FEATURE

Using Graphics in an Introductory Computer Science Course

A. T. Chamillard, Jason A. Moore, and David S. Gibson
U.S. Air Force Academy

Abstract

One of the major obstacles facing instructors of introductory computer science courses is motivating the students to excel while still covering the foundational programming topics. This is especially true in courses taught to both computer science majors and non-majors. We believe that having students include graphics in their programs can enhance student motivation, and in this article, we describe how we have incorporated graphics into an introductory computer science course taught to both majors and non-majors. We also present anecdotal and empirical evidence indicating that our efforts have been successful in motivating students. Although this work is presented in the context of an undergraduate freshman-level course, we note that the approach can also be used to enhance student motivation in high school programming courses.

One of the major challenges for instructors of introductory computer science courses, both at the high school and undergraduate levels, is finding ways to motivate students to excel and to help them enjoy the course. Of course, any techniques used to provide this motivation and enjoyment must be selected carefully so that students still learn the foundational material required as they begin their formal programming education. The motivation problem is exacerbated in those courses containing both computer science majors and non-majors. The approach described here is applicable to high school programming courses, undergraduate courses for computer science majors and non-majors, and courses for computer science majors only.

All students attending the U.S. Air Force Academy are required to take an introductory course in computer science (Computer Science 110). Because the course is taken by all students in either their freshman or sophomore year, it assumes no prior knowledge about computers. The key topic in this course is problem solving with computers. Because students need to know how to solve problems before they can solve them using computers, we start by helping the students develop their problem-solving skills. They then learn how to use these skills to solve problems using computers and the Ada programming language.

To evaluate the students' programming ability, we use a number of evaluation tools in the course. Specifically, students are evaluated on collaborative labs, individual programming assignments, a group case study, tests, and a final examination. In the past, all the evaluation techniques except the case study have been text oriented. For the case study, which typically has been some form of game-playing program, we have provided all the graphics required; students have simply written input, processing, and output subprograms without being concerned with how the graphics worked. Starting in the fall 1999 semester, we have incorporated graphics into all our student programming assignments and other assessment techniques.

One of the primary reasons for doing this is to provide motivation for students as they try to learn new, sometimes difficult, concepts. The students have enjoyed the graphical capabilities of the case study code we have provided to them in the past, so it seemed reasonable to let them try some of the graphics input and output on their own. Though we have only recently instituted this change, anecdotal evidence seems to support the motivational benefits of incorporating graphics into the course. For example, some instructors have received applause in class simply by showing students how to write a program that draws a circle to the screen! In addition, some students have even written simple graphical animations on their own because of their excitement about their ability to generate graphics.

Research has also shown that different people learn new material in different ways, based on their learning style (Chamillard & Karolick, 1999). Because some students learn new material visually rather than verbally, developing graphical solutions to the problem statements may help those students approach and learn the material more easily.

Others have also recognized the benefits of using graphics in early computer science courses. Roberts (1995) points out that students are much more enthusiastic about writing programs containing graphic functions, and graphics have been incorporated into general education computer science courses (Stegink, Pater, & Vroon, 1999) and more traditional CS1 courses (Astrachan & Roger, 1998). Panels have been convened to discuss the role of graphics across the computer science curriculum (Hunkins, 1993), and educational languages that are primarily based on simple graphic mechanisms have been developed (Clements & Meredith, 1992). This article provides a detailed description of how key

Table 1. Key Topics for Lab Assignments

Assignment	Key Topics
1	Variables, numeric input, graphical input and output
2	Condition-controlled looping, selection
3	Count-controlled looping, nested count-controlled looping
4	Subprograms (Ada procedures)
5	Arrays
6	File input/output

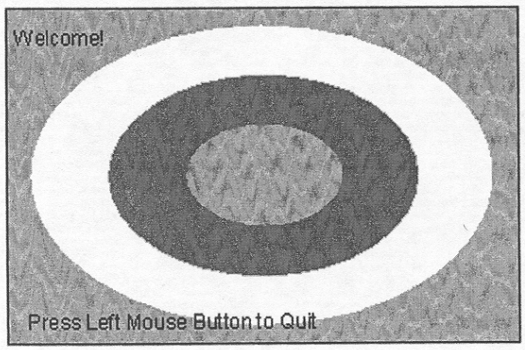

Figure 1. Example Lab 1 output.

