Through the Looking Glass: Teaching CS0 with Alice

Kris Powers
Tufts University
161 College Ave.
Medford, MA 02155
+1 (617) 627-4924
Kris.Powers@tufts.edu

Stacey Ecott
Tufts University
161 College Ave.
Medford, MA 02155
+1 (617) 627-2225
Stacey.Ecott@tufts.edu

Leanne Hirshfield
Tufts University
161 College Ave.
Medford, MA 02155
+1 (617) 627-2225
Leanne.Miller@tufts.edu

ABSTRACT
This work analyzes the advantages and disadvantages of using the novice programming environment Alice in the CS0 classroom. We consider both general aspects as well as specifics drawn from the authors' experiences using Alice in the classroom over the course of the last academic year.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms
Human Factors, Languages.

Keywords
novice programming environments

1. INTRODUCTION
Learning to program is hard. And so for the last few decades, computing educators have developed a myriad of environments to help novices learn to program. Currently, a veritable barrage of these environments is in development. Many of these environments are not well studied, and few make any impact outside of the circle of influence of the developer. Among the few exceptions to this is the 3-D graphical programming environment, Alice.

Alice is in the "microworld" category of novice programming tools which generally allow storytelling to be incorporated into programming. This approach is captured by numerous novice programming environments, dating to back to the original Karel the Robot [21] and continuing today with environments like Jeroo [14], Greenfoot [12], and Alice [1]. In microworld environments, students create characters and program their behavior. Alice is unique amongst such environments in several ways. It supports programming of a rich collection of 3-D characters that can be chosen from a sizeable library, and supports graphical drag and drop programming in which syntax errors cannot occur.

The importance of Alice amongst the CS community cannot be denied. It has been the topic of over $3.5 million in grant development [7], and has been the subject of related efficacy studies. Alice has had a consistent SIGCSE and ITiCSE presence over much of the last decade in paper presentations, posters, and workshops. The most recent summer 2006 NSF workshop for teaching with Alice drew a crowd of over 120 educators, drawn from academic institutions ranging from high schools to research universities [7]. Over 100 academic institutions have tried Alice and 3 textbooks for Alice have been published in the last year.

In the fall 2005 and spring 2006 semesters we decided to teach our CS0 course using Alice. The course was a three semester-hour class with an additional 75-minute closed lab component. The class attracted a bi-modal student population, with a subpopulation of stronger students drawn from prospective CS majors and mathematics majors and a subpopulation of weaker students who selected the course as a "soft option" for fulfilling their math distribution requirement. In each semester the course was taught as a purely programming course (i.e., no breadth components were included, as is sometimes done in CS0). As will be discussed more fully later in the paper, two different approaches to using Alice were employed in each semester.

Since Alice has received such noteworthy attention, many CS educators are currently using, or are considering using the environment. While a wealth of publications praise the Alice environment, we address some important issues that should be considered when using Alice. We believe that Alice has the potential to be an incredible teaching tool. Our hope is that combined wisdom of many Alice educators can influence the development of Alice, make future uses of Alice more effective, and move Alice toward its potential. In this paper we give an overview of the Alice environment, share our experience of incorporating Alice into the CS classroom, and we discuss issues in transitioning from Alice to a high-level language such as Java or C++.

2. BACKGROUND
Alice is a learner-centered environment that facilitates programming by allowing the direct manipulation of objects using a limited set of simple commands. Alice allows users to manipulate 3D objects in a 3D world in order to create program animated movies. Information about each individual object can be accessed and manipulated independently of any written code. Simple stories can be realized by choosing an object in the world, such as an ice skater, and calling one of its methods, like Skate(). Students can incorporate this method call in their program using a graphical interface in which they drag the name of the method from the object and drop it into the calling method at a valid location as is done in Figure 1 (a main world method is provided, and is called when a program is run). This textual representation

Conference'04, Month 1–2, 2004, City, State, Country.

Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

is designed to allow lines of code to read as sentences describing the action an object is to take.

