Summer School's Computer Science S-1: Great Ideas in Computer Science, the summertime version of a course at Harvard College. Our goal was not to improve scores but instead to improve first-time programmers' experiences. We ultimately transitioned to Java, but we first introduced programming itself via Scratch. We present in this paper the results of our trial.

We find that, not only did Scratch excite students at a critical time (*i.e.*, their first foray into computer science), it also familiarized the inexperienced among them with fundamentals of programming without the distraction of syntax. Moreover, when asked via surveys at term's end to reflect on how their initial experience with Scratch affected their subsequent experience with Java, most students (76%) felt that Scratch was a positive influence, particularly those without prior background. Those students (16%) who felt that Scratch was not an influence, positive or negative, all had prior programming experience.

Categories and Subject Descriptors:

D.3.2 [**PROGRAMMING LANGUAGES**]: Language Classifications—*Scratch*; D.3.m [**PROGRAMMING LANGUAGES**]: Miscellaneous K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Computer science education*;

General Terms: Human Factors, Languages

Keywords: Java, languages, programming, Scratch

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.
Copyright 2007 ACM 1-59593-361-1/07/0003 ...$5.00.

1. INTRODUCTION

Most programming languages, on first glance, "look like Greek" to the untrained eye, an amalgam of English and unfamiliar syntax. Consider, after all, even the simplest of Java programs (Figure 1), whose sheer volume of keywords and syntax defies explanation on an introductory course's first day. The simplest (if not most common) explanation of why "hello, world" must be written in this way is perhaps a wave of the hand and a promise to revisit `public`, `static`, `void` and other potential distractions in the future. Even if these and other other keywords are introduced gradually, too often do students remain distracted on subsequent days by semicolons and other, fundamentally uninteresting, details. To be sure, appreciation and mastery of precision is important in learning to program computers. But, in the first weeks of an introductory course (for majors or non-majors), too often do semicolons and their syntactical cousins delay, if not downright discourage, students' appreciation and mastery of more fundamental programmatic constructs (*e.g.*, conditions, loops, variables, *etc.*) as well as logic itself. We daresay that languages like Java challenge students to master programmatic overhead before programming itself: students must become masters of syntax before solvers of problems.

Moreover, so accustomed are students today to graphical interfaces, "hello, world," whether written by student or teacher, cannot help but underwhelm. And, yet, in courses designed to recruit and retain budding computer scientists, it is perhaps just as important to excite as it is to instruct. To introduce from the start a graphical library like AWT [16] or Swing [15], though, is likely to overwhelm.

It is with these hurdles in mind that we propose Scratch [8] as a first language for first-time programmers in introductory courses, for majors and non-majors alike. Developed by the Lifelong Kindergarten (LLK) research group [6] at MIT's Media Lab [12], Scratch is a "media-rich programming environment" [10] that empowers students—on day one—to implement animations, games, and interactive art. Although Scratch was designed "to enhance the development of technological fluency [among youths] at after-school centers in economically disadvantaged communities" [10], we find remarkable potential in this programming environment for higher education as well. Scratch enables students to program with a mouse, presenting programmatic constructs as blocks (*i.e.*, puzzle pieces) that only fit together if "syntactically" appropriate. Among these blocks are such fundamentals as statements, Boolean expressions, conditions, loops, and variables. Other blocks, meanwhile, offer pseudorandomness as well as multithreading and event-handling,

```
class Hello
{
    public static void main(String [] args)
    {
        System.out.println("hello, world");
    }
}
```

Figure 1: The sheer volume of keywords and syntax in even the simplest of Java programs defies explanation on an introductory course's first day.

Figure 2: With Scratch, Figure 1 becomes the above.

features whose immediate deployment usually isn't practical (or, at least, typical) in first courses in computer science with languages like Java. With these blocks can students program one or more "sprites" (*i.e.*, characters) on a "stage," the end result of which is Scratch's promise of some animation, game, or interactive art.

In effect, Scratch lowers the bar to programming, empowering first-time programmers not only to master programmatic constructs before syntax but also to focus on problems of logic before syntax. We thus view Scratch as a gateway for students to languages like Java.

