Editors: Beatriz Sousa Santos and Ginger Alford

Algorithm Animation with Galant

Matthias F. Stallmann
North Carolina State University

Algorithm animations abound: students love to interact with them, instructors use them in classroom demonstrations, and they are useful shortcuts to detailed tracing of algorithm behavior. Although surveys suggest positive student attitudes toward the use of animations, it is unclear that they improve learning outcomes. The Graph Algorithm Animation Tool, or Galant, challenges and motivates students to engage more deeply with algorithm concepts, without distracting them with programming language details or GUIs.

Galant is also a term for a musical style that featured a return to classical simplicity after the complexity of the late Baroque era. The goal is to achieve the same in the Galant approach to algorithm animation, which offers three key advantages:

- Students can write animations in a simple procedural language. The result resembles pseudocode with animation directives inserted.
- They can create problem instances (graphs) by drawing them, issuing keyboard commands, or importing them from external sources (such as files in various formats or random generators).
- With intermediate Java expertise, they can enhance Galant's algorithmic capabilities by importing their own Java classes.

A side benefit of these features is that Galant has been used extensively in graph algorithm research. Even though Galant is specifically designed for graph algorithms, it has also been used to animate other algorithms, most notably sorting algorithms.

Algorithm Animation

Algorithm animation has a long history, dating back at least as far as the work of Marc Brown and Robert Sedgewick[1,2] and that of Jon Bentley and Brian Kernighan[3] in the 1980s. The Balsa software, developed by Brown and Sedgewick, is a sophisticated system that provides several elaborate examples of animations, including various balanced search trees, Huffman trees, depth-first search, Dijkstra's algorithm, and transitive closure. The Bentley-Kernighan approach is simpler: an algorithm implementation is annotated with output directives that trace its execution. These directives are later processed by an interpreter that converts each directive into a still picture (or modification of a previous picture), and the pictures are composed into a sequence that the user can navigate as if it were a video.

In discussing algorithm animation software, we distinguish between three primary roles: the *observer* simply watches an animation; the *explorer* interacts with an animation by, for example, changing the problem instance; and the *animator* designs an animation. The animator may also be referred to as a developer if the process of creating animations is integrated with that of implementing the animation system. Guido Rössling and Bernd Freisleben articulated a similar classification of roles.[4]

A hypothesis, at least partially validated (using student attitude surveys[5]) posits that, although students acting as observers or explorers of animations may gain some benefit, a student who takes on the role of animator has a much richer learning experience. This hypothesis is Galant's major motivating factor: it should simplify the animator's task as much as possible while enhancing the ability to produce compelling animations. A first step in that direction was GDR (Graph Drawing),[6] which is still used in algorithm and automata theory classes. GDR's user interface is primitive, however, and it requires writing C programs (with powerful macros) to implement animations.

Previous Algorithm Animation Software

The variety of algorithm animation software is overwhelming. Most of it either treats the user as an observer or, at best, an explorer. In the latter

 Published by the IEEE Computer Society

category, two stand out because of their sophistication and the variety of (graph) algorithms offered. Both Javenga[7] (an online applet) and j-Alg (j-algo.binaervarianz.de) have excellent user interfaces and a large collection of graph algorithm animations: breadth-first and depth-first search, topological sort, strongly connected components, shortest paths algorithms, minimum spanning tree algorithms, a network flow algorithm, and others. Animations in each of these are created by developers.

Although many animation tools facilitate the creation of observer-based animations, only a few were designed to create animations for exploration. Edgy is an applet that combines exploration, simple programming techniques (using Snap), and a simple graph interface (see http://snapapps.github.io). Because the programming capabilities are primitive, it would be difficult, for example, to add sophisticated data structures such as priority queues or external Java classes. Galles is more sophisticated (see www.cs.usfca.edu/~galles/visualization/Algorithms.html); animators can use Java or Flash to create compelling animations. However, the graph algorithm examples provided have only limited exploration capability, and there is no mechanism for a user to create or import arbitrary graphs. And, as is the case with most animation software, Galles focuses the animator on the graphics rather than on the algorithm or the graph.