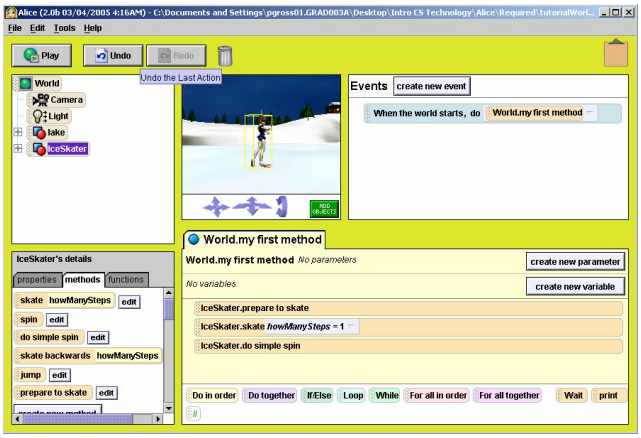

Figure 1 – The Alice Environment

Alice's drag and drop interface separates learning syntax from learning semantics. In fact, Alice forces students to create code that is always in a runnable state. For example, when a student drags an if-statement into the program code Alice forces the student to choose an appropriate Boolean value for the condition. Students will only be able to drag and drop valid relational statements into the if-statement condition in subsequent coding. The structure, therefore, does not allow for syntax errors, only errors in logic.

3. OBSERVATIONS AND ANALYSIS

The Alice website claims that Alice "addresses both the mechanical and sociological barriers that currently prevent many students from successfully learning to program a computer". In this section we provide feedback on Alice's strengths and weaknesses at addressing these barriers based on our experiences teaching our two Alice CS0 courses.

3.1 Addressing Sociological Barriers

Studies suggest that Alice can be used to attract and retain majors in CS. This is especially true for *at risk* majors (students without prior programming experience and/or students with weak mathematics backgrounds) [19]. Our observations are in line with these findings. Our Alice courses had a diverse population of students. Some students were taking the course for their math credit and had no direct interest in CS, others were taking the course because they were curious about the field, and a few others were programmers with prior programming experience.

Overall, students who came into the course with little to no programming and math experience had positive attitudes throughout the Alice portion of the course. When some students made mistakes in their code, they would laugh out loud at the resulting animation on the screen. The graphical output of the system created a comfortable programming environment where students could 'play around' with their code and visualize the program execution. However, we noted situations when the graphical output was frustrating to students because they wanted to program their 3-D objects to move like their real world counterparts. For example, students programming a rabbit to hop across the screen expected their rabbit to move like rabbits move in the real world. Striving to make characters move in realistic ways is often difficult to implement in Alice. We noted that many of our students became engrossed in the task of making their 3D object's movements fluid and realistic, while overlooking the more important goal of learning basic programming concepts.

Although making naturalistic character movements may have been slightly frustrating, Alice was successful at increasing *at risk* students' self confidence in their programming abilities. In fact, while working with Alice several *at risk* students commented to teaching assistants (TAs) that CS was not as difficult as they thought it would be. However, when we transitioned to C++ or Java these students became easily frustrated. These *at risk* students felt that although they could program in the 'easy' Alice environment, they did not really have the skills needed to do CS. One of the primary causes of this transitional frustration involved syntax. Since learning syntax rules can be difficult and frustrating for new CS students, Alice enables them to focus on more important structural programming components. While this approach is very useful at raising students' confidence in programming, we found that this confidence applies mostly to programming within the Alice environment and not to a transition from Alice to other programming languages.

Perhaps one of the most influential factors on student attitudes is the storytelling in Alice and other microworlds, which is considered to be an "intrinsically motivating activity"[1]. Interestingly, a review of APAOnline, PubMed, ACM digital library, and IS Web of Knowledge did not turn up any studies addressing storytelling as a motivation for learning as previously suggested [9]. In fact, in our experiences, the storytelling in Alice may have contributed to the 'wall' that students hit when transitioning to other languages. The storytelling pedagogy may address the sociologic issues students have with programming (not understanding relevance of CS or viewing CS as a socially isolating educational path) [15] [17], while misleading students as to the difficulty and nature of the discipline. While we were encouraged to see *at risk* students programming with confidence in Alice, new techniques are needed to improve student confidence during the transition from Alice's graphical, syntax free, storytelling environment to object-oriented textual programming.