To validate our proposal, we recently deployed Scratch for the first time in higher education by way of Harvard Summer School's Computer Science S-1: Great Ideas in Computer Science, a summertime version of a course at Harvard College by the same name. Per its syllabus, this course "is a broad introduction to the most important concepts in computer science." Not only does the course present programming as one such concept, it also laces programming throughout the course as a mechanism for exploring other concepts. Although we ultimately transitioned to Java for most of the course's examples and problem sets, we first introduced programming itself via Scratch.

Insofar as our goal was not to improve scores but instead to improve first-time programmers' experiences, we surveyed students throughout the summer for their thoughts on Scratch and its impact on their education. Ultimately, most students (76%) felt that their exposure to Scratch was a positive influence on their subsequent experience with Java. Among those (16%) who felt neither positively nor negatively influenced by Scratch, each had prior programming experience.

In the section that follows, we provide an overview of Scratch's interface and capabilities. In Section 3, we cite alternatives to Scratch, including one environment that we used in prior semesters. In Section 4, we describe our deployment of Scratch, thereafter elaborating on the results of our trial in Section 5. We conclude in Section 6.

2. ABOUT SCRATCH

Written in Squeak [1], an open-source implementation of Smalltalk-80, Scratch runs atop a virtual machine, ports of which exist for several flavors of Linux, Mac OS, UNIX, and Windows. According to LLK, a player for Scratch projects will soon exist as a Java-based plug-in for browsers as well.

Not only does Scratch allow students to import "costumes" and sounds for sprites, it also provides a built-in paint editor and sound recorder with which students can create the same. Scratch even allows for interaction with the physical world by way of sensors connected via USB.

Among other controls, Scratch's interface (Figure 3) offers students a *blocks palette*, a *scripts area*, a *selection area*, and a *stage*. The blocks palette offers students eight categories of color-coded building blocks, puzzle pieces of sorts that collectively govern sprites' behavior on the stage. Programming a sprite is as simple as selecting it in the selection area (after creating it with a click of a button) and dragging two or more blocks to the scripts area, where they will snap together if "syntactically" appropriate. Among these blocks are statements, Boolean expressions, conditions, loops, and variables as well as support for multiple threads and events (Figure 4). Execution begins when the student clicks the interface's green flag.

With Scratch, then, does "hello, world" (Figure 1) become a two-piece puzzle (Figure 2).

3. ALTERNATIVES TO SCRATCH

By no means is Scratch the first programming environment to lend itself to deployment among first-time programmers. Most computer scientists recall Logo [9], "the name for a philosophy of education and a continually evolving family of programming languages that aid in its realization."[1] More recent incarnations of Logo include NetLogo [17] and StarLogo [11]. Related in spirit, meanwhile, are Alice [4], Crickets [7], Karel the Robot [13], Karel++ [2], Karel J Robot [3], and JKarel [5].

If each of these environments has one weakness in our eyes, it is that its world is too restricted or its learning curve is too high, at least vis-à-vis Scratch. In fact, for a number of years, we deployed JKarel (as well as its predecessors). Though JKarel offers a Java-like syntax that does allow for a more seamless transition to Java itself (as do Karel and Karel++ for C and C++, respectively), Scratch's sprites are not limited to mere navigation along walls and collection of beepers, as are JKarel's robots. Rather, Scratch's world has "wider walls" [14], whereby students are free to express themselves programmatically in many more ways. In fact, among our own students' submissions were implementations in Scratch of fairy tales, fish tanks, and Frogger!

4. DEPLOYING SCRATCH

With our course's summertime version compressed into eight weeks (with two 2.5-hour lectures scheduled for each), we opted to spend two lectures and two problem sets on Scratch (*i.e.*, one week), after which we transitioned to Java for the remainder of the course.

In the first of our lectures, we introduced students to some of programming's most fundamental constructs, including statements, Boolean expressions, conditions, loops, and variables. We first presented each construct in the context of real-world "programs" written in pseudocode (*e.g.*, algorithms for changing a baby's diaper and putting on socks). We then re-visited each construct in the context of programs

[1]Harold Abelson, 1982.

