ALICE: A 3-D TOOL FOR INTRODUCTORY PROGRAMMING CONCEPTS

Stephen Cooper
Computer Science Dept.
Saint Joseph's University
Philadelphia, PA 19131
scooper@sju.edu

Wanda Dann
Computer Science Dept.
Ithaca College
Ithaca, NY 14850
wpdann@ithaca.edu

Randy Pausch
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
pausch@cs.cmu.edu

ABSTRACT

In learning to program, many students struggle with developing algorithms, figuring out how to apply problem solving techniques in their programs, and with how to use common programming constructs. In this paper, we present a new tool that provides a possible approach to actively engage students in increasing their knowledge and skills in these areas. The tool is Alice, a 3-D interactive animation environment.

1 INTRODUCTION

Experienced teachers of CS1 and other introductory programming courses are often painfully aware of the wide spectrum of backgrounds students bring with them upon entry to the course. The dilemma is that if we teach at the level of the most poorly prepared students, the top-level students are bored and may become highly frustrated at the lack of progress and challenge. If we teach at the level of the best-prepared students, the students at the lower level become so confused that they are likely to drop the course. And finally, if we teach at the level of the students in the middle, the needs of students at both upper and lower levels are not being addressed.

We could debate at length the relative merits of each approach as well as possible teaching methodologies (such as cooperative learning). In this paper, however, we focus on the preliminary stage -- preparing students for learning programming concepts. While students at the top-level could skip this preliminary stage, good students who have weak backgrounds could be given an opportunity to fill in the gaps.

1.1 What Is Missing?

First, we need to think about what might be missing in the backgrounds of students in the lower range of the spectrum, as described above. It is easy to say that they “do not know how to solve problems.” But this is too simplistic. These students have likely achieved a certain level of competency in mathematics and problem solving–at least through courses such as algebra and pre-calculus. We contend that these students are not strong problem solvers in the particular ways that are necessary for success in computer programming. We believe students need to develop an increased level of competency in how to design an algorithm for solving a problem and how to use specific programming statements in accomplishing that goal. As stated by Soloway[1], the real difficulty for novice programmers lies in "putting the pieces together", i.e. in figuring out what constructs to use and how to coordinate those constructs. Without this background, many students are not prepared to solve problems in the way they need to be solved in a computer program.

One other serious problem that plagues many students is that learning to write, test, and debug programs requires that the student learn why and how the program (computer) solves the problem. We have observed that many students are unable to visualize the steps of the execution of the program. As a result, they cannot figure out what went wrong when things do not work. In imperative languages, a trace of the program with memory snapshots can be used in an effort to assist students in figuring out what is going on. However, using traces may actually add to some students' confusion! We believe the source of confusion in figuring out what went wrong, in all but the most trivial code, is an inadequate understanding of the program's state.

What we want to do is provide an environment in which students can learn the kinds of problem solving strategies and the necessary concepts and skills needed for creating computer programs. Animation of program execution can be used to help the student "put the pieces together". Visualization is one approach to assisting the learner in finding out what task each piece can be expected to perform and how the pieces work together to perform the overall task of solving the problem at hand. For this purpose, we are using a new 3-D interactive animation tool, *Alice*.

1.2 Previous Work

The use of animation to show program execution is not a new idea[2, 3, 4]. Among the many researchers who argue for visualization, Shu[5] presents a particularly strong case. He considers programming to require both parts of the brain, and focuses on the need to involve the artistic half – expressing the need to involve pictures in the process.

A popular attempt to provide visualization has been the use of sophisticated graphics packages/libraries (often developed by the instructor or a textbook author). These packages have introduced visual aspects into programming. However, our experiences in working with such packages have been that they were fairly complex, and often difficult for beginners. They also tended to rely on an underlying programming language (such as C or scheme) that the students needed to learn and master.

Attempts in algorithm animation, e.g. XTANGO[6] and BALSA[7], have been developed with the idea of incorporating visualization into the learning process. Most of these attempts have been targeted at CS1 students and at students in higher levels.

[1] Soloway, E.M. Learning to Program = Learning to Construct Mechanisms and Explanations. *Communications of the ACM*, 29 (1986), 850-858.

