Teaching Objects-first In Introductory Computer Science

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@cmu.edu

Abstract
An objects-first strategy for teaching introductory computer science courses is receiving increased attention from CS educators. In this paper, we discuss the challenge of the objects-first strategy and present a new approach that attempts to meet this challenge. The new approach is centered on the visualization of objects and their behaviors using a 3D animation environment. Statistical data as well as informal observations are summarized to show evidence of student performance as a result of this approach. A comparison is made of the pedagogical aspects of this new approach with that of other relevant work.

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

General Terms
Documentation, Design, Human Factors,

Keywords
Visualization, Animation, 3D, Objects-First, Pedagogy, CS1

1 Introduction
The ACM Computing Curricula 2001 (CC2001) report [8] summarized four approaches to teaching introductory computer science and recognized that the "programming-first" approach is the most widely used approach in North America. The report describes three implementation strategies for achieving a programming-first approach: imperative-first, functional-first, and objects-first. While the first two strategies have been utilized for quite some time, it is the objects-first strategy that is presently attracting much interest. Objects-first "emphasizes the principles of object-oriented programming and design from the very beginning…. [The strategy] begins immediately with the notions of objects and inheritance….[and] then goes on to introduce more traditional control structures, but always in the context of an overarching focus on object-oriented design" [8, Chapter 7].

*This work was partially supported by NSF grant DUE-0126833

SIGCSE'03 February 19-23, 2003, Reno, Nevada, USA.

The Challenge of Objects-first: The authors of CC2001 admit that an objects-first strategy adds complexity to teaching and learning introductory programming. Why is this so? The classic instruction methodology for an introduction to programming is to start with simple programs and gradually advance to complex programming examples and projects. The classic approach allows a somewhat gentle learning curve, providing time for the learner to assimilate and build knowledge incrementally. An objects-first strategy is intended to have students work immediately with objects. This means students must dive right into classes and objects, their encapsulation (public and private data, etc.) and methods (the constructors, accessors, modifiers, helpers, etc.). All this is in addition to mastering the usual concepts of types, variables, values, and references, as well as with the often-frustrating details of syntax. Now, add event-driven concepts to support interactivity with GUIs! As argued by [11], learning to program objects-first requires students grasp "many different concepts, ideas, and skills…almost concurrently. Each of these skills presents a different mental challenge."

The additional complexity of an objects-first strategy is understood when considered in terms of the essential concepts to be mastered. The functional-first strategy initially focuses on functions, deferring a discussion of state until later. The imperative-first strategy initially focuses on state, deferring a discussion of functions until later. The objects-first strategy requires an initial discussion of both state and functions. The challenge of an objects-first strategy is to provide a way to help novice programmers master both of these concepts at once.

2 Instructional Support Materials
In response to interest in an objects-first approach, several texts and software tools have been published/developed that promote this strategy (such as [1, 12]). Four recent software tools are worthy of mention as using an objects-first approach: BlueJ [9], Java Power Tools [11], Karel J. Robot [2], and various graphics libraries. Interestingly, all these tools have a strong visual/graphical component; to help the novice "see" what an object actually is – to develop good intuitions about objects/object-oriented programming.

BlueJ [9] provides an integrated environment in which the user generally starts with a previously defined set of classes. The project structure is presented graphically, in UML-like fashion. The user can create objects and invoke methods on those objects to illustrate their behavior. Java Power Tools (JPT) [11] 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. Karel J. Robot [2] uses a microworld with a robot to help students learn about objects. As in Karel [10], Robots are

added to a 2-D grid. Methods may be invoked on the robots to move and turn them, and to have the robots handle beepers. Bruce et al. [3] and Roberts [13] use graphics libraries in an object-first approach. Here, there is some sort of canvas onto which objects (e.g. 2-D shapes) are drawn. These objects may have methods invoked on them and they react accordingly.

In the remainder of this paper, we present a new tactic and software support for an objects-first strategy. The software support for this new approach is a 3D animation tool. 3D animation assists in providing stronger object visualization and a flexible, meaningful context for helping students to "see" object-oriented concepts. (A more detailed comparison of the above tools with our approach is provided in a later section.)

3 Our Approach

Our motivation in researching and developing this new approach is to meet the challenge of an objects-first approach. Our approach meets the challenge by:

- Reducing the complexity of details that the novice programmer must overcome
- Providing a design first approach to objects
- Visualizing objects in a meaningful context

In this approach, we use Alice, a 3D interactive, animation, programming environment for building virtual worlds, designed for novices. The Alice system, developed by a research group at Carnegie Mellon under direction of one of the authors, is freely available at www.alice.org. A brief description of the interface is provided.

