

Running Head: Animated 3D To Prepare Novices

Using Animated 3D Graphics To Prepare Novices for CS1

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

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

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

ABSTRACT

A new programming course to prepare novices for the traditional Computer Science 1 course (CS1) is proposed. The course uses 3-D interactive animation in a novice-friendly environment to introduce object-oriented programming concepts and help students develop problem-solving skills. Pedagogical issues are presented that involve the use of visualization and graphics concepts, the notion of state, and programming language concerns. A study of practice and experimentation with this course is underway. Early results are summarized along with observed benefits and concerns.

USING ANIMATED 3D GRAPHICS TO PREPARE NOVICES FOR CS1

INTRODUCTION

Attrition from the computer science major has been a longstanding problem. Work in 1970's and 80's identified certain characteristics a student could have that increase their probability of success in a computer science curriculum. The characteristics identified include: the ability to think algorithmically, to apply problem-solving techniques, and to perform logical reasoning. (Butcher & Muth, 1985; Stephens et al., 1985; Oman, 1986) Studies in the past few years (Davy et al., 2000; Hagan & Markham, 2000) have shown that students without prior programming experience are at a decided disadvantage in being able to complete a computer science degree program. This is in strong disagreement with the work of many earlier researchers who observed little to no correlation between prior programming experience and success in a computer science curriculum. We believe that this shift in explanation may be related to the change by most computer science departments from imperative languages to object-oriented (OO) languages. OO languages not only require the teaching and learning of all of the material for an imperative language, but also add the concepts of class, object, information hiding, inheritance, and polymorphism. Many computer science programs also use event-driven GUI programming in CS1, which adds yet another model of computation that students must master. As the length of the semester remains constant, the increased breadth of material overwhelms students with no prior programming experience, leading to the observations by Davy et al. (Davy et al., 2000) and Hagan and Markham (Hagan & Markham, 2000). The density of concepts and information now present in CS1 exacerbates an already troublesome dropout situation.

How can we provide the necessary previous programming experience, suggested by (Davy et al., 2000) and (Hagan & Markham, 2000)? How can we support the development of fundamental problem solving and logical reasoning skills? A possible answer is a pre-CS1 course in which novices (students with little or no previous programming experience) can develop the necessary problem solving and logical reasoning skills, at the same time learning programming language constructs, the design of algorithms, and the connection between the two. In this paper, we propose a pre-CS1 course, for novices only, with the heavy use of 3D graphics and visualization as a key component.

Why graphics?

Students are often attracted to computer science because of their experiences with video games, animated films, and multi-media. They are disappointed to discover that they cannot write programs that perform these (or similar) tasks until 2-3 (or more!) years into the curriculum. Use of 3D graphics animation as an essential component to an introductory course may meet their expectations "up front" in their computer science curriculum. This may help to further establish their interest, and may help in their persistence, as they cover topics in their curricula that are more abstract.

Why visualization?

Soloway (Soloway, 1986) notes that the real difficulty for novice programmers lies in "putting the pieces together"; that is, in figuring out what constructs to use and how to coordinate those constructs. Novice programmers not only need to learn how to design an algorithm for

solving a problem but also need to learn how to translate the steps of the algorithm into specific programming constructs. 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 of the program at any given time. The use of graphics and visualization is recognized as an effective teaching tool in computer science to help students put together programming constructs and algorithms (Naps, 1996). Among the many researchers who argue for visualization, Shu (Shu, 1988) 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.

Relevant Work

Over the last two decades, researchers have developed and used several visualization tools. LOGO, created by Papert (Papert, 1980) at MIT in the early 80's, uses an egocentric coordinate system and turtle graphics to provide a friendly programming environment for children. Using LOGO, children learned 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 that shows the program state (the values of the variables) any time during a program's execution; but, novice programmers tend to struggle with debuggers. Novices do not know where to set breakpoints or which variables to examine. Thus, the debugger adds a layer of complexity that can make things worse, not better.

GROK (Dann, 1997), designed by one of the authors, is a data visualization tool that helps students more easily view their data transformation during program execution. GROK is only designed to work for Pascal programs and does not handle sophisticated data structures (such as linked lists or records). Karel, a robot world simulation, was used in the early 1980's to introduce structured programming concepts, similar to Pascal (Pattis, 1981). In the mid-1990's, Karel was adapted for use as an introduction to OO concepts, similar to those used in C++ (Bergin et al., 1997). But Karel++ introduced a more complicated code structure, and was more difficult to program (being less intuitive for students). Specifically, Karel++ moved from using one robot (and the actions it could perform) to a robot model. Inheritance makes the robot models far more complex, as multiple robots can have multiple capabilities. Students often found that the higher level of complexity was not counterbalanced by a higher payoff.

