Alice, Greenfoot, and Scratch – A Discussion

IAN UTTING
University of Kent
STEPHEN COOPER
Stanford University
MICHAEL KÖLLING
University of Kent
JOHN MALONEY and MITCHEL RESNICK
Massachusetts Institute of Technology

This article distills a discussion about the goals, mechanisms, and effects of three environments which aim to support the acquisition and development of computing concepts (problem solving and programming) in pre-University and non-technical students: Alice, Greenfoot, and Scratch. The conversation started in a special session on the topic at the 2010 ACM SIGCSE Symposium on Computer Science Education and continued during the creation of the resulting Special Issue of the ACM Transactions on Computing Education.

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

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Alice, Greenfoot, Scratch, visual programming language, programming language, programming environment

ACM Reference Format:
Utting, I., Cooper, S., Kölling, M., Maloney, J., and Resnick, M. 2010. Alice, greenfoot and scratch – A discussion. ACM Trans. Comput. Educ. 10, 4, Article 17 (November 2010), 11 pages. DOI = 10.1145/1868358.1868364. http://doi.acm.org/10.1145/1868358.1868364.

1. INTRODUCTION

The articles presented elsewhere in this issue raise a number of interesting topics which cross the boundaries between the three systems, both in terms

Authors' addresses: I. Utting and M. Kölling, School of Computing, University of Kent, Canterbury, UK; email: I.A.Utting@kent.ac.uk, M.Kolling@kent.ac.uk; S. Cooper, Computer Science Department, Stanford University, Stanford, CA; email: coopers@acm.org; J. Maloney and M. Resnick, Media Laboratory, Massachusetts Institute of Technology, Cambridge, MA; email: jmaloney@media.mit.edu, mres@media.mit.edu.

of what the systems are trying to do, and of their audiences. In this article, we pick up some of those issues to take a broader look at them between the authors. These discussions started in the Q&A of the special session at the ACM Symposium on Computer Science Education in 2010 [Fincher et al. 2010] and continued in face-to-face meetings after the special session, and in ongoing electronic communication during the preparation of this issue.

The questions below are posed by the first author, and the authors of particular responses are indicated by their initials: [SC], [MK], [JM] and [MR].

2. AGE-GROUP ISSUES

You each talk about your systems as being aimed at particular age groups: CS1/pre-CS1 for Alice, 8-16 years old for Scratch, and 14+ years for Greenfoot. Is that because you feel that these ages are a developmental "sweet spot" for learning to program? Or is it a corollary of design choices you made for other reasons? Do you think there is such a sweet spot?

[MK] I don't think there's a sweet spot for programming concepts in general. It's a bit like learning natural languages: the earlier the better. But I think there are boundaries in cognitive development for specific concepts and technologies. In Greenfoot, for example, we use standard Java syntax. That imposes a fairly hard lower maturity boundary for the kids. I experimented with 10 year olds, and they clearly have trouble with syntax errors. They get the brackets out of synch, and they can't fix it. They can understand the underlying programming concepts quite well, but they cannot deal with this technology. The necessary understanding of syntactic abstraction seems to develop a bit after that. We haven't done a study, but our experience seems to suggest that bright 13 year olds can handle it and 10 year olds generally can't.

So, yes, our age limit is a side effect of the decision that we wanted to support teaching close to existing, traditional programming curricula with a traditional language.

[JM+MR] In developing Scratch, we wanted to "lower the floor" for programming, so that children could get started earlier. In our view, learning to program is somewhat like learning to write. In both cases, children should start as soon as they are interested. It makes sense for children to start with simple forms of expression, and gradually learn more subtle and sophisticated ways of expressing themselves over time.

[SC] I don't see a specific "sweet spot" for Alice per se. Certainly, the Alice community has a wider view than we the developers do about (a) how to use Alice and (b) where Alice is best used. We have in general been much less interested in using Alice as a "Power point on steroids" than some educators. Likewise, we have been less interested in teaching animation techniques per se although there are many folks using Alice specifically to teach animation and as a 3D game engine.

