# Active Visual Scaffolding

Charles F. Kelemen and Eugene R. Turk
Swarthmore College

## I. Introduction

Reading code and viewing algorithm animations are helpful in understanding data structures and algorithms. However, neither of these is a substitute for writing code to implement an algorithm or proving theorems about algorithms. Watching a visualization is a fairly passive process. Most visualization systems do not allow students to experiment with changing the underlying code and seeing an updated visualization. Interactive debuggers allow students to follow the execution of their own code but often at too low a level of abstraction with too much detail.

We have developed a prototype of a Java graphical infrastructure that will allow students to follow the execution of their code at a level of abstraction somewhere between that offered by a typical algorithm animation and a typical interactive debugger. This infrastructure requires a more active role on the part of the student than either an interactive debugger or an algorithm animation, potentially resulting in a greater understanding of the underlying concepts of a student's program. It is easy enough to master that it will not require the removal of any content from a first year CS course, which would be the natural place for a project like this at this stage in its evolution. It is our hope that first year students will enjoy using this graphical tool to reveal information about their programs as they run them, and thus speed the debugging process, in addition to providing useful information about the actual way in which data in manipulated "behind the scenes." Ultimately, we plan on extending the ideas of this prototype in such a way as to accommodate more complex data structures than the initial one we chose, and also to make it so that students will be able to create and use their own library of graphical functions that interface with our infrastructure to animate their algorithms in a more customizable manner.

## II. Background

Much work has been done on Software Visualization [38]. John Stasko has experimented with providing students with visualization software and having students write code for algorithm execution and visualization using his graphics libraries [37,39]. Students must learn commands that are outside their programming language environment and use a package that is a bit like a 'black

box' in order to see these animations. Additionally, changes to source code will not be reflected in visualizations without explicit changes to the visualization calls. What we have done draws heavily upon Stasko's pioneering work. However, by using Java as the student programming language and the visualization language, we hope that students will be starting with an 'opaque box' that will become a 'clear box' by the end of the course. By putting our graphical interface at the disposal of the student and leaving the actual creation of the custom visualization methods open, the student will have to understand the visualization process on an increasingly intimate level the farther he or she progresses in writing complex algorithms. In leaving the visualization program entirely behind the scenes as Stasko and others have done, students can never achieve the degree of interactivity with their visualizations required for some of the more complex data structures and algorithms that are taught in CS courses. In using our infrastructure, students not only gain a tool with which to debug their programs and see their algorithms “in action,” they ultimately gain an understanding of the algorithm visualization process itself through constantly interacting with the interface we provide.

In "The Mythical Man-Month", Frederick Brooks says, "Build plenty of Scaffolding. By scaffolding I mean all programs and data built for debugging purposes but never intended to be in the final product." [10] One extremely effective kind of scaffolding is the “extra” print statement. Consider the following advice to students:

> When given a problem for computer solution, many students (especially smart students) rush to the computer and start entering code. They write the whole program; then try to compile and run the program. This works for easy exercises at the beginning of a CS1 course. For some students it may even work for programs that require 50-100 lines of code. But at some point it fails for all. At that point (let's say a program of 100 lines of code) programming can become very frustrating. With 100 lines of code, even syntax errors may be difficult to find. But the more insidious logical errors could be anywhere in 100 lines. It becomes very frustrating to think you have discovered an error only to have the program fail over and over.
>
> Do not code programs in one step.
>
> The first thing to do is think about the overall design of a solution away from the keyboard. Consider both top-down and bottom-up design. Once you know about object-oriented ideas, you should ask, "What are reasonable objects?", "What kind of communication should take place

between objects?", "What should be public, private, protected?", "How can I exploit inheritance?", "What are good superclasses?", etc. Once you have an outline of the overall design, you can begin to think about coding. You can code in small steps whether you choose to code in a top-down manner (use stubs) or in a bottom up manner (use simple 'driver' programs). Take many small steps compiling and testing at each step. Each time a small step works you get a little Computer Science High (these are absolutely legal everywhere). To avoid gigantic downers, develop your programs in small steps so you get lots of highs.