Algorithm animation tools, such as XTANGO (Stasko, 1992) and BALSAM (Brown, 1988), were developed to incorporate visualization into the learning process for algorithm analysis courses. The instructor programs an animation for commonly used algorithms (e.g. a quicksort animation shows little shapes bouncing around from one array slot to another). The student runs the *prepared* animation, observing the animation of the algorithm when using different inputs. Bower (Bower, 1998) writes, "(i)t is important that such simulations are manipulative in order to actively involve the learner rather than...passively watch pre-programmed instruction." The Generalized Algorithm Illustration through Graphical Software (GAIGS) package, developed by Naps (Naps & Swander, 1994) offers the ability to rewind a prepared animation to view it again. Algorithm animation tools have helped students understand algorithm analysis. But students can only view previously prepared animations; thus, algorithm animation packages generally do not help students in the visualization of *their own* programs.

Turing's World (Barwise & Etchemendy, 1993) and JFLAP (Rodger, 1996) 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 novices. Toontalk (Kahn, 1996) is a 2D virtual reality package designed to introduce elementary students to the world of computers. While it is designed for novices, its target audience is too low for our freshmen students.

Another 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 tend to rely on an underlying programming language (such as C or scheme) that the students need to learn and master before they can effectively use the visualization tool.

In summary, visualization seems to be a key pedagogical tool. We believe teaching fundamental concepts in a *preliminary* course using graphics and visualization in an OO environment, will provide sufficient "prior programming experience" (as recommended by Davy et al., 2000; Hagan & Markham, 2000). Our suggested use of 3D graphics in a first programming course follows in the footsteps of House and Levine (House & Levine, 1994) who used Jabka to render 3D models in a computer graphics course for non-majors. However, none of the current tools provide program visualization for the novice programmer within the confines of an OO world in a way that we need.

Our approach to teaching introductory programming to novices centers around a software system named Alice, similar in flavor to Karel and LOGO. By using the 3D animation environment, students can create their own virtual worlds and view the state of their world as their program executes. We use Alice as a convenient and easy-to-use 3D graphic animation tool to support the pedagogical goals of the course: a fundamental introduction to objects, methods, decision statements, loops, recursion, and problem solving.

WHAT IS ALICE?

Alice is a 3D Interactive Graphics Programming Environment built by the Stage 3 Research Group at Carnegie Mellon University under the direction of Randy Pausch (Pausch et al., 1995). Readers interested in using Alice may download the free program from <http://www.alice.org/jalice>. Originally designed for Windows, Alice now has an Open-GL rendering option suitable for any platform. A goal of the Alice project is to make it easy for novices to develop interesting 3D environments and to explore the new medium of interactive 3D graphics. Alice offers a full scripting and prototyping environment for 3D object behavior in a virtual world.

3D models of objects (e.g., animals and vehicles) populate a virtual world, providing an object-oriented flavor. By writing simple scripts, users can control object appearance and behavior. In fact, it is not necessary to type lines of code at all. Using the smart editor, students can drag and drop the instructions that make up their programs. 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 was originally built on top of the programming language Python (www.python.org) but is now fully implemented in Java (java.sun.com). Scripting in Python is still available for use by more advanced programmers. It is important to note that novice programmers can easily program in the Alice environment without first learning either Java or Python. Additional information on programming with Alice is available at <http://www.ithaca.edu/wpdann/alice2001/alicebook.html>.

Programming 3D animations allows students to immediately see how their program code executes. The highly visual feedback allows the student to connect individual lines of code to the animation action. This leads to an understanding of the actual functioning of different programming language constructs. Many of the programming language constructs used in building 3D animations in Alice are described below.

Actions (State Transformers)

Action commands are similar to assignment statements in more traditional programming languages. They are state transformers (i.e., they change the state of the world.) 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 translational movement of objects within the world (e.g. **Move**), rotating them about their 3D axes (e.g. **Turn** and **Roll**), and pointing at other objects (e.g. **PointAt**). Commands that change the physical nature of objects include object resizing (**Resize**) and making objects visible/invisible (changing the degree of opacity).

While it is beyond the scope of this paper to discuss all of the action commands, we illustrate the rotational actions **Turn** and **Roll**. Turning is allowed in 4 directions: Forward, Back, Right, and Left. Rolling is allowed in two directions: Right and Left. Together, turn and roll actions provide six degrees of rotational freedom. In both commands, it is only necessary to specify which object is to be turned/rolled, the direction, and how much. Figures 1, 2, and 3 illustrate turning along each of the rotational axes.