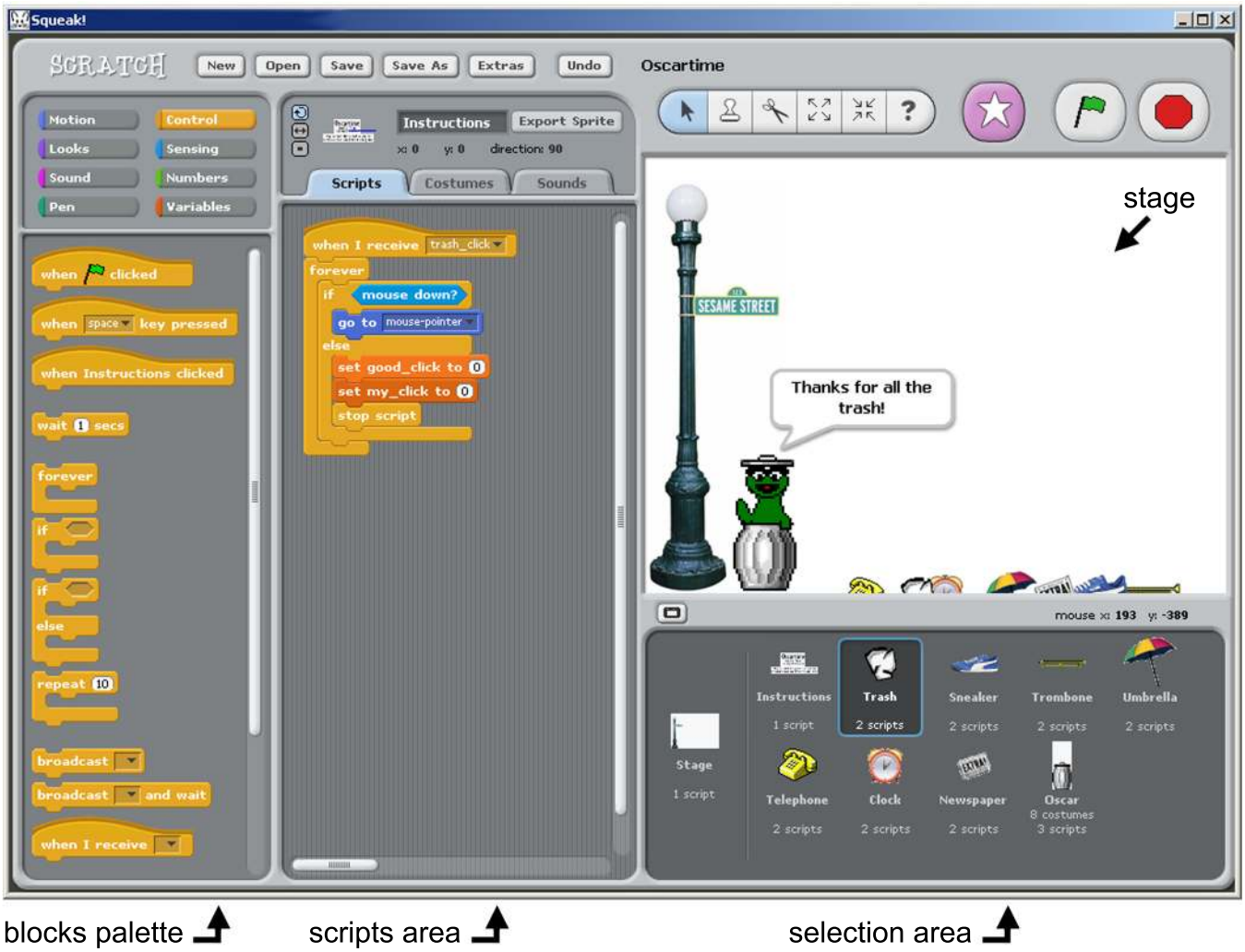

Figure 3: Scratch's interface consists of a *blocks palette*, a *scripts area*, a *selection area*, and a *stage*, along with other controls. Pictured is Oscartime, a game with nine sprites and nineteen scripts whose implementation we explored, among others, in class.

written in Scratch. Over the course of mere minutes, the latter grew in complexity from cats (*i.e.*, sprites) meowing once, to cats meowing pseudorandomly, to cats meowing only when "petted," to cats chasing birds (*i.e.*, other sprites). With each of these programs (and others) did we introduce additional programmatic constructs. With its controls so intuitive and its blocks so self-explanatory, we spent only moments explaining Scratch's interface during this introduction. Scratch was designed, after all, for youth, who likely have little patience for manuals, let alone lectures. To our own students, then, usage of the environment itself seemed obvious. We concluded this lecture with an introduction to threads and events.

