Objects: Visualization of Behavior and State

Wanda Dann*
Toby Dragon
Computer Science Dept.
Ithaca College
Ithaca, NY 14850
1-607-274-3602
wpdann @ ithaca.edu

Stephen Cooper*
Kevin Dietzler
Kathleen Ryan
Computer Science Dept.
Saint Joseph's University
Philadelphia, PA 19131
1-610-660-1561
scooper@ sju.edu

Randy Pausch
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
1-412-268-3579
pausch @cs.cmu.edu

ABSTRACT

Animated program visualization can be used to support innovative instructional methods for teaching beginners about objects, their behavior, and state. In this paper, we present a discussion of methods that define object behavior and character (class)-level state variables that track state changes for 3D animated objects in small virtual worlds. We have found that character-level methods provide a means to demonstrate inheritance. Examples of worlds and program code used in instructional materials are provided.

Categories and Subject Descriptors
K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education.*

General Terms
Documentation, Human Factors

Keywords
Visualization, Animation, 3D, Objects, Behavior, State

1. INTRODUCTION

This paper presents an instructional approach that uses animated program visualization as a technology to support teaching beginning programmers about behavior and state in object-oriented programming. The key to object-oriented programming is, of course, the *object*. In *Object-Oriented Design with Applications*, Booch (one of the original text writers on object-oriented programming) wrote: "An object has state, behavior, and identity..." From a theoretical perspective, the concept of identity is difficult. But, students are familiar with naming things. So, teaching students to name objects presents little difficulty (at least not until objects are passed as parameters). However, the concepts of behavior and state, as they apply to objects, present particular challenges to the instructor of introductory courses. Behavior is challenging because it is more complicated than method/function calls in an imperative language. Some methods are associated with certain objects while other methods seem not associated with an object at all (such as static methods called from main in Java). State is challenging because objects require the use of the program heap for implementation. Drawing traditional memory maps to explain objects, variable scope, and how methods work requires both a stack and a heap. After a short lecture session, an instructor is likely to find she has created a confusing clutter of arrows and boxes on the board or projection screen. Student focus may well be on deciphering the maze of memory maps and the concept of *object* gets lost along the way.

* This work is supported in part by a grant from the National Science Foundation, DUE- 0126833.

The technology of animated program visualization offers a way to keep the focus on objects while teaching about behavior and state. In section 2, we describe the animated program visualization tool. Section 3 provides details and examples of world-level and character-level methods that define object behavior. Section 4 gives an interactive example illustrating the use state variables in a character-level method. Finally, we discuss some results of this approach, a discussion of other tools, and present our conclusion in sections 5 and 6.

2. PROGRAM VISUALIZATION TOOL

This section describes the visualization tool. (The knowledgeable reader may wish to skip ahead to the next section.) The tool used in our approach is Alice, a freely available 3D Interactive Graphics Programming Environment, developed at Carnegie Mellon University (CMU) under the direction of Randy Pausch [9]. A new Java-based version runs on the Windows operating system, with support for Macintosh, Linux, and web browser viewing of Alice worlds projected for early 2003. Alice is a rapid prototyping environment for 3D object behavior, designed to make it easy for novice programmers to develop interesting 3D animations and explore interactive 3D graphics.

In Alice, 3D models of objects (e.g., buildings, people, furniture, scenery) populate a virtual world. Alice programs, which have a strong object-oriented flavor, allow students to control the appearance and behavior of objects, have objects respond to mouse and keyboard input, or do any sort of computation that would normally be done in an introductory programming class. Alice supports trial and error as well as a designed approach to programming. Alice is also supportive of collaborative programming. Students are immediately able to see how their own animated program runs, affording an easy relationship of the program construct to the animation action. Figure 1 displays scenes from the execution of a typical virtual world. In this scene, a skater performs a traditional figure skating action.