AlgoViz (algoviz.org) is a large catalog of algorithm animations, continually updated by contributors who either submit new animations or comment on existing ones. Like any large repository with many contributors, AlgoViz is difficult to monitor and maintain. The OpenDSA Project aims to create a textbook compilation of a variety of visualizations, mostly designed for observers.[8] Stina Bridgeman has also provided a survey of educational applications of graph drawing software.[9]

Galant Features

What distinguishes Galant from other recent animation software is that it trades off more sophisticated visual effects in favor of animator capabilities that focus on the algorithms, leaving the software to handle graphics. In addition, Galant is a tool rather than a closed system—that is, it can easily interact with other software.

Figure 1 illustrates Galant's design. Unlike most animation software, Galant is specifically designed so that all of its functionality can, if desired, be outsourced to external programs that will work with GraphML representations (for the graphs) or act as program editors (for the animation code). Additional functionality such as graph drawing, random or structured instance generation, other types of graph visualization, and any functionality provided by Java can also be provided by external software. Galant provides the most important facilities directly.

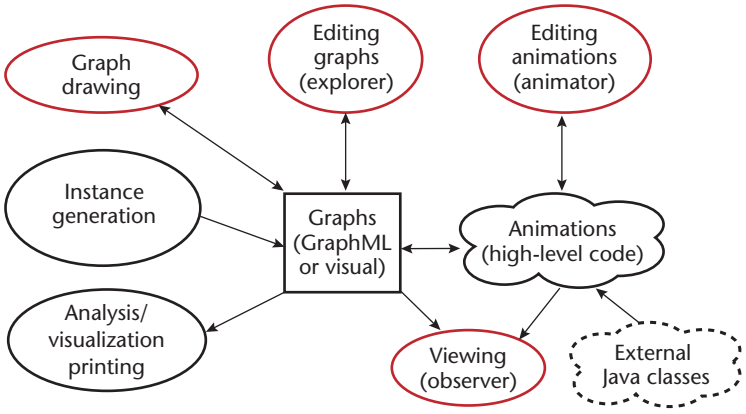

Figure 1. Galant design and functionality. Arrows indicate data flow. All functions, shown as ovals, can be performed easily outside of Galant. The functions with thick red borders are part of current Galant functionality.

Features for the Animator Role

Galant offers the animator distinctive advantages over other animation tools. Unlike j-Alg, which requires nontrivial Java expertise, Galant's interface is accessible to novice programmers familiar with graph algorithms.

Editing and compiling an algorithm animation is similar to performing the same operations on a Java program. In fact, an animation's code is like a Java program with

- predefined types for nodes and edges;
- a programmer interface that interacts with the graph and with intended animation effects;
- functions that manipulate attributes such as node positions and weights, labels, color, highlighting/marking, and node and edge visibility;
- built-in data structures, such as lists, stacks, queues, sets, and priority queues of nodes and edges;
- macros that allow the program to traverse, for example, all incident edges of a node;
- a mechanism for specifying steps in the animation execution (the default is to take a new step for every change in the display); and
- procedural syntax for object-oriented constructs.

Although Galant was designed for animators with only rudimentary Java skills, it uses the full power of Java and therefore has advantages over other simple animation languages such as Samba[5] and Edgy. The animator can develop animations