Rather, our focus has been on using the context of animation to provide an exciting and engaging environment in which the student wants to learn problem solving and programming as an implementation of a design/algorithm for

solving a problem. Thus, our belief is that Alice is best suited for those students who are already coming across problems in other STEM[1] classes (word problems in Algebra 1 in the 7th and 8th grades immediately spring to mind), who have sufficient background to be able to tackle a more rigorous approach toward problem solving.

Syntactically, there is no actual free-text typing (much as in Scratch), so that clearly younger children can mechanically handle Alice. For example, I have run two-hour sessions using Alice with Girl Scout troops the summer after the girls completed 4th grade (so the girls were likely only 10 years of age). That being said, the focus was not on sophisticated problem solving, but rather on the mechanics of how to use Alice to tell a story. At the other end of the spectrum, Alice is frequently taught to college students (most commonly in non-majors courses or in pre-CS1 courses, though we do see several folks using Alice in CS1).

3. THE GENDER AGENDA

There is considerable concern about the gender imbalance in CS, especially the under-representation of women at the more "technical" end of the discipline. Do you believe that systems such as Alice, Greenfoot, and Scratch can help here, and especially is there anything that you think your particular system brings to the table?

[MK] We haven't got any evidence for this, but we certainly hope so, and it's one of our goals. However, rather than aim at typically "girly" topics (which are often in danger of becoming a bit condescending, I feel), we hope to help with this by emphasizing individuality and empowering users. Experience does show that girls are, on average, interested in different things than boys. They are often more socially oriented, more interested in context, making a difference in society, while boys are often happy just to play for play's sake. However, this is obviously not true for every boy or girl—it's a generalization, and as such fraught with problems.

We are trying to get learners quickly into a position where they can take control and bring in their own interests into the project. This way, they are able to relate the programming work to what motivates them in their existing life. Greenfoot, however, does not automatically do that—it's an enabler. It still needs a good teacher presenting good challenges and allowing them to run with it.

[JM + MR] The National Center for Women & Information Technology [NCWIT 2008], in a case study about Scratch, calls Scratch a "promising practice" for increasing gender diversity in IT. The NCWIT study notes that Scratch "uses hands-on, active learning; it is visually appealing; it allows users to express their own creativity and to build on their own experiences; it gives immediate, understandable feedback; and it allows users to avoid syntax errors without focusing on minutiae, freeing them to focus on processes and concepts."

There is preliminary evidence that Scratch can help attract a broader range of students to computer science. For example, the introduction of Scratch to

[1]Science, Technology, Engineering, and Mathematics.

the introductory computer-science course at Harvard led to a sharp reduction in the number of students dropping the course or receiving a failing grade, and a marked increase in the retention of female students [Malan and Leitner 2007].

[SC] We published three studies involving the use of Alice. The first [Moskal et al. 2004] showed significant improvement for "at risk" computing majors with respect to retention. Exposure to Alice prior to or concurrent with CS1 led to improved performance in CS1 (approximately one full letter grade improvement) and significantly higher retention into CS2 (47% to 88%) for these students. Women were not separated out as there were not enough women participating in the study. The latter two studies [Hutchinson et al. 2006, 2008] looked at results of Alice instruction at the community college level. With a large enough student population, positive results were seen with female students. While student attitudes and performance were higher, it is otherwise hard to measure what "success" meant, as many of the community college students took Alice instruction as part of a computer literacy course, and many others were other flavors of computing majors (IS, IT, Computer Systems Technology, Computer Graphics, Animation, etc.)

Others have reported positive results with Alice usage, though the focus has often been with underrepresented minorities rather than with women. For example, Gersting at the University of Hawaii worked with the Hawaiian Studies department to create a first-year course combining Hawaiian history/culture with programming in Alice. Students learned enough programming to animate stories of Hawaiian mythology. We believe that the strong storytelling aspect to Alice helps to make it appealing to many underrepresented groups, especially those for which an oral tradition or a tradition of storytelling is an important part of their culture.

4. TINKERING, OR DISPLACEMENT ACTIVITIES?

The ability to "tinker" can be a double-edged sword. While we want students to experiment and explore environments and possibilities, we want to guide their tinkering into useful channels rather than have them stall at changing colours/images. Do you see this as a problem in general or with your environment in particular?