Start with a very small (like 5-15 lines of code) program that you compile and run. Then add at most 5-15 lines; compile debug and run. At each iteration, be certain that your code so far is CORRECT. In order to accomplish this you need to add 'scaffolding'. This is code that may not appear in the final product, but allows you to be certain that your current code is correct so far. Figuring out good scaffolding goes hand in hand with loop invariants and really understanding how your program works. You should never have to spend hours until you get something to compile. You should always have a program that is no more than 15 lines different from your previous version (that you know runs perfectly and that you understand). Then when you have an error in your latest attempt, you can be pretty certain that your error is within 15 lines of code.

When you get an error you must find out why? If your program won't compile, try paying attention to the error messages (although they can sometimes be misleading). If necessary 'comment out' part of your code until you get something that does compile and then add things a bit at a time. If the program does compile but does not run correctly, your scaffolding should let you know where the problem occurs. If not, you need to add better scaffolding. Understand what is wrong before you try to fix it.

The above is extremely good advice. Some students take it but most do not. Extra lines of output are just not very exciting to most students. On the other hand, almost all students will work extremely hard on any assignment that uses graphics. We want to exploit the student excitement associated with graphical output to encourage them to include visual scaffolding in their program development.

## III. Development

For this work we assume students will be programming in Java. Utilizing Java 2D and Swing API's, we created an initial visual scaffolding infrastructure so that students will be able to think about issues of algorithm understanding and program correctness through the use of visual scaffolding in code development.

As a first step in the much larger task of generating a full infrastructure for students to use when displaying the majority of common algorithms and data types, we chose to use an array data structure as our testing grounds for the visualization concepts we wanted to illustrate. Arrays can be illustrated in real time through a very minimal addition of code in a student's program. There are drawbacks and benefits to this approach. Although the overhead for using the visualization is fairly minor, it masks the actual creation of the graphical object, which would certainly be useful for students to ultimately understand. Dealing with the sticky issue of to what degree students should be involved in the visualization process is truly challenging. It is difficult to know whether providing pre-written animation methods is a more useful tool than having students write them individually to fit their needs. For example, take a method that highlights two elements in the array and changes the background color to something other than the default. Even this very simple method may not have the flexibility that a particular student might need when attempting to debug his or her program. With that in mind, it may be a better approach to give the students more basic tools and have them write the animation functions themselves. At this early stage in development, however, it seemed to be more prudent to just write more complex animation methods that students could use in a smaller set of circumstances. Increasing the level of student involvement in the writing of their own methods that interface our GUI will certainly occur alongside the increase in complexity of their programs in CS 1. When students are first learning how to use arrays, however, such a level of interactivity is unnecessary and counterproductive to the students' understanding of algorithms and data structures.

After declaring a new visualization variable and constructing a visualization object, the student must initialize the array visualization so that it correctly points at the student's data array. Our array GUI shows the contents of each array element through the simple use of the common "toString" method that each object in the student's array will possess. If a student wants different aspects of an object's data displayed, he or she merely needs to override or replace the standard toString method with a more appropriate one for the task at hand. One or two calls to an update method at appropriate places within an algorithm that modifies the student's array will enable the visualization to change in "real" time as the

program executes. In order to avoid the too-rapid display of change in a student's visualization, the array GUI has a built-in slidebar that regulates the rate at which the student's program executes. The benefits of this method of visualization are that any errors in a student's algorithm are reflected in the visualization. Should a student make a logical misstep in the writing of a given algorithm, the animation will show a corresponding error as it occurs in the execution of the program. By building visual scaffolding into the objects and container operations, changes in source code will often be reflected in the visualization without requiring new calls to the visualization infrastructure.

