Pedagogically Effective Effortless Algorithm Visualization with a PCIL

Brandon Malone, Travis Atkison, Martha Kosa, and Frank Hadlock
bm542@msstate.edu, tla96@msstate.edu, mjkosa@tntech.edu, fhadlock@tntech.edu

Abstract - **Visualization is a promising approach in improving the teaching of algorithms because it can give a pictorial representation of the effect of every step of an algorithm. However, traditional implementations of visualizations require much additional coding to support the infrastructure necessary to step through an algorithm. In this work, we embark on a different path for implementing visualizations, PCIL (PseudoCode Interpreted Language). We believe that PCIL distinguishes itself from other approaches to algorithm visualization by incorporating visualization into its specification. Each language primitive, such as a variable, natively supports a graphical representation. The PCIL interpreter automatically derives visualizations from algorithm implementations. In addition, PCIL includes constructs to facilitate pedagogically effective visualizations, such as the ability to specify custom inputs to algorithms and the ability to ask the student to predict algorithmic behavior. Experimental results indicate that not only do students enjoy using PCIL, they also perform much better on tests after using it compared to students who simply use traditional study aides. Furthermore, the students who use the application for longer amounts of time derive more benefit from the tool than those who only use it for a short time.**

Index Terms – Algorithm visualization, Computer science education, Pedagogical effectiveness, Programming languages.

INTRODUCTION

In today's learning environments, instructors are presented with a classroom of students that are visual learners. The Institute for the Advancement of Research in Education composed a literature review of 29 visual learning studies [1]. The authors summarize that student learning improves with visual learning in areas of retention, comprehension and achievement. Therefore, developing visual learning methodologies and techniques are crucial to instructing current and future students. For this research effort, we restrict our learning domain to teaching the understanding of algorithm behavior to college students. These students are at the forefront of the visual learning generation. Current techniques used in the classroom provide students with static drawings of algorithm behavior; however, this is not an effective way to describe these concepts. To understand algorithm behavior, the student must understand how algorithms handle memory, what their flow is, and how they handle variables which are all dynamic in nature.

We need mechanisms that can inform and teach today's students in a manner in which they can learn productively and with a minimal amount of construction before the results are garnered. Ihantola et al. [2] in their research describe three categories (scope, integrability, and interaction techniques) that "characterize effortlessness in algorithm visualization." Our work attempts to follow these suggested categories by graphically representing algorithms in a way that a student can interact with the algorithm, have the system ask the student questions about the algorithm's behavior, and show the student visually how dynamically the algorithms interactions take place. Though this is not new [3 - 7], we feel that our approach is novel and have shown through actual student-involved experimentation that our system has merit in providing an algorithmic visual learning experience for today's and future visual learners.

The next section provides a short background description of methodologies and techniques of algorithm visualizations. Section 3 discusses PCIL, a new visualization framework. In Section 4, an experimental design is described. Section 5 presents the results of the experiment. Finally, Section 6 concludes with future directions and final thoughts.

BACKGROUND

Algorithm visualization has a long history in computing. The first known visualization, for linked lists, dates back to the 1960's [8]. In the 1980's, the first concentrated efforts to create visualizations with pedagogical value began, with the groundbreaking Brown ALgorithm Simulation and Animator project (BALSA) [9]. The effort required to produce visualization was enormous; Sorting Out Sorting [8], a 30-minute film, took 3 years to produce. In the 1990's, research began to focus on identifying educationally effective aspects of visualizations. Current research continues to examine the educational utility of the visualizations and contexts in which the visualizations offer the most benefit, along with investigating frameworks and approaches to simplify the creation of visualizations.

BALSA led to TANGO (Transition-based ANimation GeneratOr), which allowed gradual changes in the visualization [10]; many descendants followed TANGO. Stasko et al. [11] observed that visualizations augmented with textual descriptions were more beneficial than only the

visualizations. Lawrence [12] demonstrated that students limited to using predefined data to control an animation benefited less than students who could provide their own data. Kehoe et al. [13] showed that visualizations achieved more pedagogical utility when used together with other techniques.