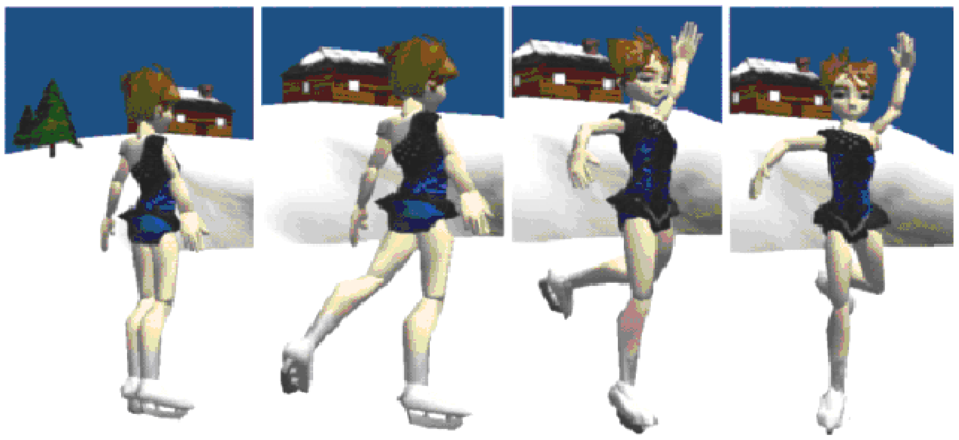


Figure 1. Animated Skater

The interface provides a smart editor for creating programs. Students drag-and drop program constructs (e.g., if, while) to form programs that are equivalent in expressive power to Java/C++/Pascal class languages. However, students are protected from making syntax errors by the programming environment, which only allows students to drag and drop tiles of program components to syntactically correct locations. For example, students may only drop an expression of type Boolean into the condition of a while loop. Full support is provided for all common control structures (e.g., if/then/else, while, for all) and data types (including 3D-object, Number, Boolean, String, etc.). Both array and list aggregate data structures are provided and students can write arbitrarily large methods and functions that take parameters of any type. Alice supports recursive method calls, limited polymorphism and a rich, interactive event structure for creating interactive worlds (programs) and characters (objects and classes that can be reused in other worlds/programs).

The design of Alice is driven by several key concepts: (1) *Make as much state visible to the student as possible*. Most changes are immediately visible (e.g. set the color of the frog from green to red.). (2) *Animate all changes of state*. Students are able to see the change of state, whenever possible, through an animation: objects that are told to change their position move through space; objects that are told to change their color animate through color space, etc. (3) *Do not allow ill-formed programs at any time*. Alice's sophisticated user interface allows students to build extremely complex programs that are always syntactically valid. (4) *Reify the notion of an object*. For many students, objects are very ethereal, abstract concepts. The objects in Alice are clearly visible on the screen: the Ice Skater object shown in Figure 1 is clearly an object in the 3D world and students find that creating and associating methods with such visible objects (such as Spin, Jump, and Bow) is more intuitively obvious. (5) *Use 3D graphics to engage the students*. The importance of this last point cannot be overemphasized.

3. VISUALIZATION OF BEHAVIOR

3.1Character-Level and World-Level Methods

Behaviors in an animated world are either *character-level* (equivalent to methods associated with objects of a given class in Java) or *world-level* (equivalent to static methods called from main in Java). Each character class (from a gallery of hundreds) has a built-in set of primitive behaviors the character object already knows how to perform (e.g., move, turn, roll). To motivate the introduction of programmer-defined character-level methods, students are asked to complete an exercise to "teach" a character a new behavior. For example, the skater in Figure 1 can be taught how to perform a simple spin movement (by combining several primitive instructions in a method). The visual nature of the behavior provides a context in which the instructor can discuss with students the concepts of methods and method calls.

A more difficult programming exercise, such as animating two characters in a dance as seen in Figure 2, is used to motivate the introduction of world-level behaviors. Students discover that creating a program of hundreds of primitive motion instructions makes their code not very readable, not easy to modify, and a problem when the characters need to perform the same dance action at several different times. Students learn that gathering a set of actions that collectively have meaning into a single method makes sense. They can then use those simple methods to build more complicated methods, as is illustrated in the two-step dance method of Figure 3, shown as it appears in the Alice interface. This dance situation provides a visual representation of a behavior that is world-level, rather than character-level, because it involves multiple characters.

