Making the Connection: Programming With Animated Small World

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

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

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

Abstract

In learning to program, students must gain an understanding of how their program works. They need to make a connection between what they have written and what the program actually does. Otherwise, students have trouble figuring out what went wrong when things do not work. One factor that contributes to making this connection is an ability to visualize a program's state and how it changes when the program is executed. In this paper, we present Alice, a 3-D interactive animation environment. Alice provides a graphic visualization of a program's state in an animated small world and thereby supports the beginning programmer in learning to construct and debug programs.

1 Introduction

Instructors of introductory programming classes are faced with the challenge of helping students learn to design, build, and debug computer programs. In a large class, the challenge is often overwhelming, and teachers find themselves questioning why some students experience such a struggle in the process of learning to program. Many factors may contribute to the situation. To say that some students "do not know how to solve problems" is perhaps too simplistic. Most students have likely achieved a certain level of competency in mathematics and problem solving. We contend that these students are not strong problem solvers in the particular ways that are necessary for success in computer programming. In this paper, we look at visualization of program execution and program state as fundamental concepts that are especially important for learning to program. We propose using Alice, a 3-D interactive animation environment, to help students visualize program execution and program state.

Visualization of Execution and Program State: Soloway [12] notes that 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. We believe that beginning programmers not only need to learn how to design an algorithm for solving a problem but also how to translate the steps of the algorithm into specific programming constructs. But, making the connection between an algorithmic step and a specific programming construct requires some concept of how the computer executes that programming construct at runtime.

We have observed that many students have difficulty visualizing the steps of the execution of a program, or the current state at any given time. As a result, students have trouble figuring out what went wrong when things do not work. The source of confusion in building and debugging programs, in all but the most trivial code, may be an inadequate understanding of the program state during program execution. The role played by an understanding of the program state can be observed when students are asked to draw labeled boxes in tracing a function. An example is the swap function, commonly used to lay the foundation for understanding array sorting, as illustrated in Figure 1. We observed that calls to the swap function become difficult for students to trace. As an aside, novice programmers may stumble into the problem of interference (when functions with reference parameters are incorrectly invoked, see Reynolds [10]). Explaining the cause is difficult without a firm notion of program state. Thus, we argue that making the connection between steps in the algorithm and specific programming constructs is, to a significant extent, dependent on the student's notion of the program's state/environment and how the state changes when that program construct is executed.

As opposed to traditional, text-oriented programming languages, a 3-D virtual world has the advantage that some state (like object position and color) is intrinsic in the "natural" way to view the data itself. Students see the state and how it changes over time. Additionally, 3-D worlds, are more realistic than their 2-D counterparts and seem to encourage student exploration and curiosity in creating variants of existing worlds and making novel worlds of their own.

```
void swap (int& x, int& y)
  {  int temp;
     temp = x;
     x = y;
     y = temp;
  }
```

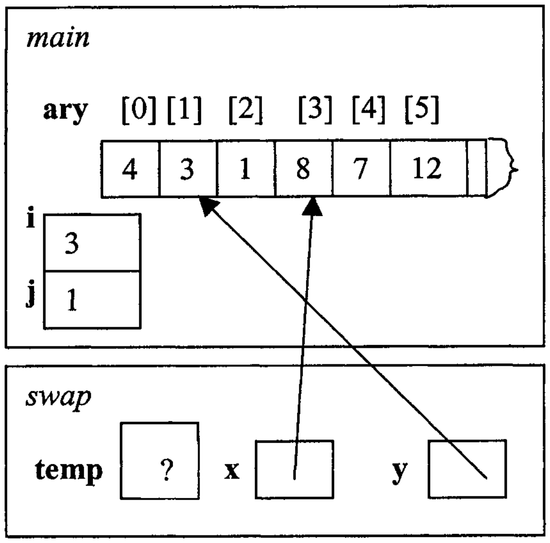

Figure 1. Trace of swap(ary[i],ary[j])

Related Work: A work that resembles Alice is LOGO, created by Papert[8] at MIT. LOGO uses an egocentric coordinate system and turtle graphics to provide a friendly programming environment for children to learn about geometry and related mathematical concepts. However, Papert discovered two troubling concepts, for beginning programmers using LOGO: variables and debugging.

Many programming language packages now include some sort of debugging utility. Debuggers allow the programmer to examine the program state (the values of the variables) any time during the program's execution. But, novice programmers tend to struggle with using debuggers. They do not know where to set breakpoints or which variables to examine. The debugger intrinsically adds a layer of complexity, which can make things worse, not better.

GROK [5] was designed as a data visualizer to help students more easily view their data transformation during program execution. However, it is only designed to work for Pascal programs and does not handle sophisticated data structures (such as linked lists or records).