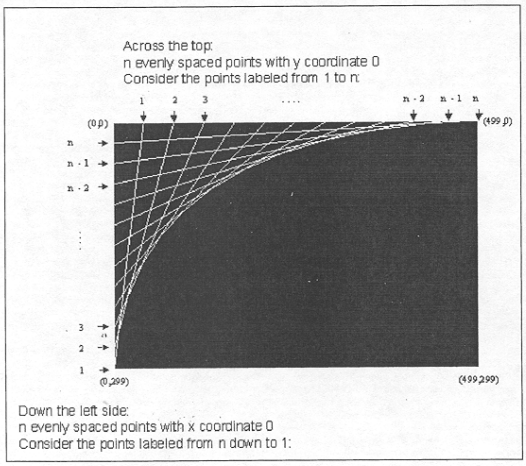

Figure 2. Example Lab 3 output.

programming topics can be covered using graphics and also provides anecdotal and empirical evidence supporting the assertion that including graphics improves student motivation.

The next section presents the topics covered in each of our labo assignments and discusses how we have incorporated graphics to help teach and evaluate those topics. The section on Risks discusses some of the risks associated with incorporating graphics into this course, and the Performace Comparison section presents a comparison between student performance on a text-oriented assignment from fall 1998 and a graphics-oriented assignment from fall 1999. The final section presents our conclusions.

Lab Descriptions

There are six lab assignments in this course. Table 1 provides an overview of the key topics covered by each of the labs.

Lab 1 introduces the students to variable declarations and provides them with practice doing both numeric input and graphical input and output. For this assignment, students read in a width and height for a graphics window and display a window of the appropriate size. They then display a text message in the window and draw different colored ellipses of appropriate sizes (90%, 60%, and 30% of the window size). Finally, they print an exit message in the window and wait for the user to press the left mouse button to exit the program. Figure 1 shows example output from this assignment.

Despite the simplicity of the assignment, students are exposed to a number of important concepts. Because they need to store the dimensions of the window, they learn about variable declarations and numeric input. They later need to calculate appropriate ellipse sizes, which introduces them to assignment statements and the importance of data types. They also gain experience displaying both text and graphics (the ellipses) in the graphics window, as well as using graphics input to determine when the left mouse button is pressed.

In Lab 2, students use a condition-controlled loop to ensure the user enters a valid width for a graphics window. They then open a graphics window and use another condition-controlled loop to continuously draw a circle at the current mouse location until the user presses a key to quit the program. The circle must be a different color for each quadrant of the graphics window; the most common implementation is to use a selection (if) statement to determine the appropriate mouse color.

Students are introduced to count-controlled loops (and nested count-controlled loops) in Lab 3. In this assignment, they get a valid number of lines (n) from the user, open a graphics window, and draw n lines in the window as shown in Figure 2. They then count the number of non-black pixels in the window, display the count in a text window, and exit the program when the user presses the mouse button.

In addition to requiring students to write their first count-controlled loop, this assignment provides a requirement that motivates the use of nested count-controlled loops. In the past, we have had difficulty providing students with a realistic problem that required such nesting, but scanning the x and y coordinates of the graphics window to count non-black pixels is well suited to this construct.

Lab 4 is the first assignment in which the students write their own subprograms (Ada procedures). Students write a game program in which a ball randomly bounces in a graphics window while the user tries to place a box (by clicking the left mouse button) in front of the ball as it bounces. Each time the ball passes through

the box, the user scores a point. The game ends after approximately 30 seconds, at which point the user's score is displayed.

This assignment gives students their first experience creating an animation while also providing a motivational vehicle for using subprograms. Many students enjoy playing computer games and find playing a game they implemented themselves even more rewarding.

In Lab 5, students create an animation in which lines are repeatedly drawn between two points that are moving across the screen in independent directions. The assignment requires that the points "bounce" off the edges of the window and the number of lines on the screen never exceed 20. In other words, when the program is ready to draw the 21st line, it needs to erase the oldest line first, then draw the new line.

This assignment provides an entertaining animation while also requiring students to make effective use of arrays and subprograms. For example, one reasonable use of arrays stores the end points for 20 lines, erasing the oldest line and storing new end points in that location when the 21st line needs to be drawn.