[JM + MR] We see "tinkering" as a style of interaction that can be useful for people at all levels of expertise (not just for beginners). In our view, a "bottom-up" style (tinkering) is just as legitimate as a "top-down" style (planning). One style might be better than the other in some particular contexts, but neither is inherently better. People don't necessarily change from "tinkerers" to "planners" as they gain more programming skills and expertise. Some of the best "hackers" program in a tinkering style.

Of course, it is a problem if people simply mess around and never focus toward any goal. But that's shouldn't be confused with bottom-up tinkering. It is just "bottom" (with no up). We need to make sure that children, as they start tinkering, learn how to build upon their explorations.

[SC] In Alice, there is a good and a bad with tinkering. Because Alice worlds (and those of Scratch and Greenfoot) are so visual, students can and do directly map instructions to the result of what they see onscreen. This is a real power of program visualization. When an animation doesn't work correctly, the student almost always knows which are the offending sections of code that drive, and so must have led to, the undesired behavior. So, much of the early frustration associated with debugging gets minimized, as students are able to make progress on their own, as they try to solve specific problems. The bad part is that if the instructor doesn't design labs and assignments carefully, students can depend on trial and error and tinkering as a crutch. And students do need to develop other successful strategies toward problem solving.

5. COMMON THEMES

In reading the articles and hearing about the design goals and processes, it is really interesting how some common themes emerge in all of them. Which of these do you think are particularly important?

[MK] Engaging and empowering the user is the most important common theme for me.

[JM] Agreed. All three systems strive to engage the user by allowing them to write programs about things that connect with their interests (stories, games, simulations, etc.) in contrast to more conventional programming instruction where examples might be things like generating prime numbers or sorting a list of numbers. Other similarities: all three systems also focus on hands-on problem solving, all are intended as a way to introduce programming before (or perhaps in) CS1, all strive to encourage and retain "at risk" students, and all reach out to a wider audience.

[SC] I read John's comment here, and first thought that it was one of mine, I am in such strong agreement. Steve Cunningham and I [Cooper and Cunningham 2010] just wrote an article about the importance of context. Connecting students with their interests should not, and indeed cannot, be under-emphasized if we want to appeal to today's students.

[JM] All three systems reify objects so that the result of command execution is visible as the position, size, rotation, and other visible state of the object changes. All three systems provide high-level commands like move() and hide low-level details such as the interaction loop, display redraw and graphics primitives. This approach to graphics harks back to Logo's turtle.

[MK] Yes, and that gets into the interesting question of where the right abstraction level is to give to students. You want something that's high level enough to do something useful, but low-level enough to allow flexibility, and to allow composing your own actions. The details of the goals are a little different here between the systems. I guess, in Scratch, the main focus is on easily achieving something interesting. In Greenfoot, we also have user-defined methods—users get to build their own commands by composing existing ones. This is one of the very important lessons that we want kids to understand: You needn't only take what you're given, you can make your own! And they look just like the ones that were there. You are not only a user of language, you are

also a designer. This power of abstraction is one of the most fascinating things in computing.

Alice 3 is currently confronted with exactly this question. In integrating the Sims characters into the existing Alice universe, you have to somehow merge the more high-level actions that the Sims provide with the lower-level, more generic capabilities of the traditional Alice characters.

[SC] This issue of what is the right level of abstraction (versus concreteness, perhaps) is a hard one to answer. I think that each of Alice, Greenfoot, and Scratch answer the question similarly. I find it interesting to note that Brooks, in No Silver Bullet, considers this one of the accidental (versus essential) advances of software engineering. Brooks refers specifically to the added layers of abstraction that high-level languages provide, but I consider that situation quite analogous to what we are discussing here. And while the right levels of primitives, so to speak, may be an accidental advance for the professional software engineer, I think it is absolutely essential for the novice as they learn to program.

[JM] All three systems strive to eliminate gratuitous difficulties, although in different ways. Alice and Scratch remove syntax issues. Greenfoot provides the "main" method and a tiny yet powerful class library that shields the user from the overwhelming large and complex set of Java APIs. The Actor/World framework makes it possible to start by writing short, self-contained methods (e.g., act()) without having to build a Java application from scratch.