The use of animation to show program execution is not a new idea [3,7,9]. Algorithm animation (e.g. XTANGO [13], BALSA [4], and program visualization [7]) has been developed with the idea of incorporating visualization into the learning process. The instructor programs the algorithm animation. The student just runs the program, observing the animation of the algorithm when using different inputs. Thus, algorithm animation packages generally do not help students in the visualization of their own programs.

One of the most successful educational simulation software packages has been *Karel, The Robot* [9]. Karel was used to prepare students for success in learning Pascal. Students were able to view the entire world (environment) of Karel, and watch how the world changed as their programs executed. 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++* [3]. Karel++ introduces a C++ like syntax for the Karel robot along with a significant level of code complexity over that required in the original Karel.

Turing's World [2] and JFLAP[11], are both state simulators used to demonstrate state and transitions in finite automata and other abstract machines. Students can easily watch the state change as their "programs" are run. However, these packages are designed to demonstrate state transitions for more advanced programmers, not for novice programmers. Toontalk [6], is a 2-D virtual reality package designed to introduce elementary students to the world of computers.

We believe that Alice can be used to follow in the tradition of LOGO and Karel. By using the 3-D, animated environment students can create their own virtual worlds. Alice allows students to view the state of their world as their program executes.

2 What is Alice?

Alice98 (www.alice.org) is a 3-D Interactive Graphics Programming Environment for Windows built by the Stage 3 Research Group at Carnegie Mellon University under the direction of Randy Pausch. A goal of the Alice project is to make it easy for novices to develop interesting 3-D graphic animations. Alice98 is a full 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. Each action is animated smoothly over a specified duration. (This replaces the traditional animation methodology, where the animator prepares many frames and uses a frame animator to view a succession of frames in rapid sequence.) Alice is built on top of the Python language (www.python.org) and uses many of Python's features.

Users interested in using Alice may download the free program from www.alice.org. Users who would like additional information may wish to look at the authors' textbook (www.ithaca.edu/~wpdann/alice1298).

3 Experiences with using Alice

We have used Alice as an instructional tool for two summer sessions at Ithaca College. High school students were enrolled in a special Summer College program. Our observations of students working with Alice are the basis of many of the viewpoints presented in this paper. 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 next year at Ithaca College. Discussions have begun about doing the same at St. Joseph's University.

4 An Introduction to State in Alice

While Alice was not originally designed as a teaching tool for novice programmers, it serves this role quite effectively. Its high visual content allows students to immediately see how their animated programs run. Perhaps one of the most valuable features of Alice is that the animated virtual world ***is*** the program state. In other words, the program's state is immediately and always visible to the user. There are no variables, *per se*, in Alice. The absence of variables in Alice follows in the SELF [14] and ACTORS [1] model. The program's state is composed of each of the objects that populate a virtual world, together with information about those objects (such as the object's position in the world, its orientation in the world, and its color, parts, size, and child objects). The objects themselves contain all state information generally needed.

When the programmer issues a command to cause an object to move, she immediately sees the object move through the virtual world. This visualizes the change in the object's (and program's) state. The highly visual feedback allows the student to relate the program "piece" (issuing a command to change the state of an object) to the animation action (in which the state change is displayed on screen).

Alice provides several built-in action commands that change the state of an object. 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, Turn**, **Roll**, and **PointAt**). Commands that change the physical nature of objects include object destruction (e.g. **Destroy**), dynamic object creation (e.g., **AddObject**), object resizing (e.g. **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

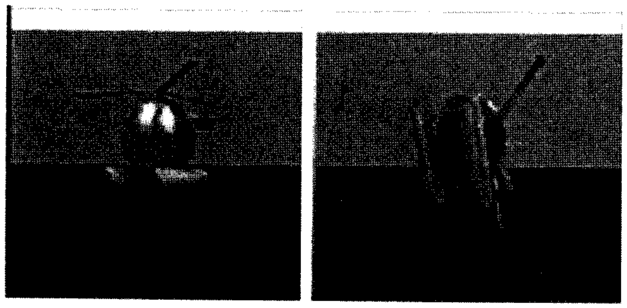
Figure 2. Turning backwards

subset), we describe here two action commands to illustrate details.

Turn: Turning is allowed in 4 directions: Forward, Back, Right, and Left. In the Turn command, it is only necessary to specify the object to be turned, the direction, and how much it is to be turned. Figure 2 illustrates turning along one of the rotational axes. Explaining the necessary changes to the helicopter's rotational orientation is hard to do. But as the student sees the helicopter turn backwards, it becomes easy to see the change in state.