Lab 6 integrates file input and output with further use of arrays. The students write a program that reads in a list of data from a file and finds numerous data points of interest (high and low data points, for example). The program then scales a graphics window appropriately (so that the high and low values will be displayed near the top and bottom of the window), displays all the data in the window, and prints a formatted report to an output file.

Risks

Although we believe the lab assignments discussed have include graphics in a logical manner, there are clearly some risks associated with incorporating graphics into this course. For example, the students could become so engrossed with the graphics that they overlook important programming concepts. Similarly, students could become entangled in the syntax required to use graphics routines to the detriment of more general topics. Though the discussion above and the key topics listed in Table 1 indicate that we have not missed key concepts by including graphics, the graphical syntax issue requires further discussion.

Though we would like our students to be able to recognize and process certain events (e.g., a key being struck, the mouse being moved, or one of the mouse buttons being clicked), we want to avoid having the students process low-level hardware events in their programs. We have therefore added a layer of abstraction between those hardware events and the interface the students use to process the events.

For example, rather than having the students try to process both a left mouse press and a left mouse release to determine where the mouse was when the left mouse button was clicked, we provide a subprogram called Get_Mouse_Button. An example call to the subprogram appears in Figure 3. In this call, X_Location and Y_Location will be the coordinates of the mouse when the left button is clicked (the right button can also be checked, of course). We provide similar abstractions for determining the mouse location (without a button click), determining whether a particular mouse button was clicked, determining whether a keyboard key was hit, and determining which keyboard key was hit. By abstracting these low-level hardware events to a higher level, we have made it relatively easy for the students to understand and process these events without having them deal with the underlying complexity.

Of course, this approach can also be used if the course uses a programming language other than Ada. For example, Java provides a package to support graphics processing—the java.awt package—but using this package requires that the students use more advanced constructs than we would like to require in an introductory course. For example, we could certainly teach the students to process low-level events such as mouse clicks, key strokes, and so on using event listeners or adapters, but we believe providing a more abstract interface to mouse and keyboard actions is a better approach for beginning programmers.

Another risk involves using graphics early in the course, because even the first lab assignment includes graphical input and output. The topics of subprograms and parameters historically cause problems for our students, so some risk is associated with requiring students to use subprograms and parameters so early in the course. To help minimize this risk, we give the students access to a Web page that contains templates for calls to all the graphical subprograms they would use in the course. Students can simply copy the appropriate call into their program, then complete the call by filling in the actual parameters. Though students clearly do not have an understanding of how the actual parameter passing works until later in the course, they are able to correctly use the subprogram calls given the provided templates.

One way to determine whether the risks mentioned here affect student performance is to examine student grades on the lab assignments to determine whether the students do better on text-oriented or graphics-oriented assignments covering similar key topics. We provide such a grade comparison in the next section.

Performance Comparison

Because the order of topic presentation has been changed significantly in the course, we only compare student grades on Lab 1 from the fall 1998 and 1999 semesters. In both semesters, the key topics for Lab 1 were (a) variable declarations and use and (b) data input and output. For Lab 1 in 1998, the students wrote a program that read in a set of numeric data from the user, converted the data to the metric system, and output the data; Lab 1 for 1999 was described earlier. The only significant difference between the 1998 and 1999 assignments is the addition of graphics in 1999; a comparison between the two is therefore reasonable and enlightening. We begin by comparing the means for the two semesters, then we

```
Get_Mouse_Button (Left_Button, X_Location,Y_Location);
```

Figure 3.