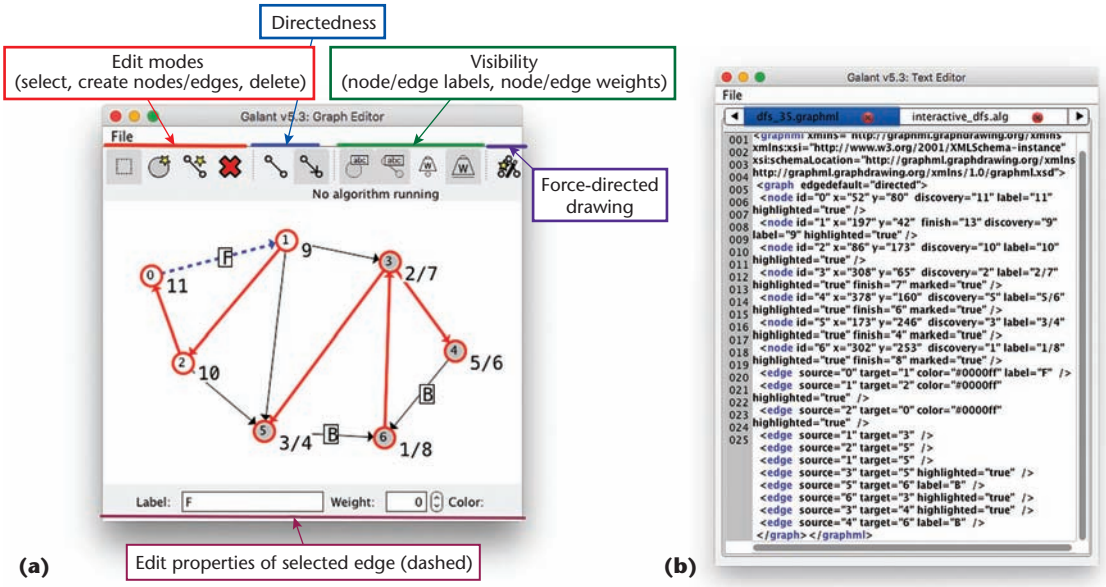

Figure 2. The Galant user interface. (a) Using the graph window, users can add or delete nodes and edges, move nodes, and so forth. (b) The text window shows the GraphML representation.

of arbitrarily complex algorithms using additional Java classes and thereby offer new functionality with only a modicum of Java training. For example, some students in a data structures class have taken advantage of this capability, as have I, to explore algorithms in research applications. In a few cases, the new capabilities have become part of Galant.

Galant User Interface

The Galant GUI lets the user edit both graphs and algorithm animations, either loading existing files or creating new ones. At any point, the user can apply a selected animation to a selected graph.

Two windows appear when Galant is initiated: a graph window that shows the current graph and a text window that shows editable text (see Figure 2). Depending on the currently selected tab in the text window, the text can be either a GraphML representation of the current graph (as in Figure 2) or an algorithm implementation (see Figure 3). The graph in Figure 2a is a midstream export during a depth-first search (via `File` → `Export` in the graph window). The GraphML in Figure 2b shows visible attributes (`label`, `color`, `highlighted`, and `marked`) and invisible attributes controlled by the algorithm (`discovery` and `finish`). The attributes `id`, `x`, and `y` are mandatory for nodes, whereas `source` and `target` are mandatory for edges.

The user interface is designed for all three roles. The observer (or an instructor demonstrating an algorithm) follows the following steps:

1. Load a graph using the file browser.
2. Load an algorithm.
3. Push the Compile and Run button (see Figure 3a).
4. Use buttons or arrow keys to step through the algorithm forward or backward as desired.

When held down, the arrow keys can be used to fast forward/reverse or to create a movie effect.

A typical explorer might edit or create a graph using the graph window and then follow steps 2–4, repeating steps 1 and 4 to try out different graphs. An animator can load and edit an existing algorithm or create one from scratch using a tab in the text window (or with an external program editor). The compiler reports errors with line numbers (the numbers on left of the text window in Figure 3a). Galant catches most runtime errors before they trigger lower-level Java exceptions. Either way, the stack trace, displayed both on the console and, at the user's discretion, in a pop-up window, shows text window line numbers.

Figure 4 illustrates use of the force-directed drawing feature.[10] The graph (25 nodes and 40 edges) was randomly generated with no position information. (It was created for minimum spanning tree experiments and converted by script to GraphML.) After force-directed drawing, the user can move nodes around to easily achieve an even better layout and edit edge weights to explore alternative executions of a minimum spanning tree or shortest path algorithm.

Animation Example

An interactive depth-first search implementation can help illustrate Galant's capabilities. Other animations implemented so far include Dijks-

tra's algorithm, breadth-first search, Kruskal's and Boruvka's minimum spanning tree algorithms, several sorting algorithms, and algorithms used for research in graph drawing, vertex cover, and medians in trees. The software download at github.com/mfms-ncsu/galant includes these algorithms, an updated technical report, and documentation for users, programmers, and developers.

Figure 3 shows both the text window with the algorithm's tab active and snapshots of four key steps in the graph window. The Galant function names are highlighted in red, macros are in green, and Java keywords are in blue. (The user can select different colors using a Preferences panel.) The animator defines a function using the `function` macro, translated to a Java static method; an optional return type can be inserted after `function`. The `algorithm` macro defines the main program. The macro `for_outgoing(v,e,w)` creates a loop whose body is executed once for each edge leading out of *v*. In the body, *e* refers to the current edge and *w* to the other endpoint. In an undirected graph, *outgoing* applies to all incident edges.