JHAVÉ (Java-Hosted Algorithm Visualization Environment) [5, 6] was designed specifically to incorporate pedagogically effective elements. Its key design goals included presenting both smooth animations and discrete snapshots with rewind capability, accompanying animations with textual descriptions, accepting input data sets to allow unique traces through algorithms, and forcing prediction of future algorithm behavior [5]. JHAVÉ does not specifically generate visualizations, but serves as a framework to view visualizations produced by other systems. A script file supplied to JHAVÉ dictates the visualization behavior; thus, implementations in any language can take advantage of JHAVÉ. In particular, JHAVÉ interoperates with Animal, Samba, and GAIGS, a visualization language for data structures [5].

Naps et al. [7] defined four key aspects of educationally effective visualizations: responding, changing, constructing, and presenting. In responding, a student answers algorithmic behavior questions. In changing, a student changes the behavior by supplying unique inputs. In constructing, a student builds a visualization. In presenting, a student uses a visualization to explain an algorithm. These can serve as a guide to system developers. However, all known visualization systems at that time required that the developer implement an algorithm and then annotate the code with additional calls to generate script files, or otherwise include infrastructure code to animate the algorithm. Unfortunately, especially when trying to allow students to develop visualizations, added annotations and infrastructure obscure the actual algorithm.

Karavirta et al. [3] refined the phrase "effortless creation of algorithm visualization" to describe an ideal system in which visualizations are created automatically from source code. They also classified visualization systems with respect to generality of visualizations that can be produced, and the effort required to create them. WinHIPE, developed b y Pareja-Flores et al. [14], is one "effortless system"; it follows the functional model and "provides an interactive and flexible tracer, as well as a powerful visualization and animation system." [14]. Kerren et al. [15] presented work on a "system for explaining algorithms, which is based on structured hypermedia approach." Diehl et al. [16] presents a nice interface in the GANIMAL project but is restricted to "visual execution of JAVA programs, most notably the mixing of online and post-mortem visualization and the controlled visualization of loops and recursion."

Jeliot [4] has made significant progress toward this effortless goal for visualizing Java programs. We believe that our system PCIL also makes significant progress toward this effortless goal. In PCIL, we define our own grammar for a pseudocode language which has less syntactic overhead than Java. Jeliot appears only to visualize primitives and objects as collections of primitives. While PCIL typically adopts a similar approach, such as with lists, PCIL also provides more abstract data types as primitives. For example, a graph is actually rendered as nodes connected by edges, rather than just a list of node names and edges, or an adjacency matrix. PCIL strives to achieve high generality while still demanding only low effort. The audience for Jeliot is novice students, while the audience for PCIL is students taking an algorithms course, with some previous programming experience in a high-level language.

PCIL SOFTWARE

PCIL approaches the problem of pedagogically effective algorithm visualization by proposing a high-level, interpreted language which incorporates visualization into the primitive constructs and execution of the language. For example, all variables can render themselves to the screen, based on their data types. Additionally, rather than executing an entire program at once, a student controls when to advance from one line of pseudocode to the next. Other educational aspects include allowing the student to supply their own inputs to algorithms and questioning the student about future algorithm behavior.

Due to space limitations, we cannot fully describe PCIL here; Malone [17] provides a full description. We provide some highlights. The grammar for PCIL is an LL(1) grammar with 64 production rules and 24 reserved words. PCIL supports standard blocks (for, decision, and repetition), as well as functions. In addition to the usual integer and string primitives, PCIL provides primitives for structures, graphs (edges and vertices, too), lists, priority queues, and dictionaries. It has an *input* keyword to allow a student to specify the initial values for algorithm variables. For example, for graph variables, the student can use a mouse to create the nodes and edges. Currently either Euclidean distance or user-supplied values can be used for edge costs. It also provides an *ask* keyword to support questioning a student about future behavior. The *ask* keyword presents the student with an interface in which he or she specifies the changes to variables as a result of the next line of pseudocode to be executed. It then provides feedback on the correctness of the student's response. The intent of *ask* is to call attention to the most important steps in the algorithm.

The PCIL parser converts valid PCIL source to a function list and an array of postfix tokens to be processed by the interpreter. Because the conversion takes place before execution, the interpreter never attempts to interpret invalid source code. Instead, an error message indicating the vicinity of the syntax error is given. The interpreter pauses after executing each line of pseudocode to allow the student to consider the new execution state.