3.2 Improved learning with Alice

Can Alice help students to understand programming concepts better? This section explores ways in which Alice has been purported to do so, and comments on the consistency of our experience with these claims.

3.2.1 Graphical output

The 3-dimensional graphical output in Alice is appealing to students and teachers because it shows students the output of their program in a manner that is easy to interpret. Graphical output is believed to help students to understand how control structures affect the output. In particular, small changes in program control structures can result in easily observable changes in the graphical output, and so students can explore the effects of small, incremental changes in their program on the output. The perceived benefits of graphical program output have been discussed at length (see e.g., [20] for a discussion). Of these, Alice is particularly beneficial for helping students to understand

the *large* effect on program output that can result from a *small* change in their program. Dijkstra [11] noted the difficulty presented by our natural predisposition to the preconception of an analogue world: e.g., if we press a bit harder on the gas pedal, we go correspondingly faster in our car. Programs do not act this way; "… a program has, unavoidably, the uncomfortable property that the smallest possible perturbations – i.e., changes of a single bit – can have the most drastic consequences." [11, p. 1400] While this behavior is obviously present is in any programming environment, the Alice manifestations of it are graphical, obvious, and generally laughable enough to be more closely noted by the students.

Another proposed benefit of Alice's graphical output is that it enhances students' understanding of their program state: "A 3D animation visually embodies the notion of state. The advantage afforded by the visual feedback of running the animation is that, at any instance in time, the students can easily see the current state of their program. The location of each object, its color, and its distance to other objects are all intuitively known. There is no need to draw abstract versions of memory maps with labeled boxes for variables, or for tedious hand traces of variable assignments." [27, p. 15]

Dann argues that program state is one of the most important features of virtual worlds [27]. In other words, "the program's state is immediately and always visible to the user. Each 3D object in Alice contains its own state as a set of properties. This eliminates the need for mutable variables… ." [27, p. 11]. But we would argue that state variables *are* mutable variables! They simply are not variables that the programmer defines. (They come for free when an object is instantiated.) Further, whether the programmer needs to manipulate these state variables depends on the problem being solved.

Cooper, Dann, and Pausch use a "programmer-defined, mutable variables late approach" in their text's pedagogy, but this is not inherent in Alice. For example, some programs that are very naturally suited to Alice –like a game with a score – still require programmer defined variables. In our first course, we followed [the CDP text], and deferred the presentation of programmer-defined variables until the end of the Alice portion of the course. When divorced from the concept of variables, student understanding of many concepts was remarkable! Loops, conditionals, events were amazingly easy. The expected difficulty returned the moment variables were introduced. (Whether these topics divorced from variables would remain just as easy in environments other than Alice is an interesting open question.) Further, all of the difficulty of understanding and manipulating state had yet to be tackled, and exposure to Alice does not make this very difficult concept any easier. Since explicit and facile manipulation of state is at the heart of programming, the extent to which the students "know how to program" before this point in the course is debatable.

3.2.2 Object programming in Alice

The Alice environment facilitates the objects-first approach to teaching object-oriented programming concepts: Alice provides an intuitive way for students to visualize objects. Every visual entity in the Alice environment is an object, making it easy for teachers to follow an objects-first approach to programming. An object tree on the left side of the screen allows students to view and manipulate all of the objects in their current environment. All characters that students add to their world are objects. Although teaching the concepts of objects in traditional programming courses is difficult, we found that the concept of objects was very easy for students to comprehend in this visual environment. Students understood that each object had its own methods and properties, and they were readily able to understand the differences between object-level methods and world-level (the Alice analogue of "static") methods.