The animation selects (highlights with red boundary) a node at the beginning of a visit and marks (gray shading) it at the end. Edges are colored blue as they are explored and uncolored when done. Tree edges are highlighted while others are labeled by category: B for back, C for cross, and F for forward. Node labels are used to indicate discovery and finish times in this format: `d/f`.

The function call

```
getNode("node to visit", unvisited,
                "node already visited")
```

at the beginning of the main loop prompts the user to select the node to visit next, allowing exploration of multiple searches of the same graph. If the selected node is not in the set specified by the second argument, the error message (third argument) appears and the user has another chance. From top to bottom, the snapshots in Figure 3 illustrate near completion of the search that chose nodes 1 and 4, in that order, in the outer loop. Figure 2 shows a midstream export of a search choosing node 6 and then node 1.

The functions `beginStep` and `endStep` define the points at which the exploration of the algorithm stops its forward or backward motion, respectively. In their absence, any state change (such as mark, highlight, or change in weight) constitutes a step. Our approach to pausing execution has an advantage over simply specifying breakpoints. Initially, the animation (with no begin or end steps) will display every state change, which is useful for debugging. The animator can then systematically decide which display effects should occur simultaneously.

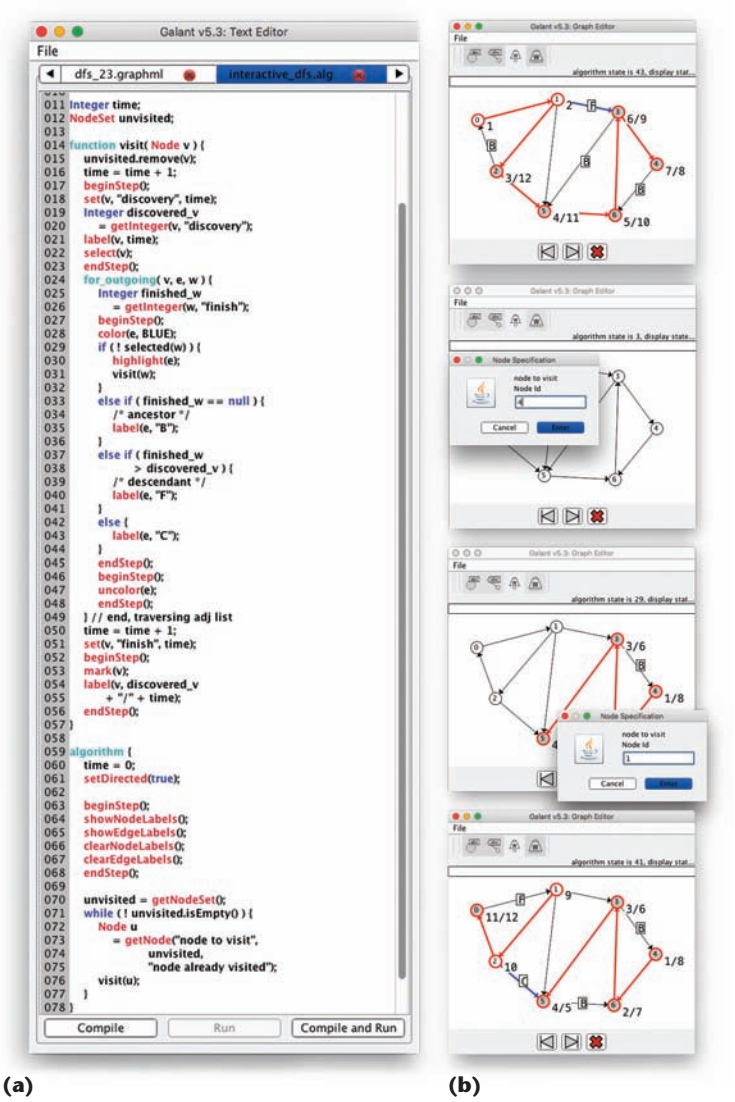

Figure 3. Animation of a depth-first search on a directed graph. (a) In the GraphML, the Galant function names are highlighted in red, macros are in green, and Java keywords are in blue. (b) From top to bottom, the snapshots illustrate near completion of the search that chose nodes 1 and 4, in that order, in the outer loop.