In their first problem set on Scratch, students were presented with a challenge of few requirements and few limits: "have fun with Scratch and implement a project of your choice." Students were told only that their project: must have two sprites; must have at least three scripts in total; must use at least one condition, one loop, and one variable; must use at least one sound; and should probably use a few dozen puzzle pieces overall. Though we considered assigning instead several bite-sized tasks, each focused on one or more concepts, we ultimately decided upon the broader assignment. Insofar as our goal was to excite students, while still acquainting them with programming, we opted to entrust our goal to students' own senses of curiosity and creativity rather than impose on the experience constraints of our own. The results (Section 5) were impressive.

In the second of our lectures, we examined several students' submissions in detail, to familiarize the class not only with the process of writing code but reading and understanding that of others. We concluded with a preview of Java, translating certain constructions in Scratch to equivalent syntax in Java.

In their second problem set on Scratch, students were challenged to "build upon the work of another student" by downloading and modifying (noticeably) another student's initial submission (which we posted, with permission, in a gallery on the course's website).

In the section that follows, we present the results of this deployment along with reflections by students on the same. A link to these lectures and problem sets as well as this gallery appears in the Appendix.

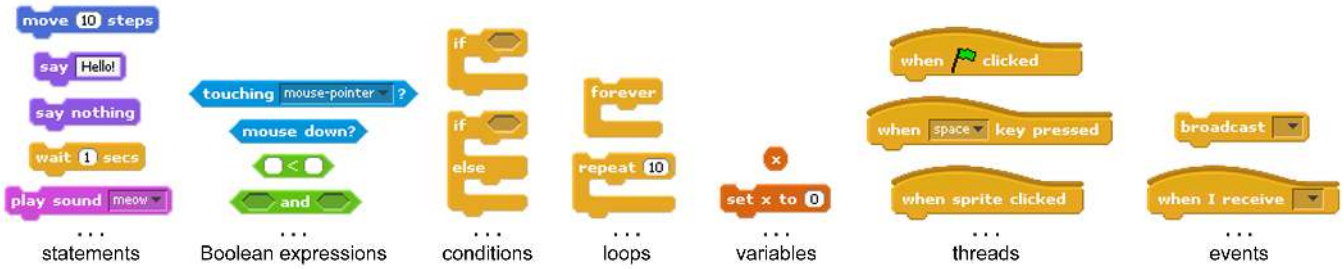

Figure 4: A sampling of Scratch's building blocks (*i.e.*, puzzle pieces), categorized in terms a budding computer scientist should understand. Blocks are shaped so that they only snap together if "syntactically" appropriate (*e.g.*, only hexagonal Boolean expressions fit inside conditions' hexagonal "holes"). Moreover, certain blocks (*e.g.*, conditions and loops) dynamically resize themselves to accommodate any number of nested blocks.

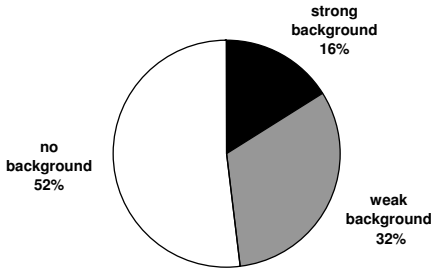

Figure 5: At term's start, we surveyed students about their prior programming experiences, if any. Among the 25 respondents, 52% had no background in programming whatsoever, 32% had weak backgrounds (*i.e.*, exposure to but limited experience with at least one language), and 16% had strong backgrounds (*i.e.*, at least one year's experience with at least one language).