Unfortunately, the Alice object model is neither thoroughly implemented, nor truly object-oriented. For instance, only generic objects can be passed as parameters. So we cannot, e.g., pass **`kermit`** as a parameter to a method, and execute his **`hop()`** method within that function. There is no polymorphism. There is no way to reference the invoking object **`this`** in a class-level method. Also, objects are never instantiated in code. The programmer must instantiate all the objects prior to running their code as a part of the world set up. The students can change the code of an object, but not the code of the class. A world with 5 frogs created from the **`Frog`** class allows all the frogs to have unique properties and methods. To create corresponding **`Frog`** classes, say **`Frog1`**, **`Frog2`** , etcetera, the student can save each object. This mechanism is actually exploited in one text [CDP] to give students the idea of inheritance. In our experience, this led even some of our brightest students to understand inheritance as a fancy word for the idea of cutting and pasting code from one object to another. In short, this implementation carries the potential to blur the relationship between objects and classes.

In the end, the visual objects in Alice made the initial shift to object-oriented programming in C++ and/or Java easier, but the analogy quickly broke down when students began exploring more advanced object-oriented concepts. This aspect of Alice is particularly disappointing as we initially thought that the object prominence in Alice programming would be its most beneficial feature.

3.3 Transitioning to other environments

Alice has long been advertised as a gentle way to prepare students to learn more typical CS1 languages like C++ and Java. This transition requires a number of significant conceptual challenges for the students, including:

- an IDE or editing/compiling tools
- syntax and debugging syntax errors
- code translation from high-level language (HLL) to low-level

The question arises as to what is the appropriate point for making this transition. In this section we consider the possibilities, and discuss the pros and cons of each.

Alice programming could be taught exclusively, in a separate course. This approach presents the advantage of having students learn only one programming environment. It also allows students to gain enough experience in Alice to develop more interesting and rewarding applications. The difficulty lies in the students' continued studies. If the next course does not explicitly build on the student's Alice background, then the impact of the Alice experience would seem to be significantly diminished. In fact, it

could even be detrimental! As was discussed in the preceding section, the object model of Alice is not consistent with real object-oriented environments. In our experience, if this difference is not explicitly taught, then the Alice (mis-) conceptions of object-oriented programming can persist. This indicates that the transition between Alice and a HLL should be explicitly taught, and not left to inference as a student moves through the curriculum.

When Alice and another HLL are taught in a single course, the decision of when to transition is significant. One possibility is to teach all Alice first and then follow it by the HLL, as we did in the fall of 2005. Another possibility is to intermix coverage. We attempted this in the spring of 2006 by interspersing topical coverage of Alice and C++. We began by covering programming fundamentals: expressions, variables, and control constructs. First the concepts were discussed in Alice, and then in C++. After that we went on to functions and parameters; again, the topics were first presented in Alice and then in C++. We are aware that other instructors are interspersing language coverage with much finer granularity, up to the point of demonstrating a concept in Alice and then explaining its analogue in HLL in a single class meeting.

In the Fall '05 semester the course was divided in half, with the first half of the course devoted to Alice, and the second half devoted to Java. The Alice portion of the course largely followed the 7-week, objects-early syllabus disseminated for the *Learning to Program with Alice* [9] text (except that list basics and additional material on variables were covered toward the end, and recursion was omitted). It is significant to note that this approach, and indeed the textbook itself, employ a late introduction of programmer-defined, mutable variables.

The second half of the course covered the first three chapters of *Objects First with Java: A Practical Introduction using BlueJ* [3]. This text starts with a "bare bones" overview of the essentials of the Java language, and then revisits concepts to develop deeper understanding. Such spiral approaches are well known to enhance learning [*] Further, this approach seemed to offer the opportunity for students to quickly see how their Alice knowledge mapped onto an industrial strength programming language.