Figure 1. The Alice Interface

Alice provides an environment where students can use/modify 3D objects and write programs to generate animations. A screen-capture of the interface is shown in Figure 1. The interface displays an object tree (upper left) of the objects in the current world, the initial scene (upper center), a list of events in this world (upper right), and a code editor (lower right). The overlapping window tabs in the lower left allow for querying of properties, dragging instructions into the code editor, and the use of sound.

Student Programs: A student adds 3D objects to a small virtual world and arranges the position of each object in the world. Each object encapsulates its own data (its private properties such as height, width, and location) and has its own member methods. While it is beyond the scope of this paper to discuss all the details, a brief example is discussed below to illustrate some of the principles. Interested readers may wish to read [4, 6, 7] for a more complete description. Figure 2 contains an initial scene that includes a frog (named kermit), a beetle (ladybug), a flower (redFlower), and several other objects around a pond.

Figure 2. An initial scene in an Alice world

Once the virtual world is initialized, the program code is created using a drag-and-drop smart editor. Using the mouse, an object is mouse-clicked and dragged into the editor where drop-down menus allow the student to select from primitive methods that send a message to the object. A student can write his/her own user-defined methods and functions, and these are automatically added to the drop-down menus.

In this example, the task is for kermit to hop over to the ladybug. The code is illustrated in Figure 3. It is interesting to note that the built-in predicates ("Questions" in Alice-lingo) "is at least m meters away from n", "is within x meters of y", and "is in front of z" all return spacial information about the objects in question. (Users may define their own, user-defined, questions, at both the world-level as well as at the character-level.) The *bigHop(number n)* and *littleHop()* methods are both character-level. In other words, the basic frog class has been extended to create a frog that knows how to make a small hop and how to hop over a large object (receiving a parameter as to how high it must hop).

This example illustrates some important aspects of our approach. The mechanism for generating code relies on visual formatting rather than details of punctuation. The gain from this no-type editing mechanism is a reduction in complexity. Students are able to focus on the concepts of objects and encapsulation, rather than dealing with the frustration of parentheses, commas, and semicolons. We hasten to note that program structure is still part of the visual display and the semantics of instructions are still learned. A switch is used to display Java-like punctuation to support a later transition to C++/Java syntax.

Three-dimensionality provides a sense of reality for objects. In the 3D world, students may write methods from scratch to make objects perform animated tasks. The animation task provides a meaningful context for understanding classes, objects, methods, and events.

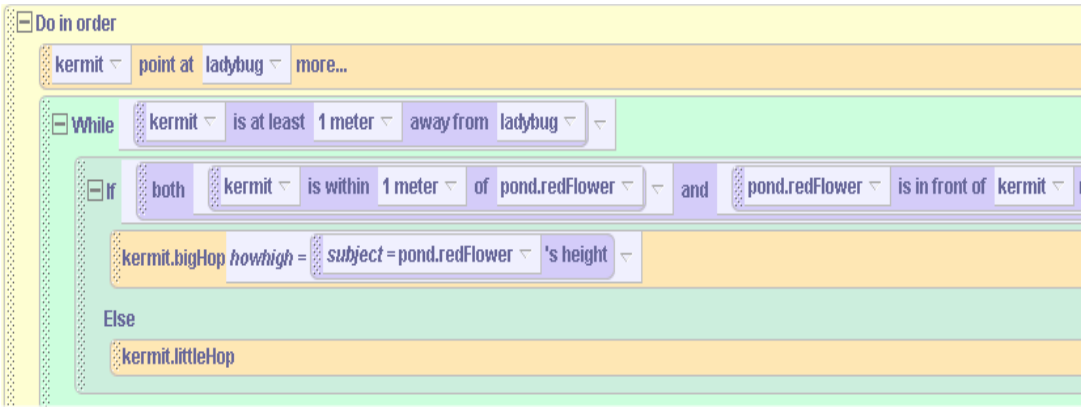

Figure 3. The code to have kermit hop over to the ladybug

4 Observations

We have been teaching and researching this new objects-first approach in an introduction to programming course for the past 3 years. One of the authors uses this approach in a ½ semester course that students take concurrently with CS1. Another author uses this approach as part of a course that students take before CS1. While early quantitative results are discussed in the next section, we present more informal observations in this section.