[2] Naps, T.L. Chair, Working Group on Visualization. An Overview of Visualization: its Use and Design. in *Proceedings of the Conference on Integrating Technology into Computer Science Education*. Barcelona, Spain,(June 1996), 192-200.

[3] Pattis, R., *Karel the Robot*. New York: John Wiley & Sons, 1981. Rodger, S.H. Integrating Animations Into Courses. in *Proceedings of the Conference on Integrating Technology into Computer Science Education*, Barcelona, Spain (June 1996) 72-74.

[4] Stasko, J.T., Dominque, J., Brown, M. and Price, B., eds. *Software Visualization, Programming as a Multimedia Experience*. Cambridge:MIT Press, 1998.

[5] Shu, N.C., *Visual Programming*. New York: Van Nostrand Reinhold Co, 1988.

[6] Stasko, J.T. Animating Algorithms with XTANGO. *SIGACT News*, 23 (1992), 67-71.

One of the most successful program simulation software packages, used for program visualization, has been *Karel, The Robot* [8]. The Karel software has been used as a gentle introduction to programming at both high school and college levels for many years. Karel is a wonderful tool for setting the stage, preparing students for success in learning Pascal. Over the last decade, curricula for CS1 have made a transition to C and then to object oriented languages. In response, Karel has undergone several updates, the latest being *Karel++* [9], a C++ like version that brought Karel into the object-oriented age. However, Karel++ introduces a significant level of code complexity (and corresponding increase in the learning curve) over that required in Karel. We believe that Alice can be used to follow Karel's tradition with a 3-D, animated environment where students can create their own virtual worlds. 3-D worlds are more realistic than their 2-D counterparts. These virtual worlds can be displayed on a web page (via a browser plug-in, similar to Macromedia Flash).

1.3 Instructional Experience

We used Alice as an instructional tool for two summer sessions at Ithaca College. High school students were enrolled in a special Summer College program. The goal of the course was to provide an opportunity for students to learn the fundamental concepts of programming and problem solving. Our observations of students working with Alice are the basis of viewpoints presented in this paper. The authors' textbook for programming in Alice (draft copy at www.ithaca.edu/wpdann/alice1298) is a reflection of experiences gained from working with these students. The true success of our approach will not be fully known until these students enter college and their performance can be compared with students who did not work with Alice. A first course in visual programming using Alice, will be offered in the fall '00 semester for full time college students. Discussions have begun about doing the same at St. Joseph's University.

2 WHAT IS ALICE?

Alice (http://www.alice.org) is a 3-D Interactive Graphics Programming Environment for Windows[10] built by the Stage 3 Research Group at Carnegie Mellon University under the direction of Randy Pausch. The goal of the Alice project is to make it easy for novices to develop interesting 3-D environments and to explore the new medium of interactive 3-D graphics. Alice is primarily a scripting and prototyping environment for 3-D object behavior.

3-D models of objects (e.g., animals and vehicles) populate a virtual world in Alice. Alice has an object oriented flavor. By writing simple scripts, Alice users can control object appearance and behavior. During script execution, objects respond to user input via mouse and keyboard. Each action is animated smoothly over a specified duration. (This replaces the traditional animation methodology, where the animator prepares many frames and then uses a frame animator to view a succession of frames in rapid sequence.) Alice is built on top of the programming language Python (http://www.python.org) and uses many of Python's features.

---

[7] Brown, M.H., *Algorithm Visualization*. Cambridge, MA: M.I.T. Press, 1988.

[8] Pattis, op. cit.

[9] Bergin, J., Stehlik, M., Roberts, J., and Pattis, R., *Karel++, A Gentle Introduction to the Art of Object-Oriented Programming*, New York: Wiley & Sons, 1997.

[10] Pausch, R. (head), Burnette, T, Capeheart, A.C. , Conway, M., Cosgrove, D. DeLine, R., Durbin,J., Gossweiler,R., Koga,S., White, J. Alice: Rapid Prototyping System for Virtual Reality , *IEEE Computer Graphics and Applications,* May 1995.