6. DESIGN PROCESS

Clearly, the three tools we're discussing here are part of a design tradition which goes back to Papert's Turtle Graphics and Pattis' Karel the Robot. To what extent were these systems (and others like them) an explicit part of your design process?

[MK] I find the design process very interesting. Sally and Ian, in their introduction, talk about craft and the embedded design. This view certainly fits very well with my experience. In the article, written in retrospect, it might appear as if all the design ideas were clearly worked out in advance, and then the implementation built to fit them. That is how it is represented, but that's just for convenience sake, for structuring a readable narrative. Read as a time line it is, of course, a lie. Some design ideas did indeed exist upfront, were discussed and written down early. But others didn't at all. This doesn't mean that the artifacts in those areas are random—on the contrary. I usually have a very clear idea what I want and what I like, even if I haven't quite formulated explicitly why. The design seems to progress by gut feeling. But it is gut feeling based on facts and experience. We often analyze why we like certain things only afterwards, and the neatly written down design goal only emerges then. It did exist all along, however, in more implicit form.

It is very interesting how the different systems, their design often being driven by gut feeling (at least in my case) end up with very similar ideas, mechanisms, and solutions. The incarnations of these are necessarily different, but the spirit, the goal, the embedded philosophy is often the same.

One of the most fundamental ideas from those early systems, that is still equally important in our systems, is the visualization not only of a computation's result, but of its ongoing execution. Steve describes that very well in his article when he talks about animating state changes.

[JM] I completely agree! In the case of Scratch, we also had many people debating the design decisions. We had a core team of four, all of whom had many years of experience with earlier programming systems (Logo, StarLogo, Etoys, Crickets, Mindstorms, . . .). Often, one person would notice design flaws that others had missed. But even with this experience, we often found problems during testing leading to more design discussion and further iteration.

[SC] I too, am in strong agreement with Michael's observation. I think that all three tools benefit from the work that has gone before them, as John notes. I think that more of the engineering/technical design issues tended to just happen (at least in Alice's case) rather than the explicit plan for them. We all recognized the importance of program visualization, and the need for the student to be the author, rather than a passive (or even interactive) observer of the instructor's animation.

7. EXPOSURE OF CONCEPTS

A crucial part of developing tools for non-expert audiences is in choosing which of the underlying concepts to reveal and which to conceal. What drove you to make the choices you did for your systems?

[MK] All three systems aim at removing or hiding accidental complexities, letting users get on with the job, and making them work with fundamental constructs. One interesting detail is the degree of directed learning embodied in the systems' interactions: Scratch tries to get out of the way completely, and lets users learn entirely through play. Greenfoot plays a little more of a teacher role: It makes you do some things that wouldn't really be necessary, because we think it's good for you. Take compilation, for example. Scratch completely hides it and gets out of your way. In Greenfoot, we have an explicit "Compile" button. It would have been trivial to automate this and hide it away, but we decided that compilation is an important concept that we want students to know about, so we exposed it. I guess Alice is somewhere in the middle. This very directly reflects the different stages of learning at the different age groups and shows, in fact, a common philosophy: Let them play first, let them achieve something, let them be creative, and then sneak the explanations in about what is going on when you're working with the system. That's why a sequence from Scratch to Greenfoot or from Alice to Greenfoot can work so well.

[JM] I totally agree with "let them play first, let them achieve something, let them be creative. . . "

[JM] Interesting that exposing the "Compile" button was a deliberate design choice—and one that definitely makes sense for Greenfoot.

[SC] In Alice 2, there isn't any compilation, as Alice is interpreted. We got, and still get, many requests to see the underlying code that is generated in Alice 2. Folks don't believe us that there really isn't any! Of course, this made the decision to only include a "Run" button somewhat more simple.

[MK] Another example where the same relationship comes into play is in dealing with errors. Scratch with its defaults and fail-soft strategy aims at always doing the right thing, to get things to work, even if sometimes users may not fully understand why. Greenfoot aims at making users understand the details of how things need to be to get them to work. And that's exactly the right order to do things in.