## IV. Results

A sample use of this initial array GUI might be as follows. If a student were asked to implement a selection sort algorithm on a randomly generated array of Integer objects, he or she could additionally be instructed to utilize our graphical infrastructure for debugging and general educational purposes. Figure 1 shows the code that a student might write for such an assignment. "window" is the array GUI object. Note that the only calls to the "window" object within the student's algorithm occur in the line that compares two objects (the method "lessthan" highlights the two elements of the array at the indices specified, then returns whether the value stored at the first is less than the value stored at the second), and at the end of the principal loop ("window.redraw").

```
import java.util.Random;
public class StudentArray {

    static int arraysize = 10;

    static public Object[] generateArray(int size) {
        Object[] temp = new Object[size];
        Random generator = new Random();
        for (int i = 0; i < size; i++) {
            temp[i] = new Integer(generator.nextInt(100));
        }
        return temp;
    }

    public static void main(String args[]) {
        Object[] testArray = new Object[arraysize];
        Object temp = null;
        int lowpoint, left;

        testArray = generateArray(arraysize);

        ArrayGUI window = new ArrayGUI();
        window.inAnApplet = false;
        window.setArraySize(arraysize);
        window.setArray(testArray);
        window.initArray();
        window.pack();
        window.setVisible(true);

        lowpoint = left = 0;
        while (left != arraysize) {
            for (int i = left + 1; i < arraysize; i++) {
                if (window.lessthan(i, lowpoint)) {
                    lowpoint = i;}
            }
            temp = testArray[lowpoint];
            testArray[lowpoint] = testArray[left];
            testArray[left] = temp;
            left++;
            lowpoint = left;
            window.redraw();
        }
    }
}
```

**Figure 1**

The resulting initial and final states of the GUI when this code is executed are shown in figures 2 and 3, respectively.


**Figure 2**

The GUI highlights pairs of array elements as they are compared at a rate of speed slow enough for the student to see what is occurring in his or her algorithm, instead of having it execute too rapidly for any information to be gleaned. If there are logic errors within the student's algorithm, they should be fairly easily seen within the animation.

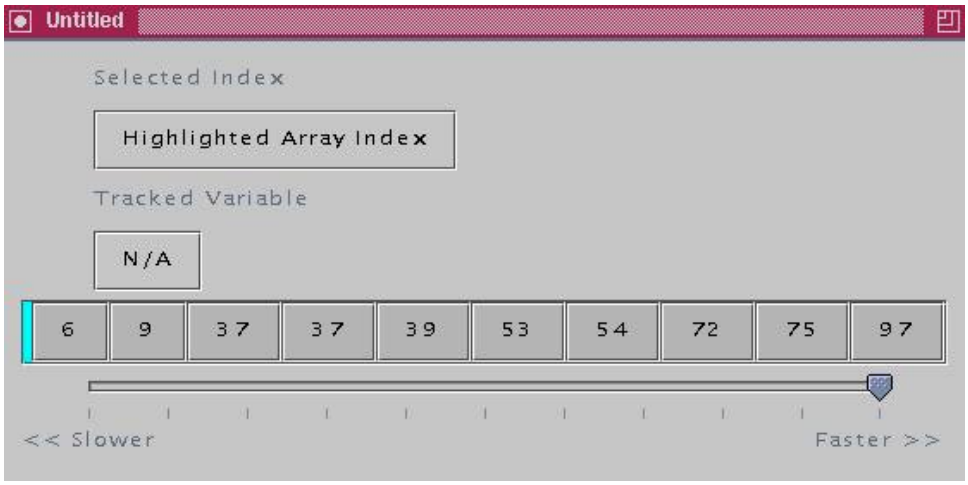


**Figure 3**

## V. Further Directions for Research

As stated above, our final plan is to create a sequence of exercises that lead students to develop their own graphics library while learning basic CS principles. We want the use of our tools to be so natural that their use will help in teaching and illustrating first year topics and not require the removal of any topics. Students will use their own library of graphical tools written through the help of our graphical user interfaces to write code for algorithm execution and visualization. Early in the process, students will be writing simple programs and relatively simple visualizations will suffice. As students get more sophisticated, the graphical interface will be able to accommodate that increase in complexity so that with careful thought, students will be able to provide visual scaffolding that will aid in understanding at higher levels of abstraction. Constructing the visualization will force students to come to grips with both the forest and the trees.

Some infrastructure aspects of visualization (such as zooming in and out, handling screen real estate, scaling, and handling the speed of the presentation) will be supplied by us and revealed to students as they progress. A good bit of our work will be designing and building this more complete infrastructure in such a way that it can be understood and used by students early and without taking time away from other important early topics. The existence of the Java 2D and Swing API's will allow us to provide this infrastructure at the source code level.