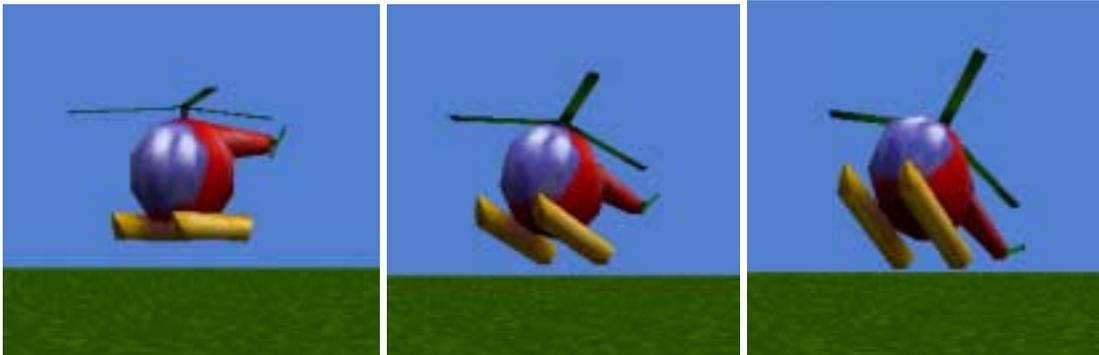


Figure 1. Turning an object backwards

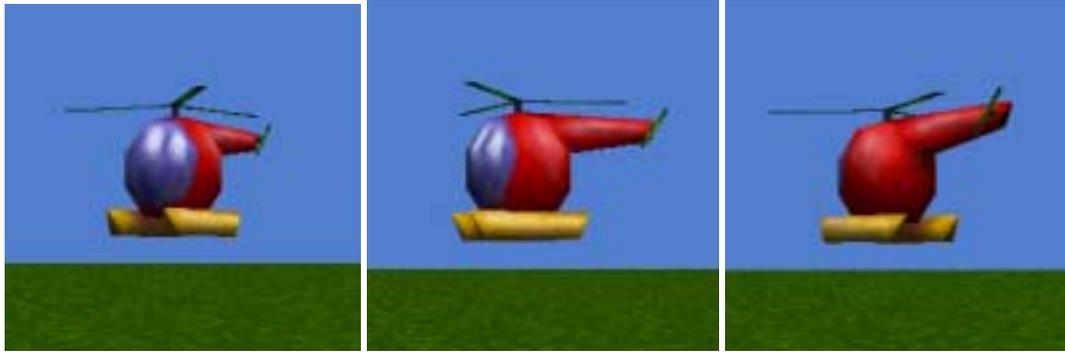


Figure 2. Turning an object right

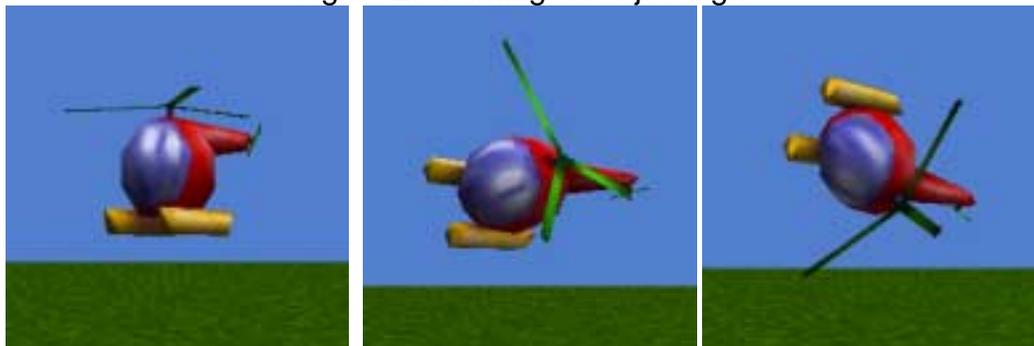


Figure 3. Rolling an object left

Behavioral Methods

The concept of a behavioral method 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 sense intuitively. 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 the lines of code shown in Figure 4 cause the bunny to hop twice. Again, the animation allows the student to immediately visualize the functionality of program constructs.