[MK] It's a learning progression: tinker first, then understand.

[JM] Since I wouldn't expect most of our younger Scratch users to go directly from Scratch to Java, it seems good to hide things like compilation and error messages. But it makes sense for Greenfoot to introduce those things as a stepping stone to programming in a professional Java IDE. I think Scratch → Greenfoot → Eclipse is a good path for Scratch users who want to get deeper into programming. And many Scratch users may not go on to learn Java or major in computer science, just as most of us who took music lessons did not grow up to become professional musicians.

[SC] This is something I really like about Scratch (and its fail-soft strategy). In a sense, I think this had been one of the original goals of Alice; I don't list in my article largely because it became challenging to realize in practice (3D is somewhat of a culprit here). I do agree that the transitions Scratch → Greenfoot, Alice → Greenfoot, and Scratch → Alice → Greenfoot wind up making a great deal of sense in terms of what gets exposed to the student.

8. ON THE VALUE OF SIMPLICITY

No matter what choices you make about what to reveal and what to conceal, there's always pressure from users (especially the more advanced ones at the periphery of your target audience) for more and more complex functionality. Of course, this tension is not restricted to systems for beginners; it's also visible in the evolutionary design of popular languages like C++ and Java. How have you dealt with it?

[MK] All three articles, reading between the lines, show this tension between scope of functionality and flexibility on one hand, and simplicity on the other. This is one of the most fundamental issues for every designer, in every design. For Alice and Scratch, one example of this is in the number and nature of blocks provided. John describes that very well in his article in the discussion of the multifunction blocks. In Greenfoot, the same exists in the decision about the Greenfoot API methods. We all try to find the right balance between being welcoming (small and simple) and giving power to the more advanced users. That's hard. But I really think it's one of the most important shared characteristics of these three systems: the fact that they value simplicity very highly and are not getting sucked into feature escalation. Once you have a real user base, that is very hard. The pressure to add features is always there, and it's strong. You have to resist it. You often have to say "Yes, that's a nice idea, but we're not gonna do it. Simplicity still is more important." Often, the right solution is to not do something. And that's a hard sell to the users who really want a feature.

[JM] Everyone on the Scratch design team wanted the system to be simple and were willing to sacrifice features to keep it that way. I sometimes think

that more Scratch features were tried and discarded than the ones that made it into the final version.

[SC] Alice gets hit pretty hard with requests for specific features. Alice's object orientation is wrong, there's no collision detection, there's no underlying Java code, saving Alice worlds as Web pages doesn't allow full interactivity, etc. Alice 3 will partially give in to these demands by providing an API for developers to interact with the underlying Alice system. I expect that most of the feature requesters will not be so eager or able (time/ability/etc.) to go ahead and actually add the missing Alice features, but it will be interesting to see how folks respond. In Alice 2, the answer was generally "no", and I think that that is probably the better way to go. Simple really is beautiful.

9. CONTRASTS

It's been remarkably hard to get you all to disagree to any significant extent about the goals and values of your systems. Where do you see the contrasts and differences?

[JM] OOP and Java: Scratch sprites are objects that own state and behavior, but they can only be copied. Scratch has no classes or inheritance. Greenfoot has the Java object model. Alice 2 is somewhere in between, with only one level of subclassing; Alice 3 generalizes that. This reflects differing goals: Scratch makes no attempt to teach OOP, whereas both Alice and Greenfoot introduce OOP in preparation for class-based languages such as Java. Similarly, the three systems differ in their relationship to Java from Scratch (no relation at all) to Alice 2 (showing scripts in Java) to Alice 3 (ability to write scripts in Java) to Greenfoot (it **is** Java).

[JM] Animation: Alice makes heavy use of transition animations. Every change of visible state (e.g., position) is animated with a default time of one second. In contrast, Scratch has only one time-based animation command (glide); other animations must be constructed using loops.