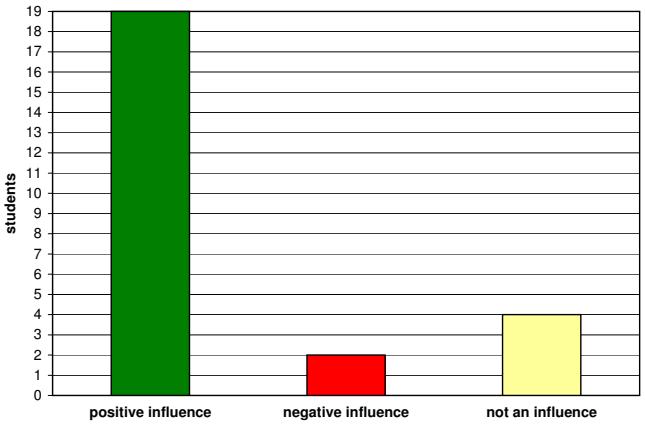

Figure 6: At term's end, we surveyed students on how their initial experience with Scratch affected their subsequent experience with Java. Among the 25 respondents, 19 (76%) felt that Scratch was a positive influence, 2 (8%) felt that Scratch was a negative influence, and 4 (16%) felt that Scratch was not an influence.

5. RESULTS

Insofar as our aim with this study was not to improve scores but to improve first-time programmers' experiences, we employed subjective measures for its assessment. To qualify and quantify the results of our trial, we surveyed students throughout the semester. On our surveys were questions about students' prior programming background, experience with Scratch, and subsequent acclimation to Java.

Among our 25 respondents, 52% had no background in programming whatsoever, 32% had exposure to but limited experience with at least one language, and 16% had at least one year's experience with at least one language (Figure 5).[2]

Though some students spent only 2 or 3 hours on their first problem set, others spent upwards of 20, implementing projects more advanced than any of those written in lecture. The median and mode of students' development times were 6 and 7 hours, respectively. Comments from students explain such investments of time: "Scratch is fun to use and really easy to learn, almost addictive in a way."

Clear from other comments was that Scratch does indeed excite: "Scratch was a ton of fun, and chances are one day when I get bored I will go back to it and make a game. It was really nice having visible rewards for the work instead of 'Oh my god! those randomly generated numbers sorted themselves!'"

[2]Among the languages that some of our students had seen or used before were BASIC, C, C++, Java, JavaScript, NetLogo, Perl, PHP, Scheme, and Visual Basic.

Common among students was this appreciation of Scratch's immediate rewards: "[My] brother is a senior programmer at Apple so occasionally he hands me a book and tells me to learn something. . . . The thing that didn't keep me learning Java and C++ was that there were hardly any tangible rewards. The thing I really wanted to make was a game but according to my brother it was next to impossible for me to do it. Where as [*sic*] with Scratch it was extremely easy for me to do it."

Confident after two lectures and problem sets that Scratch had engaged, we waited until term's end for evidence that it had also enlightened. When asked at term's end to reflect on how their initial experience with Scratch affected their subsequent experience with Java, 19 students (76%) felt that Scratch was a positive influence, 2 (8%) felt that Scratch was a negative influence, and 4 (16%) felt that Scratch was not an influence (Figure 6).

Among the positive respondents were explanations like: "The experience with Scratch helped me get a general idea of how to think like a programmer. Specifically, working with Scratch helped me to develop an intuitive sense of how loops and variables worked in a generic sense. This . . . subsequently made it much easier to adapt to the particular syntax used in Java for implementing such constructs." In the words of another, "Though we did not learn Java syntax by using Scratch, we learned the type of thinking necessary to implement simple programs. . . . I was able to approach

the first Java programs with an idea of how to tackle the problems. Though I did not yet know how to create a `for` loop, I knew when a `for` loop was necessary because I had used loops in my Scratch program."

Comments from one negative respondent were bitter-sweet: "I feel Scratch negatively influenced me for the rest of the course. Scratch was a lot of fun to use, and it was really easy. Then we started coding in Java and its [*sic*] about 100 times harder than Scratch, and the results are much less enjoyable than what I could easily achieve in Scratch. I think Scratch would have been better to have fun with after . . . Java."

Worthy of note, though perhaps not surprising, is that each of the neutral respondents admitted prior programming experience. One such student's comments stood out: "I think Scratch didn't really help me with Java. I had fun with Scratch and I see how it could serve as a didactic tool for some people but I would have preferred to jump straight into Java. The elements of programming that Scratch attempts to teach are not particularly difficult to understand and I feel may be 'safely' introduced using Java itself. . . . I feel we could have progressed a lot more into Java had we jumped directly into it."