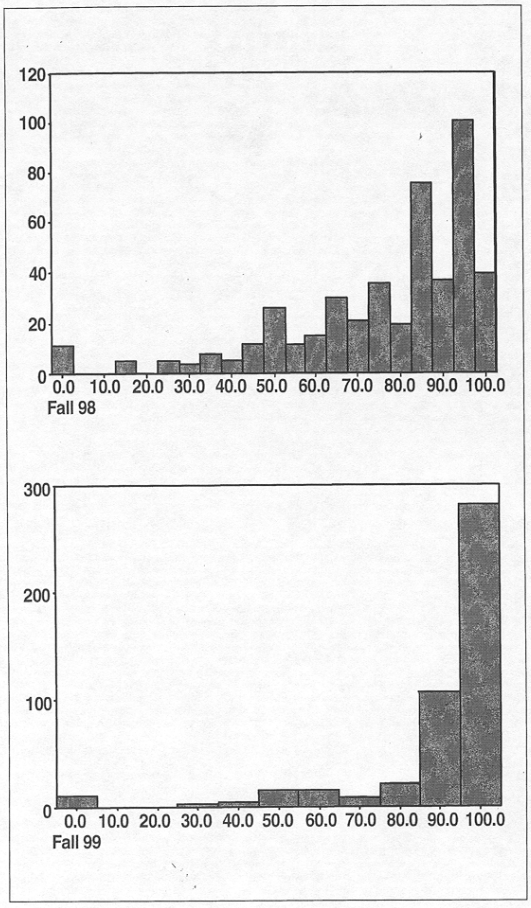

Figure 3. Grade histograms.

consider another measure that may provide some insight into the motivational effects of using graphics in the assignments.

The mean score on Lab 1 in fall 1998 was 75.7%, with a standard deviation of 23.2%. The mean in fall 1999 was 87.8 %, with a standard deviation of 19.7%. Though these means are clearly different, we would have liked to run an independent-samples *t*-test to quantify the significance of the difference. However, Kolmogorov-Smirnov tests and graphs of the grade distributions (Figure 3) indicate that the distributions are not normal; therefore, a *t*-test would not be an appropriate comparison statistic for the means. Informally, however, it is clear that the students performed much better on the graphics-oriented assignment (1999) than on the text-oriented assignment (1998). Although it would be interesting to determine whether 1998 Lab 1 data represent "typical text-oriented data," Lab 1 data from preceding years is not available for comparison.

Though the grade comparison presented here shows that incorporating graphics has improved student performance on Lab 1, we would still like to quantify the motivational benefits we are reaping by including graphics. To do this, we suggest that, if graphics-oriented labs are more motivational than text-oriented labs, fewer students will "give up" as they try to complete the assignment.

We therefore counted the number of students who received a grade of less than 33.3% in each semester. We recognize that this may also capture students who did not give up and were simply unable to master the material, but we suspect this was a small percentage of the students who received such low grades on the assignment. We also note that choosing 33.3% for our threshold is somewhat arbitrary, because we could have also used 25%, 50%, or some other percentage as our threshold; we simply chose a threshold that seemed reasonable.

In fall 1998, 39 of 528 students (7.39%) scored lower than a 33.3% on the assignment. In contrast, in fall 1999, only 14 of 467 students (3.00 %) scored lower than 33.3%. Though we recognize that this metric is at best an indirect measure of motivation, we do believe that the difference between the text-oriented and graphics-oriented assignment grades provides further support for our conclusion that incorporating graphics is beneficial.

Conclusions

Incorporating graphics into our introductory course has led to significant changes in the lab assignments in the course, though the assignments continue to cover the same set of key programming concepts. The preliminary empirical evidence presented in this article indicates that these changes have led to improved student performance and motivation.

This approach is not without risk, however, as we discussed earlier. It is not surprising, therefore, that before fall 1999, numerous faculty members expressed concerns about incorporating graphics into this course. What is surprising is that most of those faculty members have since approached the director of the course and expressed enthusiasm for the effects of that change.

Including graphics in the student programming assignments also helps students better understand the operation of other software. For example, most students understood before enrolling in this course that clicking an icon in Windows will start the appropriate program. With the inclusion of graphics input, specifically processing mouse clicks, we can now explain how such actions can be taken using operations they have already included in their own programs. This serves to help demystify the operation and implementation of graphical user interfaces.

Though the discussion and empirical results presented in this article address a course that uses Ada as its programming language, the approach and the motivational benefits that can be achieved from using the approach are clearly language independent. In addition, the approach can be applied to high school and undergraduate courses because the interfaces to the graphical routines do not require student use and understanding of more advanced programming constructs. The students effectively use

graphics in their first assignment, then continue to develop more complicated graphics-based programs as they learn more advanced programming constructs. The approach described in this article, therefore, represents a general approach that can be used independent of the programming language and the grade level of the students in the course.

References

Astrachan, O., & Roger, S. H. (1998). Animation, visualization, and interaction in CS 1 assignments. In J. Lewis, J. Prey, & D. Joyce (Eds.), *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education* (pp. 317–321). New York: Association for Computing Machinery.