Another feature illustrated in the depth-first search example is the manipulation of the integer attributes `discovery` and `finish`, defined by the animator. The Galant function `set`(*x*, attr, val) sets attribute `attr` of node/edge *x* to val, and `getInteger`(*x*, attr) retrieves the value of integer attribute attr.

Classroom Use

Galant was first deployed publicly at North Carolina State University in a Spring 2015 course (CSC 316, Data Structures). The prerequisites

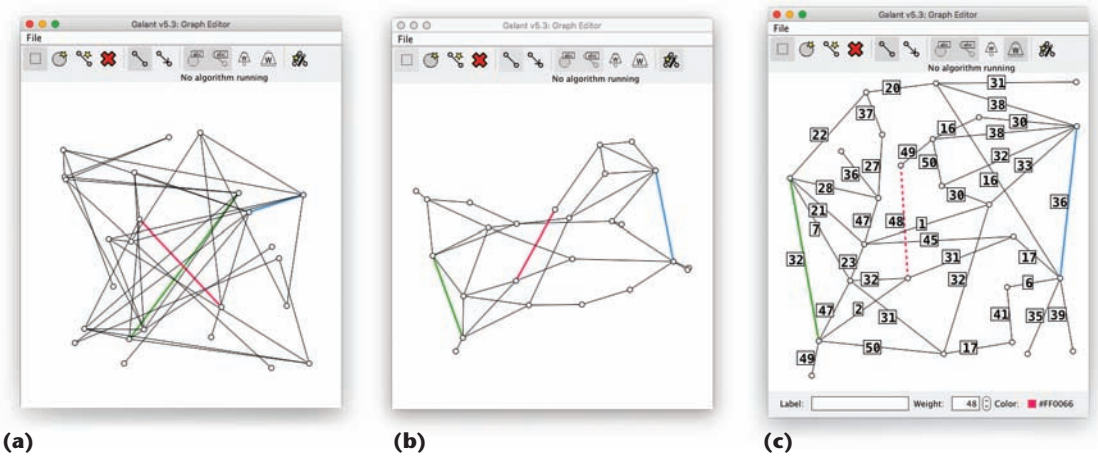

Figure 4. Using force-directed drawing to create usable graphs. (a) The initial random graph has 25 nodes and 40 edges. This illustration shows the results (b) after force-directed layout and (c) after edits.

```
algorithm Boruvka (G = (V, E))
    T    V (a forest with no edges)
    while T has fewer than |V| - 1 edges do
       for each connected component C in T do
          let e be the smallest weight edge
             from C to another component in T
          if e is not already in T then
             add edge e to T
    return T
```

Figure 5. Boruvka's algorithm. Students worked in teams to implement animations. This algorithm was one of the options listed in the instructions.

for that course are two semesters of Java and one semester of discrete math. In one of three programming projects that semester, students worked in teams to implement animations. Team composition was voluntary. There were 121 students and 34 teams, each consisting of three or four students.

The relevant instructions were as follows:

> Each team is expected to produce three algorithm animations. For one of these, choose between animating Boruvka's minimum spanning tree algorithm or the biconnected components algorithm described in a separate document. For the second, choose between merge sort, Quicksort, or Heapsort. (These will require some creativity.) For the third, choose an additional algorithm from among the ones I've listed so far or another algorithm of your choice, with my permission. (There are lots of graph algorithms that are relatively easy to understand and implement.) Also expected is a document describing various aspects of your experience using Galant. Include a paragraph or two answering these questions: What did I learn from working on these animations? Did I gain a better understanding of the algorithms by animating them instead of working through them on examples or watching examples presented in class?

Despite many technical challenges (including serious bugs forcing a new release midstream), most students rose to the occasion and produced surprisingly creative animations. Figure 5 shows Boruvka's algorithm, as presented in class. Most students used marking (shading) to indicate the nodes of the component C from which the lowest cost incident edge is chosen. They also made heavy use of the message banner at the top of the window to give a blow-by-blow description of the algorithm steps—the beginning of each iteration of the outer loop, for example.