Resize: Resizing is done to allow an object to be made larger or smaller (either the whole object, or just along 1 dimension. While it is difficult to describe the state changes that occur as a result of resizing an object, the student can see the object change size, and intuitively knows the state must have changed.

Alice also provides passive commands that change the state of an object. The most significant of these commands are **BecomeParentOf** and **BecomeChildOf**. The result of an object being a parent of another object is that when the parent object moves or turns, so too does the child object. The only immediate change that occurs after issuing such a command is the object tree hierarchy change. However, as soon as an action command is issued for the parent object, the child object will move as well. This is different in that (1) it is instant, not animated, and (2) it really is not a visible change of state. It is detected later, when a side effect of moving the parent is that child object also moves.

The net result is that the student does not have a difficult time visualizing the internal state of her world. As more complicated programming constructs are encountered, the student is able to focus on learning and understanding the concept without having to deal with variables and the internal state of the program. With no variables, what she sees is what her world really is!

Expanding the State – Allowing variables: Since the objects do not, in general, track their own histories, it may be helpful to use variables to track such information. While there are no variables, *per se*, in Alice, there are two ways to introduce variables. The first approach is by using Python variables. Since Alice is constructed on top of Python (**www.python.org**), it is perfectly legal to introduce Python variables into an Alice program. In general, writing Python code within an Alice program does not generate any problems. We do not recommend this approach because we view the lack of explicit variables in Alice as a desirable property to maintain.

A second approach is to make use of the objects, themselves. Each object has member methods that allow the object itself to store and retrieve a value. This approach can be used when widgets are introduced as part of event-driven programming. It is often useful to track how many times a button has been pressed or whether or not a particular radio button has been selected.

5 Putting the Pieces Together

Alice provides mechanisms for decision statements, repetition (including recursion), functions, and procedures. As students progress, they learn new programming constructs and how to put the pieces together. As examples, we discuss the implementation of procedures and recursion.

Procedures (implemented as named instructions): Instructions in Alice are often grouped together. A group of instructions may be run either consecutively (**DoInOrder**) or as simultaneous threads (**DoTogether**) or as a combination of both. The programmer may give a name to a group of commands. Named instructions behave like procedures by performing side effects and not returning a value. 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 then 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 absence of variables (and not having to trace internal state) allows the student to focus on visualizing the functionality of the procedure construct. This lets the student see if her *Hop* procedure makes a rabbit hop. While one could argue that a similar Hop procedure could be written in Pascal or C, the key difference is that visualization helps in making **all** named instructions in Alice conceptually simple to write.

Recursion: Alice provides support for generalized recursion through the **SetAlarm** command. While the concept of recursion may seem quite alarming, it really need not be.

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

This code checks whether the Fish is close enough to the cat. If not, the Fish moves in the direction of the cat and the alarm is set to repeat the whole process. Note that the time must be set so that the alarm does not "go off" (and repeat the whole process) until after 2 seconds. By default, each instruction takes 1 second to run. So, 2 seconds are needed before the function should be recursively called.
The key concept with respect to recursion is that the delay makes the recursion temporarily visible. The motion must be completed and the state must change before the recursive call can be made again.

6 Conclusions

Alice provides a new instructional tool for learning problem solving in the particular way used in programming. The absence of variables, combined with high visual content, provides a good environment for helping students understand the state of their programs and the state of their virtual worlds. Students were comfortable populating their virtual worlds with objects, and in invoking methods on those objects. Students were able to watch what went wrong in their programs and easily debug and correct them.

References

[1] Agha, G., *ACTORS: A Model of Concurrent Computation in Distributed Systems*, Cambridge, MA: MIT Press, 1986.

[2] Barwise, J. and Etchemendy, J., *Turing's World 3.0*, Stanford, CA:CSLI Publications, 1993.

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

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

[5] Dann, W. *Dynamic, Generic Visualization in a Programming Language Environment*. Ph.D. Thesis, Syracuse University, 1997.

[6] Kahn, K., ToonTalk - An Animated Programming Environment for Children, in *Proceedings of the National Educational Computing Conference*, (June, 1995).

[7] 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.

[8] Papert, S., *MindStorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

[9] Pattis, R., *Karel the Robot*. New York: Wiley, 1981.

[10] Reynolds,J.C., *The Craft of Programming*. Englewood Cliffs, NJ: Prentice Hall International, 1981.

[11] 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.

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

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

[14] Ungar, D. and Smith, J., SELF: The Power of Simplicity, *OOPSLA 87, Conference Proceedings*, published as *SIGPLAN Notices*,22,12,(1987), 227-241.

Acknowledgement

Alice and the Stage3 Research Group are sponsored by DARPA, NSF, Intel, Chevron, Advanced Network & Services, Inc., Microsoft Research, PIXAR, and NASA.