The Java text is strongly based on using the BlueJ IDE. Like Alice, BlueJ is one of the few well-studied (see, e.g. [22]) and influential novice programming environments in current use. One feature that seems to make BlueJ an especially good choice for following Alice is its incorporation of an "object workbench." This workbench allows programmers to instantiate objects independent of client code. The methods for an object can be executed by right clicking on the object, and then selecting from a pop-up menu of the methods for that object. The similarity of this type of object manipulation in BlueJ to the experimentation and world layout actions performed in Alice seemed to offer a compelling conceptual stepping stone for the students.

Finally, the transition from Alice to Java was supported by the use of an instructor provided language "lexicon." This lexicon showed how each Alice construct was realized Java, and was intended to help ease the transition to the use of syntax.

The success of this transition was limited. The weaker students were intimidated by the textual language and syntax, and seemed to have a difficult time seeing how the Java code and the Alice code related. Lab assignments in which students were led through writing a Java program from an Alice program using the lexicon did not seem to help. Even the stronger students reported "syntax overload." To a certain degree we believe that this was due to the inherent organization of the Alice IDE. Object declarations, state variables and methods are all graphically organized on the screen with their own panes. In Java, many students were confused about the overall organization of the code. This approach also suffered from the timing of the introduction of user-defined variables. Because this topic was covered at the end of the Alice portion of the course, many students were still grappling with it as they undertook the transition to Java. An earlier or later introduction would have served much better. Finally, the ability to instantiate objects on the BlueJ workbench may have hurt more than it helped. Students expected this to be like the creation of objects in their Alice worlds: you create objects in the IDE and then write code to manipulate those objects. In BlueJ, however, the workbench is only for testing. Objects manipulated in code must be created the usual way, by being instantiated with **`new()`**.

In the Spring '06 semester, the pedagogical approach was dramatically altered. The course was again taught half in Alice and half in an industrial strength language, but in this case C++ (the change in language being mostly the result of departmental issues). Another difference is that the language coverage was interleaved with each topic introduced in Alice and then transitioned to C++. The hope was that such coverage would exploit the educational concept of spacing and give students the time to absorb the ideas of program translation and syntax. An IDE was not employed this semester; instead, the students used Emacs/g++ on a Unix system. To help scaffold student learning of C++ syntax, the web based system TuringsCraft's CodeLab was used (see **`turingscraft.com`**). Finally, this version of the course employed an early introduction to programmer-defined mutable variables along with the standard control constructs while the discussion of object-oriented programming was largely delayed. The idea was to avoid problematic advanced concepts in object-oriented programming, and give students increased exposure to variables and their use. A language lexicon was again employed.

In this version of the course, many of the transition problems persisted or were worse. Some of the better students were resistant to switching back to Alice once they had made the initial transition to C++. They questioned, "What was the point?" Weaker students lost confidence sooner, with the earlier exposure to the HLL and variables.

Finally, in general, in both semesters when our students transitioned to either HLL we noted that too many exhibited one of the following behaviors:

- Many students paid very little attention to syntax, often thinking it was acceptable to hand in work that did not compile (or that they had never attempted to compile in the first place). Students did not recognize precision of expression as an important aspect in computer science.
- Many students became discouraged when their programs did not compile and they concluded that they were inadequate programmers, even though they were able to program in Alice.

This was exacerbated by the perception of many of our students of Alice as a storytelling environment aimed at a younger audience. When HLL programming proved more challenging than Alice, they concluded that their success in Alice had not been 'real programming' but rather just fooling with a toy environment designed for a younger audience.

4. CONCLUSIONS

Alice is one of the most well known microworlds currently helping to draw generally underrepresented groups of people into the field of computer science. While Alice has been shown to increase confidence and retention at the university level, our experiences at Tufts have demonstrated some pedagogical pitfalls to the approach. The object model in Alice can easily lead to misconceptions, and although the lack of syntax errors can raise students' confidence while programming in Alice, it can be detrimental when these same students transition to C++ or Java.