Alice serves as a good programming language for the novice programmer. Students are immediately able to see how their animated programs run. The highly visual feedback allows the student to relate the program "piece" to the animation action. This leads to an understanding of the actual functioning of different programming language constructs. Below we describe many of the programming language constructs featured in Alice.

2.1 Actions (state transformers)

Alice provides several built-in action commands. In general, actions can be subdivided into two categories: those that tell an object to perform a motion and those that change the physical nature of an object. Motion commands include moving objects within the world (e.g. **Move**), rotating them about their 3-D axes (e.g. **Turn** and **Roll**), and pointing at other objects (e.g. **PointAt**). Commands that change the physical nature of objects include object destruction (**Destroy**), dynamic object creation (e.g., **AddObject**), object resizing (**Resize**), and making objects visible/invisible (e.g. **Hide** and **Show**).

While it is beyond the scope of this paper to discuss all of Alice's action commands (the commands listed above are a subset), we discuss the **Turn** action to illustrate details. Turning is allowed in 4 directions: Forward, Back, Right, and Left. In the Turn command, it is only necessary to specify which object is to be turned, the direction it is to be turned, and how much it is to be turned. Figure 1 illustrates turning along one of the rotational axes.

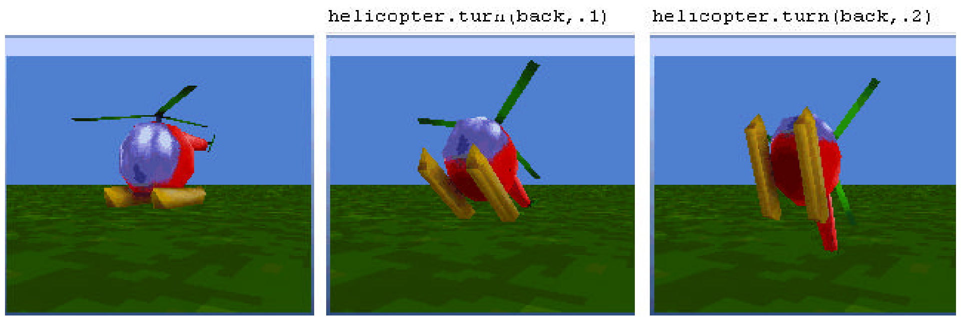

Figure 1. Rotating an object backwards

2.2 Named instructions

It is possible in Alice to name a sequence of instructions. The concept of a named instruction is similar to the procedure concept in many other programming languages (or a function that just performs side effects, not returning anything). While in traditional programming languages, it may not be clear why a group of statements should be blocked into a function/procedure, in Alice it makes intuitive sense. By collecting the 10-20 Move and Turn instructions it takes to make a bunny hop, and naming this entire sequence of instructions *Hop*, it becomes clear to the student that

```
DoInOrder(
   Hop,
   Hop )
```

causes the bunny to hop twice. Again, the animation aspect of Alice allows the student to immediately visualize the functionality of program constructs.

2.3 Functions

Functions in Alice are supported through the underlying Python language. In Alice, functions are primarily used in the implementation of recursion/looping, and in the implementation of interactions via events. Also, they can be useful in computation. For example, the *howmany* function, illustrated below, calculates how many turns a ball will make based on its diameter and the distance it will travel.

```
def   howmany (dist, diam):
  return dist / (3.14 * diam)
```

2.4 Decisions

As with functions, decisions are supported through the underlying Python language. Because of the visual feedback in Alice, students are able to see immediately the results of a decision statement. For example, in the statement:

```
if cat.DistanceTo(Fish) < 1.0:
   DoInOrder(
        cat.Move(Up, 0.5),
       cat.Move(Down, 0.5) )
```

if the distance between the cat and the Fish is less than one unit, the student sees the cat jump up and down.