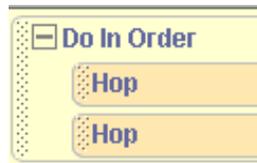


Figure 4. Hop Sequence

Functions

Functions are supported through Python scripts. However, functions are rarely used, as they 1) require the students to type code instead of using the drag-and-drop editor, 2) are written in Python, and 3) do not provide direct visual feedback as the rest of the statements in Alice do. They are primarily used when needing to do 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)
```

Decisions and Expressions

Decisions are expressed using a traditional if...then...else structure. Because of the visual feedback, students are able to see immediately the results of a decision statement. For example, in a scene where the cat and the fish are chasing each other at the beach, as seen in Figure 5, if the distance between the cat and the Fish is less than one meter, the student sees the cat jump up and down. Otherwise, nothing happens.

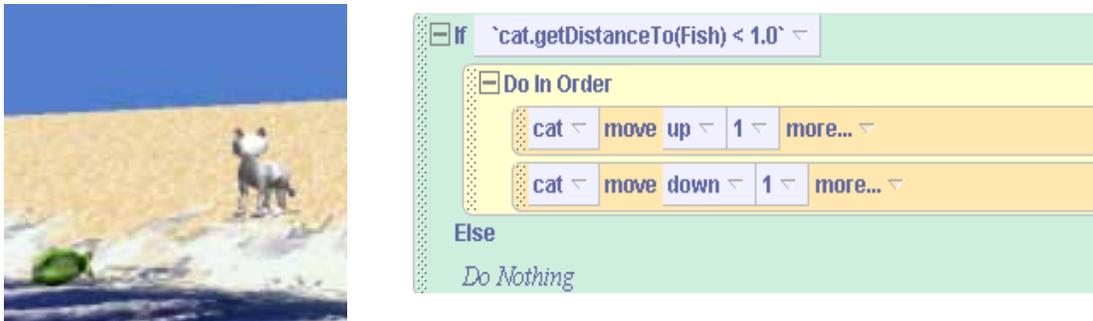
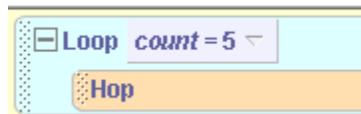


Figure 5. Decision Statement

Repetition

Alice provides three forms of support for repetition. The first is through the **Loop** instruction. For example, the Loop instruction shown below executes a Hop instruction (a named



instruction defined elsewhere in the program) that causes the bunny to hop 5 times. Other repetition instructions include a traditional while loop that uses a **While** statement and a generalized recursion, through recursive calls to the same function. The code snippet in Figure 6 defines a recursive method named Chase.

Simply put, the Recursive Chase code checks whether the Fish is more than two meters from the cat. If so, the Fish moves 1 meter toward the cat, the action waits for twoseconds, and then recursively calls Chase to repeat the whole process. 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.

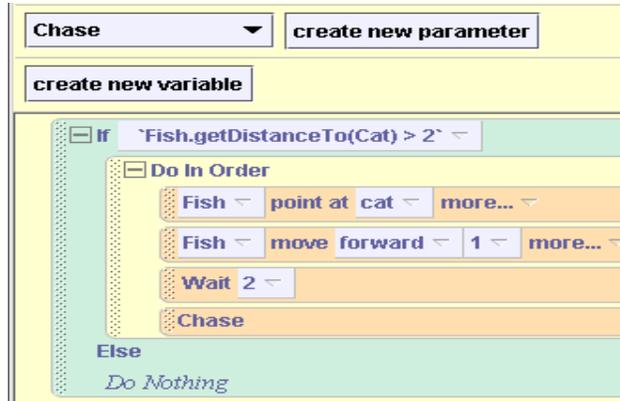


Figure 6. Recursive Chase

Events/Interactions

Alice provides support for interactive programming through the handling of events. Events allow the user to interact with the animated world. An event signals a mouse click, keyboard entry, joystick control, etc. In response to an event, a behavioral method is called (see section 2.2 above).

Concurrency

Groups of instructions may be run either sequentially (DoInOrder) or as simultaneous threads (DoTogether) or in a nested combination of both. Figure 7a shows the code snippets executed to instruct AliceLiddell to walk forward and then turn right 1/4 turn in order versus instructions to walk forward and turn right 1/4 turn at the same time. Figures 7b, 7c, and 7d illustrate the difference between the actions.



Figure 7a. Sequential versus Concurrent Code Snippets



Figure 7b. Original Scene



Figure 7c. After DoInOrder



Figure 7d. After DoTogether

Collections

Lists and arrays are handled in Alice using collections. Collections operate similarly to a Python list, allowing the grouping of several objects together. A For-Each statement block provides for automatic iteration through a collection of objects.

PEDAGOGICAL ISSUES

Designing a course that centers on 3D animation as the primary instructional tool presents a number of pedagogical issues. In this section, issues are summarized for three major areas: graphics concepts, the notion of state, and programming language concerns.

Graphic and Animation Concepts

A question that must be answered is: how much do students need to know about 3D graphics and animation? We found that Alice lowers the cognitive burden of creating 3D 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. Additionally, we found many students quite comfortable working in 3D. Students have grown up in a video-age, accustomed to playing 3D computer games.

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 8a. (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 8b, 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.

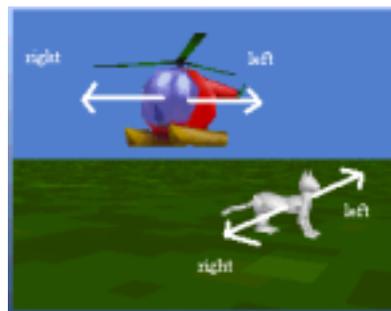
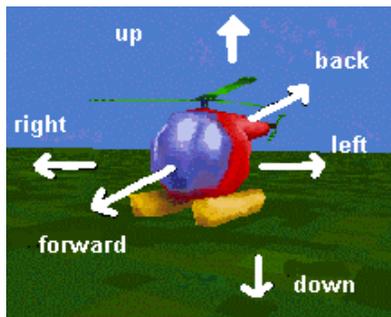


Figure 8a. Six Degree Orientation

Figure 8b. Different Orientations

Notion of State

We have observed that, in typical programming classes, many novice programmers have difficulty in understanding the program state (the values of all of the program's variables at any time during the execution of a program). As a result, students have trouble figuring out what went wrong when their programs 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. In traditional programming, 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 9. We observed that repeated calls to the swap function become difficult for students to trace. Explaining the cause for the calls 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.