The Model-View-Controller (MVC) design pattern [18] supplies the general structure of the visual interpreter, which

is implemented in Java. The model maintains information about the variables and the state of algorithm execution. The view renders the model to the screen for inspection and manipulation. It also alerts the controller when the user wishes to advance to the next line of pseudocode in the algorithm, or when the user has submitted a prediction of upcoming algorithm behavior. The controller interprets the pseudocode and makes necessary changes to the model; it also alerts the view to refresh the model either after stepping or encountering an *ask* statement. Additionally, it compares student predictions to actual behavior in response to *ask* statements.

The primitive variables in the application provide much of their own visualization ability. In addition to maintaining data, each class defining a primitive data type contains methods which define how primitives of that type should render themselves to the screen, in both a read-only and an edit mode, and also how to read a variable back into memory from edit mode rendering. These methods allow the view to render the model easily by simply looping over all variables defined by the model. Similarly, the view can easily acquire inputs of the algorithm by first looping over them and rendering them in edit mode and then reading them back into memory. Likewise, the view can determine the student's response when asked to update the state by looping over all the variables and reading them back into memory to compare with the actual next state found by the algorithm.

```
1.  input = DirectedGraph g, String s
2.  initialize newReachableVertices as PriorityQueue
3.  source = g.getVertex(s)
4.  source.distance = 0
5.  source.predecessor = null
6.  source.status = "touched"
7.  ask: call newReachableVertices.push(source, 0)
8.  while newReachableVertices.isEmpty() = false
9.      initialize i as Vertex
10.     ask: i = newReachableVertices.pop()
11.     if i.status != "visited" then
12.         i.status = "visited“
13.         forEach j in i.neighbors
14.             initialize oldDistance as Integer, newDistance
                as Integer
15.             oldDistance = j.distance
16.             newDistance = i.distance + g.d(i, j)
17.             if (oldDistance = null) or (newDistance <
                oldDistance) then
18.                 j.predecessor = i.value
19.                 j.distance = newDistance
20.                 j.status = "touched“
21.                 ask: call newReachableVertices.push(j,
                    newDistance)
22.             endIf
23.         next
24.     endIf
25. endWhile
26. finish
```

FIGURE 1
PSEUDOCODE FOR DIJKSTRA'S ALGORITHM VISUALIZATION

All primitive variables also define a number of properties unique to their variable type. For example, the *Vertex* primitive type includes a *value* property. However, the variables also allow addition of new properties by simply using an undefined property name. For example, adding a *forest* property to a *Vertex* variable named *v*, would simply require an assignment statement, e.g., *v.forest = 1*. The interpreter automatically infers the type of the new property based on the assigned value. When variables render themselves, they also render their defined properties.

Figure 1 illustrates the simplicity of programming with PCIL by providing the pseudocode for a visualization of Dijkstra's algorithm. Figure 2 shows providing input to Dijkstra's algorithm, while Figure 3 demonstrates stepping through it. Responding to an *ask* question is in Figure 4.

EXPERIMENTAL DESIGN

Our experimental setup follows a pattern similar to that of a basic pretest-posttest randomized experimental design, in which a population is randomly divided into two groups. Because of the random assignment, the groups are assumed