Chamillard, A. T., & Karolick, D. (1999). Using learning style data in an introductory computer science course. In R. E. Noonan, J. Prey, & D. Joyce (Eds.), *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education* (pp. 291–295). New York: Association for Computing Machinery.

Clements, D. H., & Meredith, J. S. (1992). *Research on Logo: Effects and efficacy* [Online document]. Available: http://el.www.media.mit.edu/groups/logo-foundation/pubs/papers/research_logo.html.

Hunkins, D. R. (Moderator). (1993). Computer graphics across the CS curriculum. In *Papers of the 24th SIGCSE Technical Symposium on Computer Science Education* (p. 295). New York: Association for Computing Machinery.

Roberts, E. S. (1995). A C-based graphics library for CS1. In C. White (Ed.), *Papers of the 26th SIGCSE Technical Symposium on Computer Science Education* (pp. 163–167). New York: Association for Computing Machinery.

Stegink, G., Pater, J., & Vroon, D. (1999). Computer science and general education: Java, graphics, and the Web. In R. E. Noonan, J. Prey, & D. Joyce (Eds.), *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education* (pp. 146–149) New York: Association for Computing Machinery.

Contributors

Dr. A. T. Chamillard is an associate professor of computer science (currently on sabbatical) at the U.S. Air Force Academy. His research interests include computer science education, the relationships between learning styles and computer science assessment performance, and the use of Ada in introductory computer science courses and courses for non-majors.

A. T. Chamillard
PO Box 384
Fort Belvoir, VA 22060
achamillard@hq.dcma.mil

Dr. Jason A. Moore, formerly an associate professor of computer science at the U.S. Air Force Academy, now manages information technology projects for the NATO Early Warning Component at Geilenkirchen, Germany. His research interests include computer networks, information warfare, and computer science education.

Jason A. Moore
PSC 7 Box 3
APO AE 09104
moorej@cs.usafa.af.mil

Dr. David S. Gibson is an assistant professor of computer science at the U.S. Air Force Academy. His research interests include reusable software components, programming languages, computer architecture, computer science education, and Web site maintainability.

David S. Gibson
Department of Computer Science
U.S. Air Force Academy, CO 80840
David.Gibson@usafa.af.mil

OOPs! continued from p. 7

Although Visual Basic does not support object inheritance, it does allow class composition. By using this simple OO technique and the ActiveX Control Interface Wizard, it is a fairly simple task to create a custom control that inherits the essential functionality of an existing control, but adds additional properties and methods that are desired for specific applications. The use of Property methods allows these custom controls to mimic the standard controls in every way.

Bakopanos presented a basic solution to the problem of modifying a standard control to achieve custom functionality and appearance, but his solution fell short of its potential. It required special code to be written at the application level rather than hiding it within the control code itself. Requiring this additional code violated the basic OO principle of information hiding and resulted in a solution that was inflexible and prone to error. In addition, by separating the custom code from the object to which it was to be applied, he also violated principles of encapsulation. The solution developed in this article, applied class composition to provide a truly OO approach that illustrated the principles of information hiding and encapsulation. We also saw how the ActiveX Control Interface Wizard could be used to imitate inheritance, even though it is not a feature of the Visual Basic language.

As a final note, regarding object reuse, the Visual Basic IDE provides a method of packaging this new control to make it instantly available to any project. By compiling the CustomTextBox control to create an ActiveX Control file (CustomTextBox.ocx), it can be included in the project components of any new project. By simply dropping an instance of this control on a form, we get all of the desired custom functionality, plus the added benefit of modifying any of its properties or methods on an individual control basis.

Reference

Bakopanos, K. (2000, February). Applying functionality on controls the OO way. *Visual Basic Online* [Online serial]. (Note: At press time, this periodical was no longer available online.)

Contributor

Dr. Margush has been teaching computer science for over twenty years. His main interests are in the area of discrete mathematics and software development in Visual Basic, C++, and Java. He has authored several custom packages for various nonprofit organizations and engaged in consulting activities with several firms. Tim is active in community organizations, and has appeared in several drama productions through his church. When he is not thinking about programming, he enjoys spending time with his family or singing and playing guitar.

Dr. Tim Margush
Department of Computer Science
The University of Akron
Akron, OH 44325
Margush@uakron.edu
www.cs.uakron.edu/~margush 330.972.7109