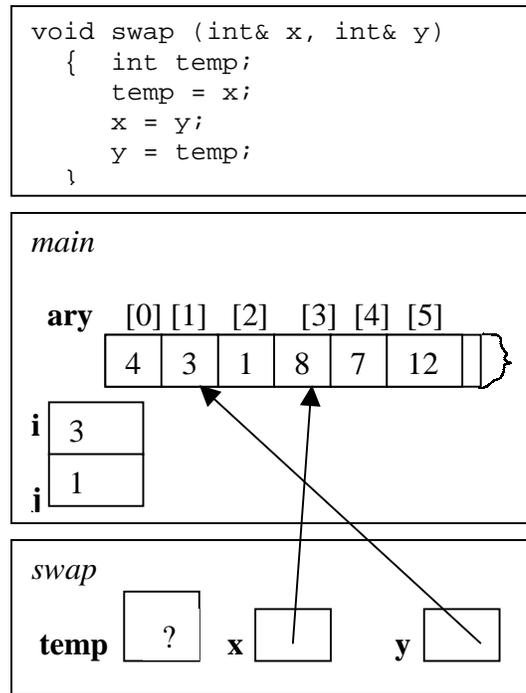


Figure 9. Trace of `swap(ary[i],ary[j])`

As opposed to traditional, text-oriented programming languages, a 3D virtual world has the advantage that some state (like object position and color) is intrinsic in a "natural" way to view the data itself. Perhaps one of the most valuable pedagogical features of an animated virtual world is the program state. In other words, the program's state is immediately and always visible to the user. There is no need for mutable variables. The absence of mutable variables follows in the SELF (Ungar, & Smith, 1987) and ACTORS (Agha, 1993) 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 opacity). The objects themselves contain all state information generally needed.

When a command is issued to cause an object to move, the student can immediately see the object moving 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). The net result is that the student does not have a difficult time visualizing the internal state of the 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 is seen is what the world really is!

Program and Language Constructs

The most interesting programming language construct to discuss is recursion. Recursion is a key computer science concept. As argued by McCracken, “recursion is fundamental in computer science, whether understood as a mathematical concept, a programming technique, a way of expressing an algorithm, or a problem-solving approach.” (McCracken, 1987) Laying a firm foundation in recursion in early courses may make topics studied in later courses easier to grasp. Unfortunately, novice programmers seem to struggle with the concept of recursion. Some computer science educators have described the process of teaching recursion as “one of the universally most difficult concepts to teach.” (Gal-Ezer & Harel, 1998) Bower (Bower, 1998) states that “(t)o understand the process of recursion and to be able to write a recursive definition or program one must be able to visualize the nature or structure of a problem and how solutions to smaller, similar problems are combined to solve the original problem.”

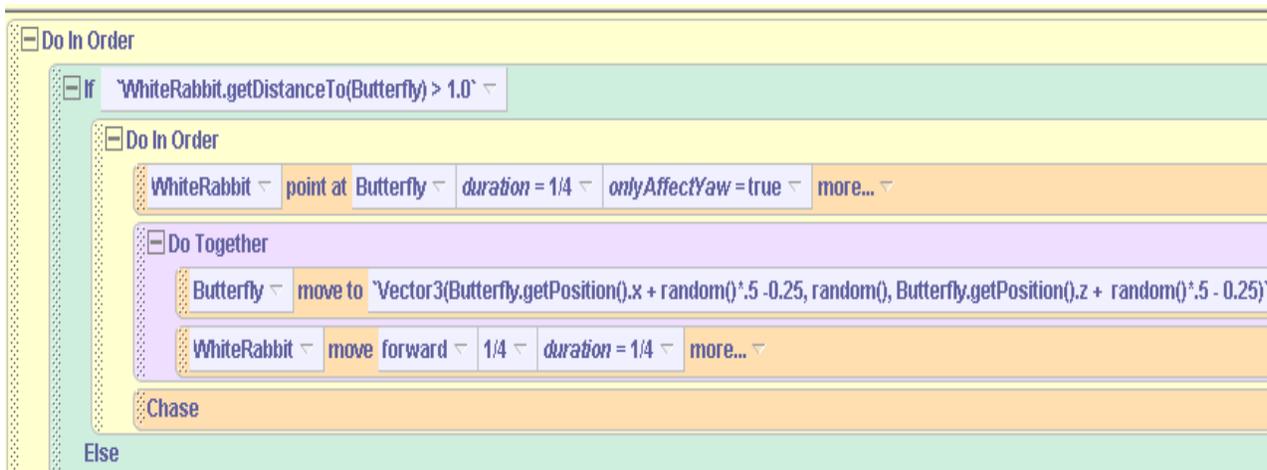


Figure 10. Recursive Chase Scene and Code

In animation programming, recursion is a means of creating more powerful animations with repeated actions that get closer and closer to completing a task. Below, two different examples of animations that use recursion are illustrated, the first focusing on generative recursion, and the second illustrating structural recursion. The generative recursion example is an animation of a

chase scene. In Figure 10, the butterfly moves in a random direction and the rabbit moves in pursuit.

If the Rabbit is within one meter of the butterfly, the chase has ended. Otherwise, three actions occur. The Rabbit points itself toward the current location of the butterfly (actually, to the point directly beneath the butterfly). Then, the Rabbit moves towards the butterfly while the butterfly moves in a random direction. Finally, a recursive call is made to the function to do the whole process again.

A structural recursion example is the Towers of Hanoi problem. In experiments with different numbers of rings students gradually recognize the pattern (and develop an abstract notion of recursive decomposition). If there is only 1 ring to move, it moves (the base case). Otherwise (the recursive case), n-1 rings move onto the spare peg. The bottom one can then move to the target peg. Finally, another recursive call is made to move the n-1 rings onto the target peg. Figure 11 illustrates the Towers of Hanoi method.

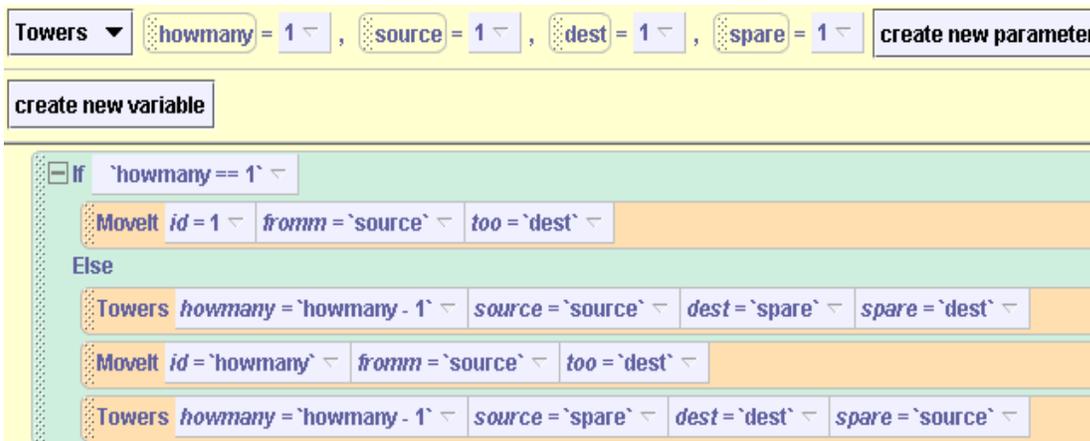


Figure 11. Towers of Hanoi Animation Method

Depending on the source and target pegs, a ring move involves an appropriate distance in a specific direction. Figure 12 shows a move in progress. Figure 12 illustrates the `moveIt()` function that accomplishes this task, returning the correct motion animation as a result. (Moving left a negative amount is the same as moving right.) The `which()` function provides a correlation between the hard-coded name of a ring and its corresponding number.



Figure 12. Towers of Hanoi Animation in Progress

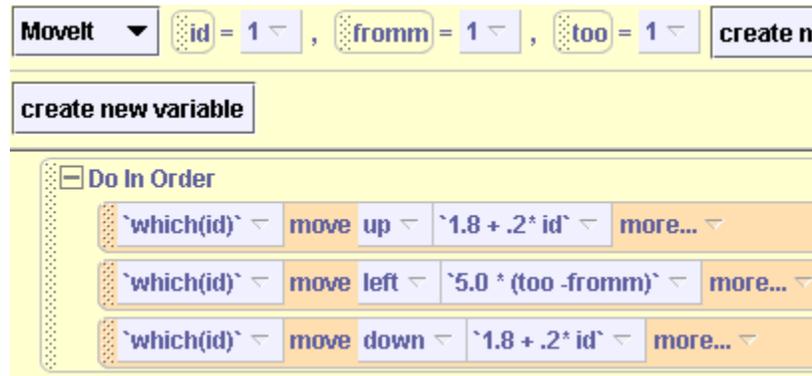


Figure 13. Towers of Hanoi MoveIt Code

COURSE PRACTICE AND EXPERIMENTATION