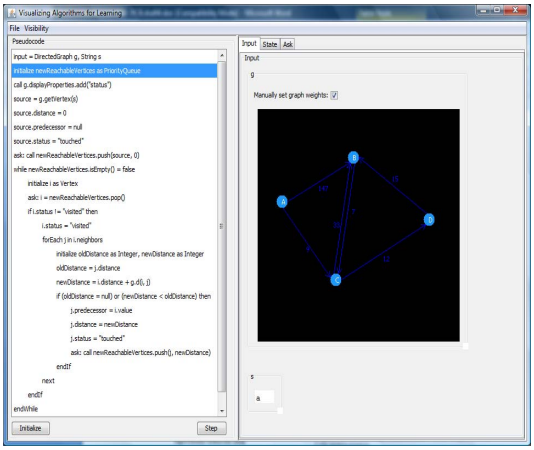

FIGURE 2
INPUT TO DIJKSTRA'S ALGORITHM

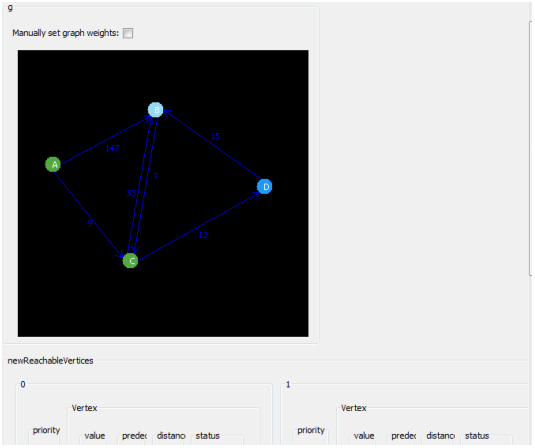

FIGURE 3
STEPPING THROUGH DIJKSTRA'S ALGORITHM

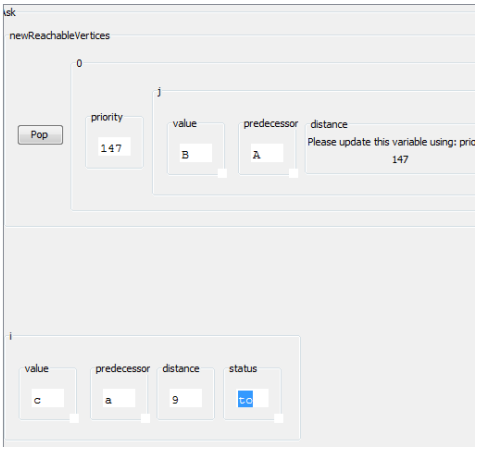

FIGURE 4
RESPONDING TO AN ASK STATEMENT

to be probabilistically equivalent. Such an assumption allays the need to pretest the groups; however, including a pretest can help reduce noise in the final analysis. However, due to the constraints of the educational setting, truly randomized assignment of groups is not possible. Rather, the existing division of students, based on which section of the lecture for which they are enrolled, determines into which group they fall. Thus, a quasi-experimental design was used. This design mimics the pretest-posttest randomized experimental design except that assignment to groups is nonrandom, or nonequivalent.

As mentioned previously, the nonequivalence in the experiment is caused because the students are assigned to groups based on the lecture section for which they are enrolled instead of at random. The pretest consisted of a graded quiz. In this experiment, the treatment was an instructional tutorial on using PCIL and access to the PCIL tool during homework. The posttest consisted of only a graded quiz. The experimental group also evaluated PCIL using a survey. Later, during analysis, the pretest was used to correct the nonequivalence between the groups.

The experimental group consisted of 26 students. The students first took a pretest evaluating their knowledge of Dijsktra's algorithm. Then, they were shown a tutorial about using PCIL. Next, the class received a homework assignment and instructions on how to download and install PCIL on their own computers. No class time was given to use the system. The homework was assigned on a Thursday and collected the following Tuesday. After collecting the homework, the students then took a posttest and a survey evaluating their experience using the software. Two of the students were absent for the posttest. Thus, the withdrawal rate, that is, the percentage of individuals beginning the experiment but not completing it, of the experimental section was 7.7%.

The procedure for the control group mirrors that of the experimental group closely. 25 students took the pretest and the same survey as the experimental section. This section did not receive the PCIL tutorial. However, they did receive the same homework assignment given to the experimental section, but were not given access to download PCIL. Again, the homework was assigned on Thursday and collected on Tuesday. Following collection of the homework, the students took a posttest. Two students were absent for the posttest, yielding a withdrawal rate of 8%.

EXPERIMENTAL RESULTS

The results are broken down in two ways. First, the results of the experimental group are compared to those of the control group. Elementary analysis of these results was used to assess the presence of threats to the internal validity of the experiments. Because the goal of the experiment is to show that using PCIL during homework results in higher test grades, these threats could lead to higher test grades, but not as a result of PCIL. Then, the experimental group was further divided based on the self-reported length of time the students spent using the tool.