With students' reflections nonetheless quite positive overall, we ultimately judged this deployment of Scratch a success. Not only does the programming environment seem to excite students whom Java, as a very first language, might fail to engage, it also appears to ease the transition for those without background to more cryptic syntax ahead. As a gateway to languages like Java, then, Scratch appears a viable choice. As one student confirms, "I had a great time using Scratch, and found it [a] very rewarding way to get into programming."

Another student puts it more simply: "It is awesome."

6. CONCLUSION

We present in this paper the motivation for and results of our deployment in higher education of Scratch, a new programming environment that empowers students to implement animations, games, and interactive art. Based on our experience with just over two dozen students in Harvard Summer School's Computer Science S-1: Great Ideas in Computer Science, we propose Scratch as a viable gateway to languages like Java. Not only does Scratch excite students at a critical time (*i.e.*, their first foray into computer science), it also familiarizes the inexperienced among them with fundamentals of programming without the distraction of syntax.

To be sure, Scratch does not provide every construct available in languages like Java. Nor does it support data types, data structures, methods, parameters, return values, inheritance, or polymorphism, all of which might be appropriate to introduce in introductory courses. But to emphasize what Scratch lacks is to understate what it offers: its simultaneous simplicity and power are what engage and excite students in the first place. Once hooked by Scratch, students can still be handed to Java.

In future semesters will we experiment with variations on this past summer's lectures and problem sets. In the meantime, it is with the success of our own deployment in mind that we propose Scratch as a first language for first-time programmers in introductory courses in computer science.

APPENDIX

Available for download at
`http://www.eecs.harvard.edu/~malan/`
are this trial's lectures and problem sets as well as students' submissions, posted with their permission.

ACKNOWLEDGEMENTS

We extend our thanks and congratulations to LLK for its wonderful work on Scratch. We are grateful, in particular, to Mitchel Resnick, John Maloney, Natalie Rusk, Amon Millner, and Shaundra Daily for their inspiration and support of this work. We also thank S-1's teaching fellows—Rebecca Nesson, Kevin Wang, and James Dowdell—as well as S-1's students, without whose thoughts and efforts this work would not have been possible.

REFERENCES

[1] Apple Computer, Inc. Squeak. `www.squeak.org`.

[2] J. Bergin, M. Stehlik, J. Roberts, and R. E. Pattis. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley and Sons, Inc., 1996.

[3] J. Bergin, M. Stehlik, J. Roberts, and R. E. Pattis. *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. CafePress.com, 2006.

[4] Carnegie Mellon University. Alice v2.0. `www.alice.org`.

[5] H. Wellenius. JKarel. `www.fas.harvard.edu/~libe50a/jkarel.html`, 2003.

[6] Lifelong Kindergarten, MIT Media Lab. `llk.media.mit.edu`.

[7] Lifelong Kindergarten, MIT Media Lab. Crickets. `llk.media.mit.edu/projects.php?id=1942`.

[8] Lifelong Kindergarten, MIT Media Lab. Scratch. `weblogs.media.mit.edu/llk/scratch/`.

[9] Logo Foundation. Logo. `el.media.mit.edu/logo-foundation/`.

[10] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A Sneak Preview. In *Second International Conference on Creating, Connecting, and Collaborating through Computing*, pages 104–109, Kyoto, Japan, 2004.

[11] Massachusetts Institute of Technology. StarLogo. `education.mit.edu/starlogo/`.

[12] MIT Media Laboratory. `www.media.mit.edu`.

[13] R. E. Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley and Sons, Inc., 1995.

[14] M. Resnick and B. Silverman. Some Reflections on Designing Construction Kits for Kids. In *Proceedings of International Conference for Interaction Design and Children*, Boulder, CO, 2005.

[15] Sun Microsystems, Inc. Java Foundation Classes (JFC/Swing). `java.sun.com/products/jfc/`.

[16] Sun Microsystems, Inc. The AWT in 1.0 and 1.1. `java.sun.com/products/jdk/awt/`.

[17] U. Wilensky, Center for Connected Learning and Computer-Based Modeling, Northwestern University. Logo. `ccl.northwestern.edu/netlogo/`.