Our course has been used in two different formats since 1998. At Ithaca College, for two consecutive summers, a 2-week version of the course was offered to gifted and talented high school students. Based on the positive findings of these programs, we developed a credit-bearing course for college students. The course was experimentally introduced in the fall semester of 2000. This year, the authors are conducting formal college courses at Ithaca College and Saint Joseph's University. Instructional materials are being developed, including a text with a large selection of sample worlds, laboratory exercises, and projects. The authors are receiving NSF support to develop these materials. Several faculty members at other colleges and universities have requested permission to use the materials this year.

The goal of the course is to provide an opportunity for students to learn the fundamental concepts of programming and problem solving. The course is recommended to students who have little or no previous programming experience before enrolling in CS1. The topic outline, taken from the syllabus of the course, is shown below.

3D Graphics and Animations

- | | |
|--|-----------------------------|
| Getting Started | * Events and Responses |
| * Why Alice? | * Sound |
| * Alice Virtual Worlds (Demo) | Powerful motion techniques |
| * Using The Alice Interface | * Expressions |
| | * Decisions: If ... Else |
| Types of Animations | Statements |
| * DoInOrder vs. DoTogether | * Random Motion |
| * Movie vs. Interactive | * MoveTo: Location Vector |
| Classes and Objects in a Virtual World | Repetition |
| * Object Properties and Parts | * Loop |
| * Orientation and Degrees of Freedom | * Recursion |
| * Motion in 3D Space | * Recursive Function Call |
| | * Structural vs. Generative |
| Building An Animation Program | * While |
| * Scenarios and Tasks | * Infinite Loops |
| * Problem Solving Concepts | Collections of Objects |
| * Storyboards | * A List of Objects |
| * Behavioral Methods | * Sorting Objects |
| * Parameters | Projects |
| * Class vs. world methods | * Collision Detection |
| * Vehicles | |

Early Findings

We are conducting a study of the course's effect on attrition, compiling data for statistical analysis. This study, under NSF funding, will play out over the next 2 years. Our experiences in developing and teaching the course and our observations of students working with 3D animations are the basis of viewpoints presented in this paper. Early findings of this study are presented in this section.

In follow-up surveys conducted for courses offered thus far:

- 85 % of students rated the course 9 or 10 (scale of 1 – 10, 10 being high).
- Students indicated a high level of comfort in working in a 3D animation environment.
- Students self-reported an average of 6 – 9 hours per week on outside-of-class work for the course.
- Students expressed a sense of self-confidence in their programming skills.

Students demonstrated a high level of involvement. One student commented: "I like programming now that I understand what it is." Teaching assistants report that about 50 percent of students enrolled in the courses spent extra hours in the lab working on projects. On exams, students were able to construct methods using if, loop, and recursion structures with no more than minor errors. We emphasize that students did not necessarily “master” the intricate concepts of recursion, but did experience working with recursion and exhibited some facility with using the technique.

Benefits

Some direct benefits have been immediate. The drag-and-drop editor relieves the initial struggle with syntax that is often encountered by novice programmers. All the parentheses, semicolons, and other syntactic details are automatically handled by the interface. Interestingly, students seemed to exhibit little or no trouble in reproducing the syntax in the code they wrote by hand, on exams given later in the course. A few undeclared or non-computer science students have changed their majors to computer science or adopted a computer science minor. In what way this compares to changes of major on a coincidental basis is not known.

In constructing animations, students necessarily gain some understanding of the 3D coordinate system and the spatial relationship of objects to one another. This experience provides a context in which to learn about pixels and screen coordinates, which may be helpful in completing graphics examples and projects in CS1 and CS2. In each course offered thus far, several students have expressed their interest in taking a later course in computer graphics. We believe that the students' study of 3D animation has laid a firm foundation for their later coursework.