Tables III and IV display the results of the experiment. Table III compares some basic statistics of the experimental section results to the control section results. Table IV delves into the results of the experimental section more deeply. That section is further divided into students who used the tool for less than half an hour and those who used it for a half hour or more. Figure 5 graphically illustrates the change in means from the pretest to the posttest for the different groups.

As Figure 5 illustrates, although the mean of the experimental section was lower on the pretest (61.9 vs. 72.2), it was higher on the posttest (81.5 vs. 78.5). According to Trochim [19], this "crossover pattern" strongly suggests the treatment had a positive effect on the students' grades. Typical threats to the reliability of the data, such as selection maturation, do not likely explain the results. Selection maturation results when individuals in one group naturally develop later than the other group. For example, if the students in the experimental group tended to develop a better understanding of algorithms a week after hearing a lecture over the algorithm, while the control groups developed understanding immediately after the lecture, then a selection maturation effect could explain an increase in the experimental group compared to the control group. Selection regression is another potential threat. It affects extreme values in the data set; they tend to move closer to the mean from the pretest to the posttest. Thus, if the experimental group contained many very low values on the pretest, selection regression would suggest that the low grades would regress closer to the mean, masquerading as improvement due to PCIL. Manifestation of these phenomena could result if the experimental grades asymptotically approached the grades of the control group, but the crossover exorcises these demons. Thus, the effectiveness of PCIL likely explains the improvements.

Speaking strictly in terms of improvement, the experimental class improved, on average, nearly two letter grades (19.6 points) from the prequiz to the postquiz. The control class, however, improved only about a half a letter

grade (6.3 points). Thus, access to PCIL, compared to just using normal homework methods, resulted in nearly a letter grade and a half more improvement from the pretest to the posttest. Using a simple gain scores ANOVA, in which the difference between pretest and posttest grades for each group define two new datasets (gainScore = posttest – pretest), this change in grade is significant (p-value = .0269). A relative difference gain scores ANOVA, which is similar to the simple gain score except that the pretest-posttest difference is divided by the pretest score (relativeGainScore = [posttest – pretest] / pretest), also suggests a significant change (p-value = .0168). As mentioned earlier, the trends in the grades themselves dispel concerns about internal validity. Hence, the results suggest that PCIL does have a significant impact on the gain scores.

Dividing the results of the experimental section further highlights the pedagogical effectiveness of PCIL. The outcomes of those students who reported using PCIL for fewer than 30 minutes were compared to those reporting using the tool for 30 minutes or more. This breakdown reveals that, although the students reporting using the tool for less than a half hour scored slightly higher on the pretest (62.7 compared to 60.9), the difference is not significant (p-value = .8747). Thus, we can reasonably assume the groups are statistically similar before using PCIL. On the posttest, however, the students using the tool for a half hour or more improved by nearly a letter grade more than those using the tool for less than 30 minutes (24.1 points compared to 15.8, respectively). While this relative difference gain score is not statistically significant (p-value = .2737), the small sample sizes (13 using the tool for less than a half hour, 11 using the tool for a half hour or more) impedes rigorous statistical analysis. The group using the tool for a half hour or more also demonstrates more consistency than the other groups, as well. Their standard deviation on the posttest was a meager 5 points, but the group using the tool for less than 30 minutes had a standard deviation of 24.6. This could imply that students using PCIL tend to concentrate on the same parts of the algorithm. Likely, the *ask* feature focuses their attention to the algorithm's key steps.

Table V indicates the results of the survey taken by the experimental group after using PCIL on a 10 point Likert scale (1 = disagree strongly, 10 = agree strongly). The questions were as follows: 1) PCIL is **easy to use**. 2) **PCIL helped** me understand Dijkstra's algorithm. 3) I would use PCIL to **study for a test**. 4) I would **use** PCIL to help **while programming**. 5) I would **recommend** PCIL to a friend.

As the table suggests, on average, the class felt slightly positively about PCIL since most responses were between 5 and 6. However, the differences between the results for those reporting using PCIL for more or less than a half hour are striking. Those reporting usage for less than a half hour did not appear to feel that PCIL helped them understand the algorithm and were not likely to recommend PCIL to a friend. Thus, they probably did not enjoy using the tool. Their responses on the questions mostly fell below 5. In contrast, students using PCIL for more than a half hour felt that PCIL helped their understanding of the algorithm, and were likely to use PCIL to help with studying for tests and while programming. They were also apt to recommend PCIL to friends.