[JM] Large-scale computation: Greenfoot can be used for big computational problems such as public-key cryptography, analysis of larger data sets, or pixel manipulation (i.e., media computation). Scratch is too slow for such problems. I'm not sure about Alice, but I'm guessing it's closer to Scratch than Greenfoot. This means that Scratch is generally not suitable for a CS1 course, although it is sometimes used as a warmup in CS1 before moving on to C, Python, or Java.

[SC] Alice is certainly closer to Scratch here. It's just too slow to handle many large data problems. It does get used in CS1, but that's been its ceiling.

[MK] Yes, that's one of the benefits of Greenfoot, where your initial higher learning investment pays off. Since it is standard Java, running on the standard Java VM, you can do pretty much anything you like. We are seeing people using Greenfoot in AI and agent programming courses at universities. They use it because they get the graphics for free, and the behavior of the actors can be as sophisticated as you like.

[JM] 3D versus 2D: Alice is the only 3D system. 3D is great for some types of projects (stories, games, virtual worlds). Scratch and Greenfoot also support stories and games, but require more effort to create virtual worlds larger than

the Stage/World. On the other hand, some projects, such as paint programs, are easier in 2D. It's much easier to draw or import 2D images than it is create or find 3D models. I have the sense (but feel free to argue, Steve) that the 2D model supports a wider range of project types than 3D, such as certain simulations, interactive art, paint programs, UI prototypes, hand-drawn animation (Japanese anime style characters are big on the Scratch Web site).

[SC] I do agree, especially with respect to games. (3D models are generally not freely available, and there are so many different file formats—and writing converters is expensive, time-wise, that students are generally stuck with the models that ship with Alice.) In fact, many 3D games are "really" 2D. I've been surprised that there have been many student Alice worlds built as 2D worlds (where all of the images get imported as billboards, and depth is simply eliminated). Certainly dealing with one rotational angle is much easier than dealing with 3. Though, stories are often more realistic in 3D, and when interactive worlds are built with 3D in mind (a flight simulator comes immediately to mind), 3D does work well.

[MK] Regarding the stories: that's actually surprisingly hard in Greenfoot. Greenfoot's framework is more geared towards a reactive system—either reacting to user input or objects reacting to each other. Straight scripting of fixed animations goes somewhat against the flow. That's much easier in Scratch and Alice. I have the same feeling about the 2D/3D effect, though.

[SC] I think that Alice and Scratch wind up quite close, with respect to stories and storytelling. My hope with respect to storytelling is that there might be some sort of hook with it into middle schools in the U.S. (and abroad, though schools for students aged 10-14 are named differently outside the U.S.). Specifically, there is a potential appeal to non-STEM instruction (though I'm not sure how to provide professional development for the teachers at scale, who have undergraduate degrees outside of STEM, and often possess an aversion to STEM).

I think the 2D/3D difference is a slightly different way to distinguish Scratch and Greenfoot from Alice. I see it as more of a complexity issue.

REFERENCES

COOPER, S. AND CUNNINGHAM, S. 2010. Teaching computer science in context. *Inroads 1*, 1, 5–8.

FINCHER, S., COOPER, S., KÖLLING, M., AND MALONEY, J. 2010. Comparing alice, greenfoot & scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. 192–193.

HUTCHINSON, A., MOSKAL, B., DANN, W., AND COOPER, S. 2006. The Alice curriculum and its impact on women in programming courses. In *Proceedings of the Annual Meeting of the American Society for Engineering Education (ASEE'06)*.

HUTCHINSON, A., MOSKAL, B., DANN, W., COOPER, S., AND NAVIDI, W. 2008. The Alice curricular approach: A community college intervention in introductory programming courses. In *Innovations 2008*, International Network for Engineering Education Research, 157–176.

MALAN, D. J. AND LEITNER, H. H. 2007. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)*. 223–227.

MOSKAL, B., LURIE, D., AND COOPER, S. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. 75–79.

NCWIT (NATIONAL CENTER FOR WOMEN AND INFORMATION TECHNOLOGY). 2008. Snap, create, and share with scratch (case study 5). http://www.ncwit.org/images/practicefiles/SnapCreateSharewithScratch_EngagingWayIntroduceComputing.pdf.

Received August 2010; accepted September 2010