## VI. Conclusion

Judicious use of visual scaffolding will permit students to think and see the working of their programs at the level of their data structures and algorithms. Single stepping will not be needed often. When it is necessary to examine the program at the level of each line of code, visual scaffolding will work with and help set the context for single stepping in an interactive debugger like JDB. It is our hope that the graphical tools for visualization that each student develops in conjunction with our infrastructure will be used as an understanding and visualization aid in both first year courses and more advanced CS courses that the student takes. Our task, then, is to expand on the encouraging results we have obtained from implementing visual scaffolding for arrays to more complex data structures. There will be greater complexity in this project as more complex the data structures are represented. The underlying lessons we have learned from this prototype, however, will continue to hold true.

References

1.Baecker, R. M. (1968). Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures (Summary Only). In Proceedings of First Hawaii International Conference on the System Sciences, (pp. 128-129).

2.Baecker, R. M. (1975). Two systems which produce animated representations of the execution of computer programs. ACM SIGCSE Bulletin, 7: 158-167.

3.Baecker, R. M. & Buchanan, J. W. (1990). A Programmer's Interface: A Visually Enhanced and Animated Programming Environment. In Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences., (pp. 531-540). New York: IEEE Computer Society Press.

4.Baeza-Yates, R., Jara, L., & Quezada, G. (1992). VCC: Automatic Animation of C Programs. In Proceedings of COMPUGRAPHICS'92, (pp. 389-397).

5.Bentley, J. L. & Kernighan, B. W. (1991a). A System for Algorithm Animation. Computing Systems, 4(1): 5-30.

6.Bentley, J. L. & Kernighan, B. W. (1991b). A System for Algorithm Animation (Tutorial and User Manual) (Computing Science Technical Report No. 132). AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

7.Bentley, J. L. & Kernighan, B. W. (1992). ANIM. Murray Hill, NJ: AT&T Bell Laboratories. a collection of ANSI C programs used to visualize programs, runs on any UNIX computer. Available by anonymous ftp from research.att.com in /netlib/research.

8.Bergin, J, Brodie, K, Goldweber, M, Jimenez-Peris, R, Khuri, S, Patino-Martinez, M, McNally, M, Rodger, S, and Wilson, J, "An Overview of Visualization and its Use and Design", ITiCSE '96, Sept. 96.

9. Boroni, Christopher M., Frances W. Goosey, Michael T. Grinder, Jessica L. Lambert, and Rockford J. Ross. " Tying it All Together: Creating Self-Contained, Animated, Interactive, Web-Based Resources for Computer Science Education." Thirtieth SIGCSE Technical Symposium on Computer Science Education, volume 31, number 1, March 1999, pages 7-11.

10.Brooks, Frederick, "The Mythical Man-Month" Addison-Wesley, 1975

11.Brown, Cynthia, Harriet J. Fell, Viera K. Proulx, & Richard Rasala, Instructional Frameworks: Toolkits and Abstractions in Introductory Computer Science, Proceedings of the 1993 Computer Science Conference, ACM Press, February 1993, 195-200.

12.Brown, M. H. (1988a). Algorithm Animation. New York: MIT Press.

13.Brown, M. H. (1988b). Exploring Algorithms Using Balsa II. IEEE Computer, 21(5): 14 36.

14.Brown, M. H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. In Proceedings of IEEE Workshop on Visual Languages, (pp. 4-9). New York: IEEE Computer Society Press.

15.Brown, M. H. & Sedgewick, R. (1985). Techniques for Algorithm Animation. IEEE Software, 2(1): 28-39.

16.Byrne, Michael D, Catrambone, Richard and Stasko, John T., "Evaluating Animations as Student Aids in Learning Computer Algorithms," Computers & Education, Vol. 33, No. 4, 1999, pp. 253-278.

17.Byrne, Michael D, Catrambone, Richard and Stasko, John T., " Do Algorithm Animations Aid Learning?", Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-96-18, August 1996.