Strengths: We have seen that students develop:

- A strong sense of design. In our approach, we use storyboarding and pseudocode to develop designs. This may be influenced by the nature of our open-ended assignments. However, we see students in later classes writing down their thoughts about an assignment on paper first, before going to the computer.
- A contextualization for objects, classes, and object-oriented programming. We believe that this is one of the big "wins" for our approach. Everything in the student's virtual world is an object! Exercises and lab projects set up scenes where objects fly, hop, swim, and interact in highly imaginative movie-like simulations and games.
- An appreciation of trial and error. Students learn to "try out" individual animation instructions as well as their user-defined methods. Each animation instruction causes a visible change in the animation. Students learn to relate individual instructions, and methods to the animated action on the screen [7]. This direct relationship can be used to support development of debugging skills.
- An incremental construction approach, both for character (class)-level as well as world-level methods. Students do not write the whole program first. They program incrementally, one method at a time, testing out each piece.
- A firm sense of objects. The strong visual environment helps here.
- Good intuitions concerning encapsulation. Some state information can be modified by invoking methods on an object. For example, an object's position can be changed by invoking a *move* method. But the actual spatial coordinates that represent the object's position cannot be directly accessed.
- The concept of methods as a means of requesting an object to do something. The way to make an object perform a task is to send the object a message.
- A strong sense of inheritance, as students write code to create more powerful classes.
- An ability to collaborate. Students work on building the characters individually and then combine them to build virtual worlds and animations in group projects.
- An understanding of Boolean types. Students are prevented, by the smart-editor, from dragging incorrect data-type expressions into if statements and loops, for example.
- A sense of the program state. This is of particular importance, as mentioned earlier in this paper. This topic is discussed at length in [7].
- An intuitive sense of behaviors and event-driven programming.

One other observation is that it is possible to have students either create their programs from scratch or to build virtual worlds with characters which already have many specialized methods pre-defined. This latter case allows students to experiment with modifying existing classes/programs.

Weakness: A strength of our approach is also a source of weakness. Students do not develop a detailed sense of syntax, even with the C++/Java syntax switch turned on, as they only drag the statements/expressions into the code window. They do not get the opportunity to experience such errors as mismatched braces, missing semicolons, etc. Our experience with students making the transition from Alice to C++/ Java is that students quickly master the syntax.

5 Results

Table 1 illustrates the results of students at Ithaca College and Saint Joseph's University who took a course using our proposed approach during the 2001-2002 school year. The weakest 21 CS majors (defined as those CS students who were not ready for calculus and who had no previous programming experience) were invited to take a course using our approach, either concurrent with, or preliminary to CS1. 11 of the 21 students took the course,

while 10 did not. (Some students who did not take the course had scheduling conflicts.)

Statistics	All	Test	Control
# Students	49	11	10
Mean	2.49	2.8	1.3
Median	2.75	3	1.25
Variance	1.62	0.75	1.22

Table 1: Students taking Alice, 2001-2002

The results show that the 11 students who took the Alice-based course did better in CS1 than the total group, and significantly better than the 10 students who were of a similar background. Not only did the control group perform better in CS1, the lower variance indicates that a smaller percentage of those students performed poorly in CS1. Perhaps the most telling statistic is the percentage of students who continued on to CS2, the next computer science class. 65% of all the students who took CS1 continued on to CS2. Of the students in the test group (who took our course with Alice), 91% continued on to CS2. Only 10% of the control group enrolled in CS2. A larger group of students is being studied (in much more detail) this current (2002-2003) academic year, as part of an NSF supported study.

The authors have a textbook (to be published by Prentice-Hall for Fall 2003). An early draft is available at www.ithaca.edu/wpdann/alice2002/alicebook.html The URL for the solutions is available by contacting the authors. And, a set of lecture notes and sample virtual worlds is available at: http://www.sju.edu/~scooper/fall02csc1301/alice.html

6 Comparison with other tools

In this section we explore what we consider to be our relative strengths and weaknesses as compared to other object-first tools mentioned earlier. It is important to note that, as we have not seen studies detailing actual effectiveness of many of the other tools, we are hesitant to state too strongly the degree to which we think such tools do or do not work.

Events: JPT makes heavy use of GUIs, and both JPT and Bruce's ObjectDraw library rely on event-driven programming. Kölling and Rosenberg [9] state that building GUIs is "very time intensive", and argue that the GUI code is an "example that has very idiosyncratic characteristics that are not common to OO in general." Culwin [5] argues "the design of an effective GUI requires a wider range of skills than those of software implementation…. Even if an optimal interface is not sought at this stage it must be emphasized to students…that there is much more to the construction of a GUI than the collecting together of a few widgets and placing it in front of the user." While we might not go as far as these criticisms, it is clear that event handling does add a layer of complexity. In our approach, the use of events is optional and is accomplished through the use of several powerful primitives. This makes the presentation of events and event handling quite simple. We disagree with the statement "it is not possible to do Objects-first" without also doing GUI First!"[11], as both our approach and some of the graphics libraries do accomplish an object-first approach without the use of a GUI (though adding events generally makes virtual worlds much more fun for the students).