The survey results could be interpreted in at least two ways. Students who used PCIL for less than a half hour may have disliked it, and thus not used it for very long; on the flipside, those who enjoyed using PCIL and felt it helped them understand the algorithm had no difficulty using PCIL longer. A learning curve for PCIL could also account for the differences. So, the results could suggest that, in addition to an in-class tutorial, using some class time to allow students to use PCIL could increase its pedagogical effectiveness.

TABLE III
RESULTS BROKEN DOWN BY TREATMENT GROUP.

Experimental Group		Control Group	
Initial Sample Size	26	Initial Sample Size	25
Final Sample Size	24	Final Sample Size	23
Withdrawal %	7.7%	Withdrawal %	8.0%
Pretest Mean	61.9	Pretest Mean	72.2
Posttest Mean	81.5	Posttest Mean	78.5
Posttest Std. Dev.	18.4	Posttest Std. Dev.	10.0
Posttest– Pretest	19.6	Posttest– Pretest	6.3

TABLE IV
EXPERIMENTAL RESULTS BROKEN DOWN ACCORDING TO SELF-REPORTED TIME USING PCIL.

< 30 minutes		≥ 30 minutes	
Sample Size	13	Sample Size	11
Pretest Mean	62.7	Pretest Mean	60.9
Posttest Mean	78.5	Posttest Mean	85
Posttest Std. Dev.	24.6	Posttest Std. Dev.	5
Posttest– Pretest	15.8	Posttest– Pretest	24.1

TABLE V
SURVEY RESULTS FROM EXPERIMENTAL GROUP ON 10 POINT LIKERT SCALE.

	Total	< 30 min.	>= 30 min.
Sample Size	24	13	11
Easy to Use	5.2	4.6	5.8
PCIL Helped	5.6	4.4	6.7
Study for Test	5.7	4.5	7.0
Use while Programming	6.1	5.2	7.1
Recommend	5.9	5.1	6.7

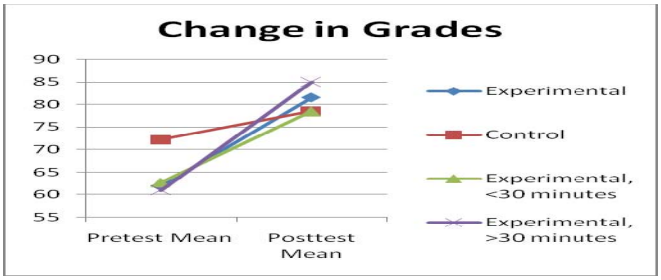

FIGURE 5
CHANGE IN MEANS FOR GROUPS.

CONCLUSIONS

Initial classroom tests for PCIL yielded positive results. The students improved their ability to answer questions about the behavior of Dijkstra's algorithm. To gain more confidence in the effectiveness of PCIL, we need to test with more questions, more students, and more instructors. The effectiveness of PCIL for lectures over a single algorithm and for the entire algorithms course can be measured; the performance of students in the course without PCIL can be compared on questions about algorithmic behavior. Another possible measurement would be to compare the performance of students using PCIL as an in-lab study tool to the performance of students in the same course not using PCIL, those using PCIL as an out-of-class aid, or those using other visualizations. We could also study the effect of PCIL on student performance at various levels of Bloom's taxonomy, as advocated in Naps *et al.* [7].

The final future focus concerns PCIL's educational features. PCIL could be improved by featuring a history stack, allowing a user to step backward and view previous execution states; many visualization systems provide this. Another improvement could allow an instructor to specify a desired algorithm behavior or state, and then check after each step to see if the input supplied by the student results in the desired state. When analyzing responses to questions, PCIL could also employ intelligent tutoring techniques to identify the cause of errors.

Due to their explanatory power, algorithm visualizations can potentially serve an integral role in education. Research in algorithm visualization has led to improvements in pedagogical effectiveness. PCIL incorporates visualization into the fabric of the language. Thus, creating a visualization only requires implementing the algorithm; painting and button handling are supported transparently. Initial results indicate that students using PCIL for homework outperform students with only conventional means of study.