The students' struggles motivated the instructor (as developer) to add features, such as selective hiding of edge weights and of the edges themselves to show components more effectively (see Figure 6). Also illustrated in Figure 6 is the `Node Radius` preference; the user can specify this at any time. (Node IDs are not visible when the radius is small enough.) Note the state information at the top of the frame: the "before" snapshot was actually taken later than the "after"; the user had already stepped the algorithm to state 33 before backtracking the display to state 26.

The sorting algorithm animations involved elaborate node movement on the screen: to show the heap for Heapsort and the recursive decomposition of the arrays for merge sort and Quicksort. Figure 7 shows one example. The animation shows each element being compared with the pivot and moved to one side of the array or the other (see

Figures 7a and 7b, respectively). Color and vertical position help illustrate the recursive decomposition of sublists (see Figure 7c). Some student sorting animations added edges to illustrate links in a linked-list based merge sort or edges in a heap-ordered tree.

Although there was no formal attitude survey, student feedback was gathered on using Galant for this project. The positive responses referenced specific algorithms where animation development enhanced learning. Boruvka's was mentioned most frequently, followed by merge sort. There were also statements about the importance of the "interplay between algorithm implementation and visualization" and that "in order to test if you really know the material, you try to explain it to someone that doesn't know it."

There were also some negative responses. In those cases, the students did not feel the exercise provided a useful learning experience. Most of these teams also had serious technical difficulties.

Of the 30 teams that responded to the questions, 16 had positive responses, 11 neutral (which were either nonspecifically positive or a mix of positive and negative statements), and three were negative.

Galant is a simple yet powerful algorithm animation tool for students that allows them to engage more deeply in algorithm details. It also lets instructors easily create compelling animations for the classroom and for student exploration and helps researchers explore novel graph algorithms and heuristics.

As I mentioned, an early version, more primitive than the one currently available, was used successfully in a class project. Students created animations of a large variety of algorithms: breadth-first and depth-first search, shortest paths, minimum spanning trees (Boruvka and Kruskal), biconnected components, finding the center of a tree, sorting (bubble, insertion, merge, heap, Quicksort), and even fractal generation based on a graph. The instructor enhanced some of these and, as a researcher, created algorithms for finding *p*-medians of trees and algorithms/heuristics for vertex cover and minimizing edge crossings in graph drawings. All these activities are evidence of Galant's success: the animations took relatively minimal effort to create (just a few hours for those created by the author, and a few weeks for students who were busy with other classes and plagued by major bugs). These resulting animations ranged from about 15 (breadth-first search) to 150 (the elaborate Quicksort) lines of code.

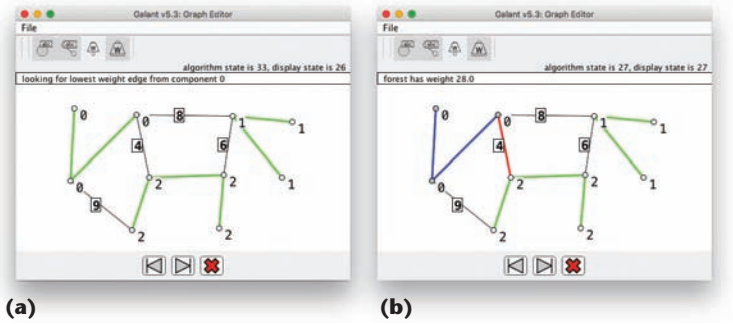

Figure 6. Two consecutive steps of Boruvka's algorithm: (a) before the top-left component chooses the min-cost edge and (b) after choosing the min-cost edge (incident on node 6). The graph has edges 0–6, 2–7, and 4–9 hidden because they connect nodes in the same component.