Modifying existing code: BlueJ and JPT depend on starting with programs that consist of previously written code. Bruce is concerned "these approaches will leave students feeling they have no understanding of how to write complete programs." The BlueJ and JPT authors maintain that, due to complexity of object-oriented design, it is favorable for novices to start with partially/completely developed projects and to modify them. Our approach allows the instructor to choose to use partially developed programs in introductory worlds. But, we generally have students build virtual worlds from scratch.

Use of the tool throughout the CS1 course: Each of these tools, with the exception of Karel J. Robot, is (or at least seems to be) capable of being used throughout the CS1 course. We have designed lecture materials to be used as an initial introduction to object-oriented programming, occupying the first 3-6 weeks of a CS1 course. It would be possible to intersperse the teaching of Alice with the teaching of, say, Java, throughout the semester.

Complexity of syntax: The use of graphics libraries is likely the most complex approach. Even though libraries are provided, students still must write Java/C++ programs from scratch, mastering a non-trivial amount of syntax (regardless whether they understand the semantics of what they are writing). Then they need to understand the particulars of the graphics library. Karel J. Robot has a fair bit of Java that needs to be mastered before being able to write a program. The BlueJ and JPT approaches are somewhat simpler, as students only modify existing code. Yet, it is still necessary to write correct Java code, and certain errors (such as missing brackets or trying to place code in the wrong location, or invoking a method with a bad parameter) can lead to errors in the code provided to the student -- and the student may not know how to start debugging code that he/she did not write.

Concurrency: As Culwin writes [5], "if an early introduction of GUIs is advocated within an object first approach, the importance of concurrency cannot be avoided." Alice supports concurrency, providing primitives for performing actions simultaneously.

Examples: This is a challenge for all objects-first approaches. Developing a large collection of examples (whether to be used as instructional aids, assignments or exam questions) is a time-consuming task that must be solved if these tools, together with their associated approach are to be successful. One product of our research efforts is a resource of examples, exercises, and projects with solutions. It does need to be made larger, which we are doing each semester.

7 Conclusions

The authors strongly believe that, as long as object-oriented languages are the popular language of choice in CS1, the objects-first approach is the best way to help students master the complexities of object-oriented programming. We believe that other tools mentioned here are quite useful in teaching objects-first. (We have used most of them ourselves.) We have been particularly impressed with the results we have seen so far with the approach we have presented here – we have been able to significantly reduce the attrition of our most at-risk majors. The current NSF study will examine the effectiveness of our proposed approach in greater detail, and with larger numbers of students. Additionally, we hope to gain feedback from some of the additional institutions that are using our materials and our approach.

References

[1] Arnow, D. and Weiss, G. Introduction to programming using Java: an object-oriented approach, Java 2 update. Addison-Wesley, 2001.

[2] Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. *Karel J. Robot a gentle introduction to the art of object oriented programming in Java*. Unpublished manuscript, available [August 31, 2002] from: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++Java Edition.html

[3] Bruce, K., Danyluk, A., & Murtagh, T. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education* (Charlotte, North Carolina, February, 2001), 6-10.

[4] Cooper, S., Dann, W., & Pausch, R. Using animated 3d graphics to prepare novices for CS1. *Computer Science Education Journal*, to appear.

[5] Culwin, F. Object imperatives! In *Proceedings of the 30th SIGCSE technical symposium on Computer Science Education* (New Orleans, Louisiana, March, 1999), 31-36.

[6] Dann, W., Cooper, S., & Pausch, R. Using visualization to teach novices recursion. In *Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education* (Canterbury, England, June, 2001), 109-112.

[7] Dann, W., Cooper, S., & Pausch, R. Making the connection: programming with animated small worlds. In *Proceedings of the 5th annual conference on Innovation and Technology in Computer Science Education* (Helsinki, Finland, July, 2000), 41-44.

[8] Joint Task Force on Computing Curricula. Computing Curricula 2001 Computer Science. *Journal of Educational Resources in Computing (JERIC)*, 1 (3es), Fall 2001.

[9] 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.

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

[11] 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.

[12] Riley, D. *The object of Java: Bluej edition*. Addison-Wesley, 2002.

[13] Roberts, E. & Picard, A. Designing a Java graphics library for CS1. In *Proceedings of the 3rd annual conference on Innovation and Technology in Computer Science Education* (Dublin, Ireland, July, 1998), 213-218.