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. There is no need for tedious hand traces of variable assignments.

Although recursion is often not taught to novice programmers, we found students' insight into chase scenes and game playing provided a context for understanding recursive solutions. In conducting the course, recursion was introduced in the 3rd or 4th week. The highly graphic and visual environment allowed students to immediately see what was going on in their world. For example, students were asked to write a recursive program that reversed a list of bunnies in a Lisp-like fashion. When one of the bunnies stopped too soon, kept moving beyond its "correct" destination, or moved in the wrong direction, a student could immediately see what was wrong with their program. This is in stark contrast with students learning recursion in a traditional environment where, for example, a missed base case leads to an infinite recursion with no output—and the student not knowing why the program fails to terminate.

Concerns

As with many programming languages, runtime error messages in the Alice environment are sometimes cryptic. An occurrence of a "null pointer exception" has little meaning to a novice programmer. In some cases, students have no idea what the source of the error is and no idea how to deal with the problem. (An example of this situation occurs when students pass the wrong type of object as a parameter to a method.) The Alice development team is working to reduce the possibility of making these kinds of mistakes and to make error messages more helpful when such mistakes occur.

At the time of this writing, functions in Alice are not easy for novice programmers to use. This is because functions must be written in Python scripts. Though, for most projects at the novice level, there is little need for functions.

The programs for 3D animation virtual worlds can occupy 2 or 3 megabytes of memory (especially when sound is involved). These worlds cannot fit on a floppy disk but can be zipped to reduce the memory storage. Even so, a floppy disk will then store only one world. We found that students adapted by switching from floppy disks to zip disks to store their collection of worlds.

Other technology-related concerns involve the use of keyboard controls and sound. In action-packed interactive worlds, keyboard controls are somewhat inadequate for steering an airplane in a flight simulation or driving a car in a race. Some students opted for buying their own joysticks to use with their interactive projects. The use of sound adds to the effectiveness and quality of an animation; but, many computer labs in campus settings are not equipped with speakers.

CONCLUSION

We will need at least two more years to collect and analyze sufficient data to ascertain the cumulative value of this approach. However, positive initial student feedback coupled with an increased interest from faculty at other universities has given us optimism in our project. Based on three years of experimentation and development of this approach, we believe that using 3D animation programming in a pre-CS1 course provides some real benefits in teaching fundamental programming and problem solving concepts. A major benefit is the high level of student interest and involvement. The students' ability to make changes in program code and, within seconds, observe the effect on their animation contributed to sustaining their interest. Students were able to watch what went wrong in their programs and easily debug and correct them. Based on their

evaluations, students seem to have 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 and recursion.

Introducing powerful programming constructs for decision-making and repetition using 3D animation employs a combination of visualization, experimentation, and mathematical explanation. Recursion can be presented as a means to create more powerful and interesting animations. From a programming perspective, a number of significant benefits should be noted. There is, in general, no need for variables. The student issues commands that directly affect the objects in the animated world. 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 external events. This is quite similar to the graphic modes of other languages, and tightly controls the environment. Students do not need to worry about input/output, except as confined to what behavioral methods should be performed in response to events such as mouse clicks or keyboard entry.