18.Domingue, John, Paul Mulholland: Staging Software Visualizations on the Web. VL 1997: 368-375

19.Domingue, John, Paul Mulholland: An Effective Web-based Software Visualization Learning Environment. Journal of Visual Languages and Computing 9(5): 485-508 (1998)

20.Haajanen, J, M. Pesonius, Erkki Sutinen, Jorma Tarhio, T. Teräsvirta, P. Vanninen: Animation of User Algorithms on the Web. VL 1997: 360-367

21.Kehoe, Colleen, Stasko, John and Taylor, Ashley " Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study",

Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-99-10, March 1999.

22.Lahtinen, S.-P, Erkki Sutinen, Jorma Tarhio: Automated Animation of Algorithms with Eliot. Journal of Visual Languages and Computing 9(3): 337-349 (1998)

23.Mukherjea, Sougata and Stasko, John T. , "Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding", Proceedings of the 15th International Conference on Software Engineering, Baltimore, MD, May 1993, pp. 456-465.

24. Mukherjea, Sougata and Stasko, John T., "Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger", ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3, September 1994, pp. 215-244.

25.Myers, B. A. (1980). Displaying Data Structures for Interactive Debugging (Technical Report No. CSL-80-7). Xerox PARC.

26.Myers, B. A. (1983). Incense: A System for Displaying Data Structures. Computer Graphics, 17(3): 115-125.

27.Naps, Thomas L., Norton, Laura L., and Eagan, James R., "JHAVÉ -- An Environment to Actively Engage Students in Web-based Algorithm Visualizations," in Proceedings of the SIGCSE Session, ACM Meetings (Austin, Texas, March 2000).

28.Naps, Thomas L., "Algorithm Visualization on the World Wide Web - the Difference Java Makes" in Proceedings of the Association for Computing Machinery's SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, Uppsala, Sweden, June, 1997

29.Naps, Thomas L., "Report of the Working Group on Using the World Wide Web as the Delivery Mechanism for Interactive, Visualization-based Instructional Modules" in Proceedings of the Association for Computing Machinery's SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, co-authored with Joe Bergin, Ricardo Jimenez–Peris, Marta Patino Martinez, Myles McNally, Viera Proulx, Jorma Tarhio, Uppsala, Sweden, June, 1997.

30.Naps, Thomas L. and Eric Bressler, " A multi-windowed environment for simultaneous visualization of related algorithms on the World Wide Web," in Proceedings of the SIGCSE Session, ACM Meetings (Atlanta, Georgia, February, 1998).

31.Naps, Thomas L., "A Java Visualizer Class: Incorporating Algorithm Visualizations into Students' Programs," in Proceedings of the Association for Computing Machinery's SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education, Dublin, Ireland, August, 1997

32.Pierson, W, and S. H. Rodger, Web-based Animation of Data Structures Using JAWAA, Twenty-ninth SIGCSE Technical Symposium on Computer Science Education, p. 267-271, 1998

33.Proulx, V, Fell, H, Rasala, R, and Brown, C, "Interactive Animations in Computer Science", Proceedings Frontiers in Education '93.

34.Rasala, Richard, "Automatic Array Algorithm Animation in C++", SIGCSE Bulletin, March 1999 257-260.

35.Rasala, Richard, "Toolkits in First Year Computer Science: A Pedagogical Imperative", Proceeding of 31st SIGCSE, March 2000.

36.Rasala, Richard, Viera K. Proulx, & Harriet Fell, From Animation to Analysis in Introductory Computer Science, SIGCSE Bulletin, March 1994, Vol 26(1), 61-65.

37.Stasko, J. T. (1990). Tango: A Framework and System for Algorithm Animation. IEEE Computer, 23(9): 27-39.

38.[Stasko et al] "Software Visualization" John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price (eds.) MIT Press, 1998

39.Stasko, John T, "Using Student-Built Algorithm Animations as Learning Aids", Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '97), San Jose, CA, February 1997, pp. 25-29.

40.Wolz, U and Koffman, E, "SimpleIO: A Java Package for Novice Interactive and Graphics Programming", Proceeding of ITiCSE 99, 139-142.