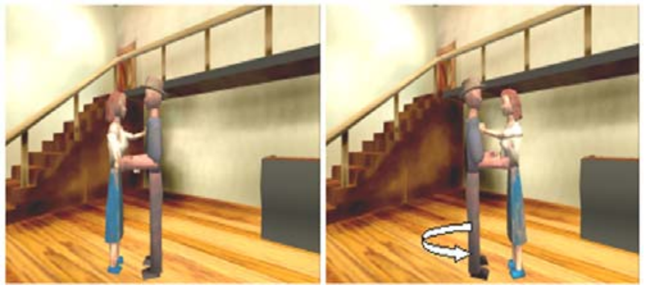
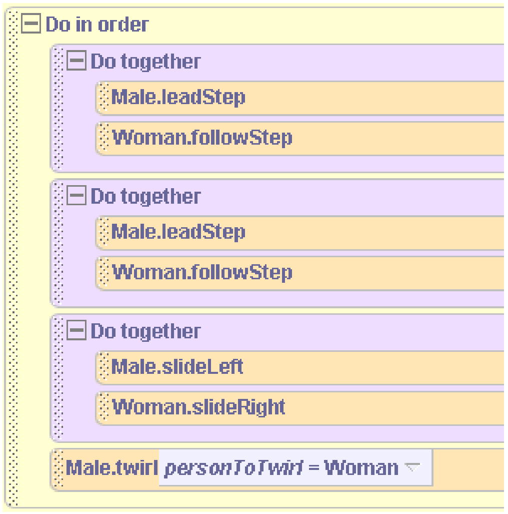


Figure 2. Scenes from a dancing world



```
Do in order
    Do together
        Male.leadStep
        Woman.followStep
    Do together
        Male.leadStep
        Woman.followStep
    Do together
        Male.slideLeft
        Woman.slideRight
    Male.twirl personToTwirl = Woman
```

Figure 3. Dance world-level method

3.2 Prototyping and Parameters

Importantly, character-level methods can be used to prototype new kinds of objects for re-use in other virtual worlds. By starting with a character from the gallery, adding functionality to the character, and then saving it back out to use in another world, students learn about inheritance. As an example, a Bunny character is used as a basis for prototyping a SuperBunny that knows how to *nibble* and *hop*. In the interest of brevity, only the nibble method is shown in Figure 4. (Note that code in Figure 4 and remaining figures is presented as simple text.) The SuperBunny character class of objects knows all the common primitive behaviors and also knows how to *nibble* and *hop*.

The character-level nibble method necessarily makes use of a parameter. The parameter *NibbleWhat* is used to specify the object (presumably a carrot or other edible object) to be nibbled. If the object nibbled were not passed as a parameter, the method would have to be world-level. This example provides a context for the instructor to discuss with students the need for parameters in character-level methods. When a SuperBunny object is created in a new world, the SuperBunny object will expect to have some edible object to nibble. But, it is not possible to guarantee that both the SuperBunny object and the original edible object will both be added to the newly built world. The nibbleWhat parameter ensures that the programmer will be prompted to provide an edible object for the SuperBunny. This use of visualization provides a context for students to learn that methods belonging to a particular type of object must use parameters to obtain information about other objects.

3.3 Functions

Behaviors discussed thus far may be viewed as methods that change the state but do not return values. Pure functions, in the form of questions, are introduced as a means of computing values. They are "pure" functions in that they do not change the state. Alice keeps an Algol-like distinction between the functional and imperative aspects of the language. Again, it is possible to have functions at both the world-level and at the character-level.

4. STATE VARIABLES

It is important to note that it is possible to teach behavior ***without*** requiring the use of mutable variables, where the student must ***explicitly*** manipulate the state of an object. Certainly an instruction such as Bunny.hop() changes the state (altering position) of the Bunny. But, the student who writes the hop() method does so by various calls to primitive move and turn methods. Access to private x, y, and z coordinates that define the object's translational position is not needed, as primitive methods are used to enable the desired behavior. This is, in spirit, quite similar to what was done with Karel the Robot [8] (one of the inspirations for Alice). Once students have mastered world-level and character-level methods then mutable character-level variables may be introduced.