A current study is following the students who took our course to compare their progress against students of similar background who did not. Future plans include a follow-up study to assess whether experience with animation programming will help students when they learn more about recursion in CS1/CS2. While our experiences involved using 3D animations to introduce fundamental programming concepts in a pre-CS1 course (or a class that runs concurrently with CS1), we anticipate experimenting with the use of course materials during the first couple of weeks of a CS1 class. This would follow the approach presented by Scragg et al. (Scragg et al., 1994) who recommended introducing students to basic computer science concepts prior to launching into the traditional course material for CS1.

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.

REFERENCES

- Agha, G., (1993). ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA: MIT Press.
- Barwise, J. & Etchemendy, J. (1993). Turing's world 3.0. Stanford, CA: CSLI Publications.
- Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (1997). Karel++, a gentle introduction to the art of object-oriented programming. New York: Wiley.
- Bower, R.W. (1998). An investigation of a manipulative simulation in the learning of recursive programming. (Doctoral dissertation, Iowa State University).
- Brown, M.H. (1988). Algorithm visualization. Cambridge, MA: M.I.T. Press.
- Butcher, D. & Muth, W. (1985). Predicting the success of freshman in a computer science major. Communications of the ACM, 28(3), 263-268.
- Dann, W. (1997). Dynamic, generic visualization in a programming language environment. (Doctoral dissertation, Syracuse University).
- Davy, J.D., Audin, K., Barkham, M., & Joyner, C. (2000). Student well-being in a computing department. Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, 136-139.
- Gal-Ezer, J. & Harel, D. (1998). What (else) should CS educators know? Communications of the ACM, 41(9), 77-84.
- Hagan, D. & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, 25-28.
- House, D. & Levine, D. (1994). The art and science of computer graphics: A very depth-first approach to the non-majors course. Proceedings of the 25th SIGCSE Technical Symposium, Phoenix, March, 334-338.
- Kahn, K. (1996). ToonTalk - An animated programming environment for children. Journal of Visual Languages and Computing.
- McCracken, D.D. (1987). Ruminations on computer science curricula. Communications of the ACM, (30,1), 3-5.
- Naps, T.L. Chair, Working Group on Visualization (1996). An Overview of visualization: its use and design. Proceedings of the Conference on Integrating Technology into Computer Science Education, Barcelona, Spain, 192-200.
- Naps, T. & Swander, B. (1994). An object-oriented approach to algorithm visualization - easy, extensible, and dynamic. Proceedings of the 25th SIGCSE Technical Symposium, Phoenix.
- Oman, P. (1986). Identifying student characteristics influencing success in introductory computer science courses. AEDS, 19(2-3), 226-233.
- Papert, S. (1980). MindStorms: children, computers, and powerful ideas. New York: Basic Books.
- Pattis, R. (1981). Karel the robot. New York: Wiley.
- Pausch, R. (head), Burnette, T., Capeheart, A.C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., & White, J. (1995). Alice: Rapid prototyping system for virtual reality. IEEE Computer Graphics and Applications.
- Rodger, S.H. (1996). Integrating animations into courses. Proceedings of the Conference on Integrating Technology into Computer Science Education, Barcelona, Spain, 72-74.

Scragg, G., Baldwin, D., & Koomen, H. (1994). Computer science needs an insight-based curriculum. Proceedings of the 25th SIGCSE Technical Symposium, Phoenix, 150-154.

Shu, N.C. (1988). Visual programming. New York: Van Nostrand Reinhold Co.

Soloway, E.M. (1986). Learning to program = learning to construct mechanisms and explanations. Communications of the ACM, 29, 850-858.

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

Stephens, L., Konvalina, J., & Teodoro, E. (1985). Procedures for improving student placement in computer science. Journal of Computers in Mathematics and Science Teaching, 4(3), 46-49.

Ungar, D. & Smith, J. (1987). SELF: The power of simplicity. OOPSLA 87, Conference Proceedings, published as SIGPLAN Notices, 22, 12, 227-241.