REFERENCES

[1] "Graphic Organizers: A Review of Scientifically Based Research", The Institute for the Advancement of Research in Education, July 2003.
[2] Ihantola, P., *et al.*, "Taxonomy of Effortless Creation of Algorithm Visualizations," in Proceedings of the First International Workshop on Computing Education Research, Seattle, WA, 2005, pages 123 – 133.
[3] Karavirta, V., *et al.* 2002. Effortless Creation of Algorithm Visualization. In Proceedings of the Second Annual Finnish/ Baltic Sea Conference on Computer Science Education (Koli, Finland, October 18–20, 2002), 52-56.
[4] Moreno, A., *et al.* Program Animation in Jeliot 3. In Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (Leeds, UK, June 28-30, 2004). ITiCSE '04. ACM Press, New York, NY, 265.
[5] Naps, T.L. 2005. JHAVÉ – Addressing the Need to Support Algorithm Visualization with Tools for Active Engagement. 2005. IEEE Computer Graphics and Applications 25, 5 (Sept./Oct. 2005), 49-55.
[6] Naps, T.L., Eagan, J.R., and Norton, L.L. 2000. JHAVÉ – An Environment to Actively Engage Students in Web-based Algorithm Visualization. In Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education (Austin, Texas, March 2000), SIGCSE 2000, ACM Press, New York, NY, 109-113.
[7] Naps, T.L, *et al.* 2003. Exploring the Role of Visualization and Engagement in Computer Science Education. SIGCSE Bulletin 35, 2 (June 2003), 131-152.
[8] Baecker, R. 1998. Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. In Software Visualization: Programming as a Multimedia Experience. MIT Press, Cambridge, Massachusetts, 369-381.
[9] Brown, M.H. 1998. Algorithm Animation. MIT Press, Cambridge, Massachusetts.
[10] Stasko, J. 1990. TANGO: A framework and system for algorithm animation. IEEE Computer 23, 9 (September 1990), 27-39.
[11] Stasko, J., Badre, A., and Clayton, L. 1993. Do Algorithm Animations Assist Learning? An Empirical Study and Analysis. In Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems (Amsterdam, The Netherlands, April 24-29, 1993). CHI '93. ACM Press, New York, NY, 61-66.
[12] Lawrence, A. 1993. Empirical Studies of the Value of Algorithm Animation in Algorithm Understanding. Doctoral Thesis. Georgia Institute of Technology.
[13] Kehoe, C., Stasko, J., and Taylor, A. 1999. Rethinking the Evaluation of Algorithm Animations as Learning Aids: an Observational Study. Technical Report. Georgia Institute of Technology.
[14] Pareja-Flores, C., Urquiza-Fuentes, J. and Velazques-Iturbide, J., "WinHIPE: An IDE for Functional Programming Based on Rewriting and Visualization," ACM SIGPLAN Notices, vol 42, issue 3, March 2007, pages 14 – 23.
[15] Kerren, A., Muldner, T. and Shakshuki, E., "Novel Algorithm Explanation Techniques for Improving Algorithm Teaching," in Proceedings of the 2006 ACM Symposium on Software Visualization, Brighton, UK, 2006, 175 – 176.
[16] Diehl, S. and Kerren, A., "Reification of Program Points for Visual Execution," in Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis, June 2002, pages 100 – 109.
[17] Malone, B.M. 2008. Incorporating Visualization in an Interpreted Language for Educational Benefit. Master's Thesis. Tennessee Technological University.
[18] Model-View-Controller. http://ootips.org/mvc-pattern.html. Accessed 14 August 2008.
[19] Trochim, W. M. K. "Nonequivalent Group Analysis," Research Methods Knowledge Base, http://www.social researchmethods.net/kb/statnegd.php. Accessed 1/22/2009.

ACKNOWLEDGMENT

Thanks to Joey Haas for help with the statistical analysis.

AUTHOR INFORMATION

Brandon Malone, Mississippi State University (MSU), bm542@msstate.edu
Travis Atkison, MSU, tla96@msstate.edu
Martha Kosa, Tennessee Technological University (TTU), mjkosa@tntech.edu
Frank Hadlock, TTU, fhadlock@tntech.edu