It is perhaps easiest to justify the introduction of state variables (which, in our approach, are not introduced until after the student has mastered behaviors) by means of an example. This example makes use of the interactive capabilities of Alice, where methods can be written to respond to mouse or keyboard events. The problem is to simulate the motion of a car in a steering

```
SuperBunny.nibble
   NibbleWhat

   Do in order
      // Bunny hops to the object
      SuperBunny pointAt NibbleWhat onlyAffectYaw = true
      SuperBunny.HopAndMove HowFar = Bunny.distanceTo NibbleWhat * 1/4

   NibbleWhat move up SuperBunny.chest.head.whiskers distanceAbove nibbleWhat
          asSeenBy = SuperBunny

   SuperBunny.WiggleWhiskers

   Do together
        //nibble
        SuperBunny.chest.armL turn forward 1/4 revolutions
        SuperBunny.chest.armR turn backward 1/4 revolutions
```

Figure 4. Character-level method for SuperBunny

simulation, where the user can “drive” the car along the street. It is simple to turn the front wheels in response to a key press. The problem is that when the car moves forward, it is not easy to calculate how much right (or left) the wheels are turned. This is because once the car has started moving forward and the wheels have begun rolling, the orientation of the wheels is no longer what it was before the car started moving. Thus, it is effective to introduce a private, character-level variable named *turningAmount* to keep track of how much the wheels have been turned (right or left). The code for turning wheels to the right is shown in Figure 5. The private variable *turningAmount* is incremented by 1 each time the front wheels (wheel1 and wheel2) turn right. (The initial test is to keep the wheels from turning too much.) Figure 6 displays successive screen captures from the police car as it is steered in a right turn. The *moveForwards* method in Figure 7 demonstrates the use of the value stored in *turningAmount* to govern the degree of the turn (the police car turns right or left at the same time as it moves forward). Finally, all four wheels are rotated the appropriate amount. The variable *Wheels* refers to a list of the car’s wheels.

5. RESULTS

We have been teaching Alice to prospective computer science majors (who do not have previous programming experience) for the past three years. And, we are currently in the second year of an NSF funded study. Our impressive early results have been discussed elsewhere [3]. We do not duplicate that report in this paper. As part of our project evaluation, students are randomly selected to participate in an exit interview conducted by an evaluator. At the time of this writing, 25 students have been interviewed. One student's comments are particularly revealing: "I was taking CS1 at the same time as the Alice course. In the CS1 course, I was totally confused about objects and these things called methods. But I can see the objects in an Alice world and watch what the methods make the objects do. It helped me figure out what an object is and what methods are all about." These kinds of comments lead us to believe that students who experience this approach gain a fundamental understanding of objects and methods. We are following these students in their first year of study and have found preliminary evidence that students do especially well in CS1 in terms of dealing with objects and classes at a more abstract level [3]. A full compilation of results of this study will be forthcoming in a future paper.

6. OTHER TOOLS

BlueJ [7] is a visual tool used in teaching an objects-first methodology. BlueJ provides an integrated environment in which the user generally starts with a previously defined set of classes. The classes’ project structure is presented graphically, in UML-like fashion. The user can create objects and invoke methods on those objects to illustrate their behavior. Later the user may create his own classes, and BlueJ provides a graphical representation of these classes as well. While BlueJ does not

```
policecar.turnWheelsRight

If policecar.turningAmount < 10
  Do together
    increment policecar.turningAmount by 1
    policecar.wheel2 turn right .002 revs
           asSeenBy = policecar.wheel4
    policecar.wheel1 turn right .002 revs
           asSeenBy = policecar.wheel3

Else
    Do Nothing
```