2.5 Recursion/Looping

Alice provides support for repetition through the **Loop** instruction. For example, if Hop, a named instruction, has been previously defined then **Loop** (Hop, 5) causes the bunny to hop 5 times. The possibility exists for constructing the traditional while loop in Alice (through the use of the **Do** (action, **EachFrame**) construct). However, our preferred approach is to use generalized recursion through the **SetAlarm** command. While the concept of recursion may seem quite alarming to students, it really need not be. The following code snippet defines a recursive function named Chase. In the Chase function, the Fish moves towards the cat until it is within 2 distance units of the cat.

```
def Chase():
    if Fish.DistanceTo(cat) > 2:
        DoInOrder(
     Fish.PointAt (cat),
     Fish.Move (Forward, 1),
     Alice.SetAlarm (2, do(Chase) )     )
```

Simply put, this code checks whether the Fish is close enough to the cat. If not, the Fish moves 1 directional unit towards the cat, and the alarm is set to repeat the whole process. In our experience with students using Alice, we observed that students find the recursive action easy to understand. Having an explicit (and therefore *visible*) delay between the recursive calls seems to help students comprehend the action.

2.6 Events/Interactions

Alice provides support for event handling and for creating GUIs with control panels, list boxes, check boxes, and sliders. Events allow the user to interact with the animated world. Alice's approach is similar to the widgets used in Java, where instructions are issued to place the appropriate widgets onto a control panel (not a drag-and-drop interface as used in Visual Basic).

3 ISSUES

Designing a course that centers on Alice as the primary instructional tool presented a number of pedagogical issues. These issues can be organized into four major areas: graphics concepts, the notion of state, programming and programming language concerns, and event-driven programming. In this section, we discuss the questions that were asked, decisions that were made, and several concerns we still have in these areas. We remain open to constructive discussion and debate on these issues.

3.1 Graphics and Animation Concepts

A question that must be answered is how much do students need to know about 3-D graphics and animation. We found that Alice lowers the cognitive burden of creating 3-D interactive programs by providing built-in methods that support object positioning and motion. Thus, it was NOT necessary to spend many hours teaching students how to move and position an object within the world. Nonetheless, students must gain an understanding of the coordinate system and the spatial relationship of objects to one another. Each object in the world has its own egocentric orientation, as illustrated in Figure 2. (Note the *right* and *left* orientation is from the perspective of the helicopter object.)

While this seems simple enough, it becomes more complex in worlds containing several objects. As seen in Figure 3, each object has its own orientation that may or may not be aligned with another object in the same scene. As objects move around in the world, their spatial orientation with respect to other objects may change. Alice provides good support for aligning the orientation of objects and allows an object to be positioned "AsSeenBy" another object within the world, or to be "placed" on top of another object.

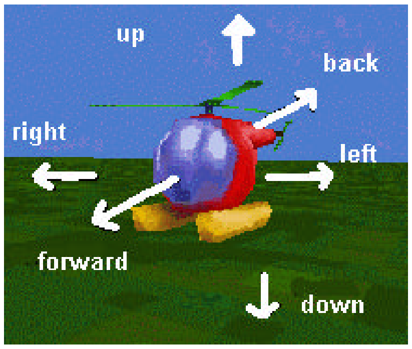

Figure 2. Six Degree Orientation

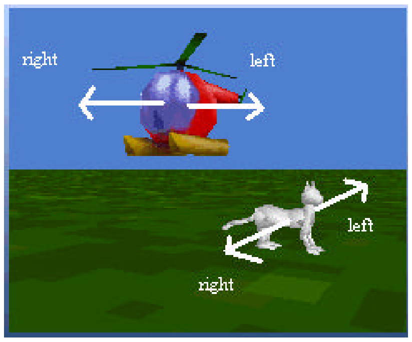

Figure 3. Different Orientations

3.2 Notion of State

An Alice world 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 student can easily see the current state. 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. There is no need for tedious hand traces of variable assignments.

3.3 Program and Language Constructs

A primary concern is that flow of execution is complicated by the fact that an Alice animation depicts simultaneously running processes. For example, suppose a virtual world where the aim is to move a bunny forward 2 directional units and then to move it backward 2 units. Here is a first try:

bunny.**Move**(**Forward**, 2)
bunny.**Move**(**Back**, 2)

When the above program snippet is executed, the bunny does not move at all! This is because, by default, Alice causes all animations to occur together, simultaneously.