Electronic Arts Inc. is currently working with the creators of Alice at Carnegie Mellon University to produce a new version of Alice that will incorporate characters from their popular game, The Sims. This is expected to have a huge impact on the popularity and accessibility of the Alice programming environment, as well as on its future development. The new version will make it clearer that instructors need to look beyond the excitement of teaching objects in a 3-D environment and be sure to carefully consider their instructional methods.

5. REFERENCES

[1] Alice, 2006. Online. Internet. Sept. 4, 2006. Available WWW: http://www.alice.org

[2] Barker, L. Factors in Participation of Women and Minorities in Computer Science by Type of Institution, 2003.

[3] Barnes, D. and M. Kölling, Objects First with Java: A practical introduction using BlueJ. Prentice Hall. 2004.

[4] Carlick, A. and F. Biley, Thoughts on the therapeutic use of narrative in the promotion of coping in cancer care. European Journal of Cancer Care. 13(4), September 2004, pp. 308-317.

[5] Cohoon, J., Department differences can point the way to improving female retention in computer science, SIGCSE Technical Symposium 1999, p. 198-202.

[6] Cooper, S., W. Dann, and R. Pausch, Alice: A 3-d tool for introductory programming concepts. Journal of Computing Sciences. 15(5), 2000, p. 108-117.

[7] Cooper, S., Private communication, Aug. 13, 2006.

[8] Cooper, S., W. Dann, and R. Pausch, Teaching Objects-first In Introductory Computer Science, SIGCSE Technical Symposium 2003, p. 191 – 195.

[9] Dann, W., Cooper, S. and Pausch, R. Learning to Program with Alice. Prentice Hall. 2006.

[10] Doran, G. and N. Downing-Hansen, Constructions of Mexican American family grief after the death of a child: an exploratory study. Cultural Diversity & Ethnic Minority Psychology. 12(2), Apr. 2006, p. 199-211.

[11] Dijkstra, E., On the Cruelty of Really Teaching Computer Science. The Communications of the ACM, 32(12), Dec. 1989, p. 1398-1404.

[12] Greenfoot, 2006. Online. Internet. Sept. 8, 2006. Available WWW: http://www.greenfoot.org

[13] Henriksen, P., and M. Kölling, 2004:greenfoot: combining object visualization with interaction. OOPSLA 2004, p. 73-82.

[14] Jeroo, 2006. Online. Internet. Sept. 8, 2006. Available WWW: http://www.jeroo.org

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

[16] Klawe, Maria. Girls, Boys and Computers. ACM SIGCSE Bulletin, 34(2), June 2002, p. 16-17.

[17] Koenig, J., and C. Zorn, Using Storytelling as an Approach to Teaching and Learning With Diverse Students. Journal of Nursing Education. 41(9), September 2002, p. 393-399.

[18] Margolis, J. and A. Fisher. Unlocking the Clubhouse: Women in Computing. MIT Press. 2002.

[19] Moskal, B., D. Lurie, and S. Cooper, Evaluating the Effectiveness of a New Instructional Approach, SIGCSE Technical Symposium 2004, p. 74-79.

[20] Naps, T. (chair) Evaluating the educational impact of visualization. A working group report of ITiCSE. Thessaloniki, Greece, Pages: 124 – 136, 2003.

[21] Pattis, R. Karel the Robot. New York: John Wiley & Sons, 1981.

[22] Ragonis, N. and Ben-Ari, M. 2005. On understanding the statics and dynamics of object-oriented programs. SIGCSE Technical Symposium 2005, p. 226-230.

[23] Shashani, L. Gender differences in computer attitudes and use among college students Journal of Educational Computing Research. 1997.

[24] Verhallen, M., et. al. The promise of multimedia stories for kindergarten children at risk. Journal of Educational Psychology. 98(2), May 2006, 410-419.

[25] Cooper, S.. W. Dann, and R. Pausch. Using Animated 3D Graphics to Prepare Novices for CS1. Computer Science Education. 13(1), 2003.

[26] Dann, W., S. Cooper, and R. Pausch, Making the Connection: Programming with Animated Small World. ITiCSE 2000, p. 41-44.