Figure 5. turnWheelsRight increments *turningAmount*

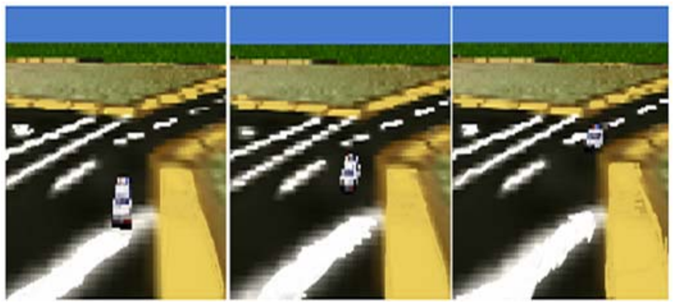


Figure 6. Steering the car right

```
policecar.moveForwards

distance

  Do together
    //Move police car forward and turn right or left
    //  amount specified by turningAmount
    policecar move forward distance meters
    policecar turn right 0.05 * policecar.turningAmount

    //turn all wheels forward at same time
    For all policecar.Wheels, one item_from_Wheels together
       item_from_Wheels turn forward policecar.rollAmount travelDistance = distance
```

Figure 7. Turningamount governs the degree of the right turn

provide 3D graphics or animation, it is being used effectively to help beginners learn about objects and their relations with other objects. Many of the examples provided with BlueJ do not keep novices from private mutable variables, though it would seem possible to create examples in which this separation was maintained.

Java Power Tools (JPT) [10] provides a comprehensive, interactive GUI, consisting of several classes with which the student will work. Students interact with the GUI, and learn about the behaviors of the GUI classes through this interaction. While GUIs may not provide the richness of interacting with 3D objects, students have grown up interacting with GUIs. Thus we believe that GUIs are a good example to use in helping students to develop object-oriented concepts. Like BlueJ, JPT starts with partially written programs that the student modifies. JPT uses students working with widgets to help them to develop intuitions about objects. While JPT does not have the widespread usage of BlueJ, we predict that once the collection of examples is enriched, that JPT will be successful as a means to teach beginners about objects.

7. CONCLUSION

We have experimented with the use of animated program visualization to support innovative instructional methods for teaching beginners about objects, their behavior and state. World-level methods (equivalent to static methods in Java) and character-level methods (equivalent to methods belonging to objects of a class in Java) can be visualized using animated virtual worlds. The 3D models provide a context for using methods to define object behavior. We have found that character-level methods provide a means to demonstrate inheritance. Further, character-level state variables that track state changes for 3D animated objects can be demonstrated in conjunction with the animations. We believe that the success of this approach is due to the visual representation of objects. Students can see and relate to the objects and their animation actions, thus developing good intuitions about objects and object-oriented programming.

8. REFERENCES

[1] Cooper, S., Dann, W., & Pausch, R. (2000) Alice: A 3-D tool for introductory programming concepts. In *Proceedings of the 5th Annual CCSC Northeastern Conference 200*0, Ramapo, NJ, April, 28-29.

[2] Cooper, S., Dann, W., & Pausch, R. (2003-1) Using animated 3d graphics to prepare novices for cs1. *Computer Science Education Journa*l, to appear.

[3] Cooper, S., Dann, W., & Pausch, R. (2003-2) Teaching objects first in introductory computer science. *SIGCSE 200*3, to appear.

[4] Dann, W., Cooper, S. & Pausch, R. (2000) Making the connection: Programming with animated small worlds. *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Educatio*n, Helsinki, Finland, July 11-13, 2000.

[5] Dann, W., Cooper, S., & Pausch, R. (2000). Using visualization to teach novices recursion. *Proceedings of the 6 th Annual Conference on Innovation and Technology in Computer Science Education, Canterbury, Englan*d, 109-112.

[6] Dann, W., Cooper, S. & Pausch, R. (2003) Learning to Program with Alice. To be published by Prentice Hall.

[7] Kölling, M. & Rosenberg, J., Guidelines for teaching object orientation with Java. In *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education* (Canterbury, England, June, 2001), 33-36.

[8] Pattis, R., Roberts, J, & Stehlik, M. *Karel the robot: a gentle introduction to the art of programming, 2nd Edition*. John Wiley & Sons, 1994.

[9] 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 Application*s, 15(3), 8-11.

[10] Proulx, V., Raab, R., & Rasala, R. Objects from the beginning – with GUIs. In *Proceedings of the 7th annual conference on Innovation and Technology in Computer Science Education* (Århus, Denmark, June, 2002), 65-69.