Fortunately, Alice offers two control structures, **DoTogether** and **DoInOrder**, that allows the programmer to choose either simultaneous or sequenced actions. So, to instruct the bunny to move forward and then move back, we write:

```
DoInOrder (
   bunny.Move(Forward, 2),
   bunny.Move(Back, 2))
```

3.4 Events and Responses

The real challenge we encountered in teaching students about events and event handling was in providing a means for understanding how the control panel (and its widgets) are linked to the response. We are still working on this issue. An example of event-driven programming using the Alice programming environment can be found at www.ithaca.edu/wpdann/alice72.zip

4 PROBLEMS AND BENEFITS

As with many languages, Alice error messages are sometimes cryptic. We encountered difficulties as students moved away from simple closed animations (no user I/O) to more complex animations (requiring responses to events). The interweaving of Alice and Python statements requires some knowledge of what is an Alice statement and what is directly drawn from Python. For example, consider the function defined here:

```
def  silly() :
   DoInOrder(
     if cat.distanceTo (Fish) > 0.5:
     cat.Move (Forward, 0.5),
     Fish.Move (Back, 0.5),
     Fish.Turn (Left)             )
```

As of this writing, this function fails because the *if* statement is a Python statement (not an Alice animation statement) and, thereby, cannot be written inside Alice's **DoInOrder** construct because Alice does not know what to do with the Python statement inside the construct. One solution is to rearrange the code to position the *if* statement outside the **DoInOrder**:

```
def  silly() :
     if cat.distanceTo (Fish) > 0.5:
        cat.Move(Forward, 0.5)
        DoInOrder(
        Fish.Move (Back, 0.5),
        Fish.Turn (Left)
     )
```

A closely related problem is that of timing. The time duration (default is 1 second) can be a real problem in some scenarios. For example, in a recursive function, the animation takes a certain amount of time but the computer goes right on with its recursive calls. The result is that successive calls to the recursive function may be run simultaneously, rather than successively, if the timing is not handled correctly.

We believe that Alice provides some real benefits in teaching concepts of problem solving in the particular way used in computer programming as well as in teaching the fundamental programming constructs. A major benefit is the high level of student interest and involvement. The ability to make changes in program code and, within seconds, observe the effect on their animation contributed to sustaining that interest. Students were enthusiastic, voluntarily devoting much extra time to projects they considered fun and worthwhile. Based on student evaluations, it seems that students developed a justifiable sense of self-confidence in their programming skills.

Importantly, we saw that students exhibited an intuitive feel for objects, methods, and programming constructs such as repetition/recursion.

On the programming side of things, a number of significant benefits should be noted. There is, in general, no need for variables in Alice. The student issues commands that directly affect the objects in the animated world. In most programs, the student does not have to think whether a variable *x* has a value 5. The student can focus on the programming language constructs and how they work rather than variables and how they are modified.

The animated, virtual world is the state in Alice. Actually, this statement is not quite true (particularly as regards event handling). But, the student may view the animation as the state, simply looking at an animated world to see the location and orientation of objects in the world. Input and output are (generally) limited to the use of widgets/events. This is quite similar to the graphical modes of other languages, and tightly controls the environment. Students do not need to worry about input/output, except as confined to what reactions Alice should perform in response to events such as mouse clicks on widgets in the GUI.

5 CONCLUSION

Alice provides a 3-D animated programming environment that supports one approach for teaching problem solving in the particular way used in programming. We used Alice to develop algorithms for animating objects that populate virtual worlds. We observed that students were comfortable using objects and invoking methods on those objects. Students were able to watch what went wrong in their programs and easily debug and correct them. Future enhancements, a “no-typing” syntax-automated editor, promise to enrich the students’ experience with Alice .

ACKNOWLEDGEMENT

Alice and the Stage3 Research Group are sponsored by DARPA, NSF, Intel, Chevron, Advanced Network & Services, Inc., Microsoft Research, PIXAR, and NASA. The Stage 3 Research Group includes Steve Audia, Dennis Cosgrove, Adam Fass, Andrew Faulring, Cliff Forlines, Caitlin Kelleher, Shawn Lawson, Daniel Maynes-Aminzade, Dan Moskowitz, Jeff Pierce, Jason Pratt, Dave Stern-Gottfried, and Desney Tan.