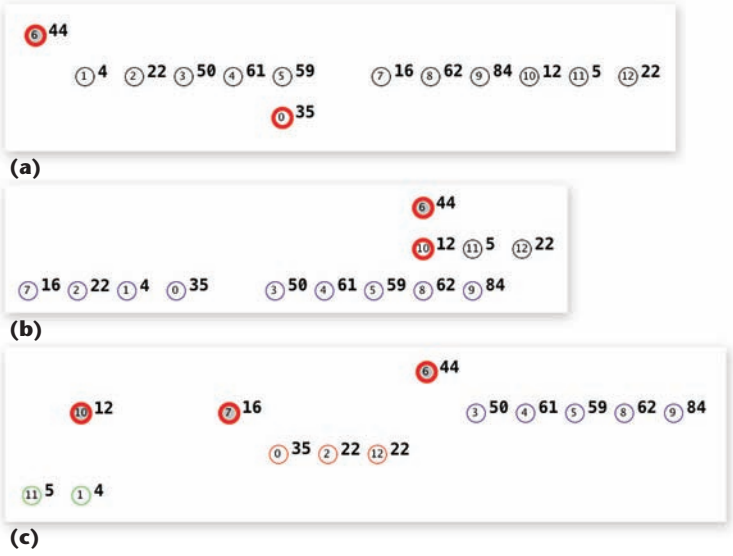

Figure 7. A student animation of Quicksort: (a) first comparison with the pivot (44), (b) original list partially decomposed into two sublists, and (c) after two recursive calls on the left side.

As the repertoire of Galant animations expands, so will its functionality. The Galant animation language has and will continue to be repeatedly enhanced to accommodate desired animation effects. Finally, the code base of Galant undergoes frequent refactoring, making it more accessible to external developers.

Acknowledgments

Special thanks are due to the North Carolina State University undergraduate students who helped develop the initial Galant prototype and/or added significant features: Jason Cockrell, Tynan Devries, Weijia Li, Alexander McCabe, Yuang Ni, and Kai Presler-Marshall. I also thank Robert Fornaro and Margaret Heil, instructors of the senior design course where the initial prototype of Galant was implemented, for coordinating the project(s) and providing instruction in com-

munication skills, respectively. Ignacio Dominguez provided valuable technical support. Thanks also to the students of CSC 316, Spring 2015, for field testing an early version of Galant and putting up with its many shortcomings. The Quicksort example is from Hayden Fuss, Solomon Yeh, and Jordan Connor. Finally, thanks to Ben Watson for suggesting CG&A *as a venue for this work and to the anonymous reviewers for their helpful comments.*

References

1. M.H. Brown, "Exploring Algorithms Using Balsa II," *Computer*, vol. 21, no. 5, 1988, pp. 14–36.
2. M.H. Brown and R. Sedgewick, "Techniques for Algorithm Animation," *IEEE Software*, vol. 2, no. 1., 1985, pp. 28–39.
3. J.L. Bentley and B.W. Kernighan, "A System for Algorithm Animation: Tutorial and User Manual," Computing Science Tech. Report 132, AT&T Bell Laboratories, Jan. 1987.
4. G. Rössling and B. Freisleben, "ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation," *J. Visual Languages and Computing*, vol. 13, no. 3, 2002, pp. 341–354.
5. J.T. Stasko, "Using Student-Built Algorithm Animations as Learning Aids," *Proc. 28th ACM Tech. Symp. Computer Science Education* (SIGCSE), 1997, pp. 25–29.
6. M. Stallmann, R. Cleaveland, and P. Hebbar, "GDR: A Visualization Tool for Graph Algorithms," *Computational Support for Discrete Mathematics*, vol. 15, N. Dean and G. Shannon, eds., Am. Mathematical Soc., 1992, pp. 17–28.
7. T. Baloukas, "JAVENGA: Java-Based Visualization Environment for Network and Graph Algorithms," *Computer Applications in Eng. Education*, vol. 20, no. 2, 2012, pp. 255–268.
8. E. Fouh, M. Sun, and C. Shaffer, "OpenDSA: A Creative Commons Active-eBook," poster presentation., *Proc. 43rd ACM Tech. Symp. Computer Science Education* (SIGCSE), 2012, p. 721.
9. S. Bridgeman, "Graph Drawing in Education," *Handbook of Graph Drawing and Visualization*, R. Tamassia, ed., CRC Press, 2013, chap. 24.
10. Y. Hu, "Efficient, High-Quality Force-Directed Graph Drawing," *Mathematica J.*, vol. 10, no. 1, 2006, pp. 37–71

Matthias F. Stallmann *is a professor of computer science and the assistant director of graduate programs at North Carolina State University. Contact him at mfms@ncsu.edu.*

Contact department editor Beatriz Sousa Santos at bss@ua.pt and department editor Ginger Alford at ginger.siggraph@gmail.com.