Algorithm Animation: Using algorithm code to drive an animation

John Morris

School of Electrical and Electronics Engineering,
Chung-Ang University, Seoul 156-756, Korea
and
Department of Computer Science,
The University of Auckland
Email: j.morrris@auckland.ac.nz

Abstract

Computer algorithms commonly involve creation, reorganization or destruction of relations between objects. This means that they are generally excellent candidates for visualization for teaching purposes. For a student to comprehend all the ramifications of certain operations, several different concurrent displays are often required: as well as a visualization of the objects being manipulated by the algorithm, commentary, highlighted source code and various statistics are commonly required.
This paper describes a strategy for rapidly building animations of algorithms: animations are driven from the source code of the algorithm itself by the addition of animation directives. These directives invoke routines from a toolkit which provides operations commonly needed to display objects and structures. By providing a standard approach to the creation of an animation, creating a new one becomes a straightforward process: many animations in the current collection have been produced by students in one semester courses.
The supporting toolkit contains classes which, in addition to animating basic structure, provide many subsidiary displays - histograms, graphs, *etc*. which are updated as the algorithm runs. This toolkit has evolved following critical analysis of the features needed to allow easy comprehension of the animated algorithms, for example, a separate commentary area has been abandoned in favour of labels on the animation and 'history' panels have been added to permit simultaneous display of several steps in the algorithm. smaller improvements. Over several years, we have implemented a set of widely referenced algorithm animations commonly taught in data structures and algorithms courses available on the web.

Keywords: Algorithm animation, algorithms, data structures

1 Introduction

Computer science and software engineering courses generally include a core course on discrete data structures and algorithms to manipulate them. These courses, which are usually found in the second year with optional advanced courses in third and later years, aim to show students efficient ways to solve common problems. The discrete nature of the structures makes the visualization of the algorithms operating on them relatively easy to design: algorithm steps generally add or delete objects or rearrange an existing structure in some way. Many of the algorithms involve novel ideas that present conceptual difficulties to students and thus visualization represents a valuable additional pathway to understanding. Many algorithm animations have been written: they range from custom-coded animations of a single algorithm - usually available on Web sites in support of individual courses(Kitchen 2004, Mitra 1999, Burgiel & Raymond 2004) - to complex systems to support generation of algorithm animations(Roessling & Freisleben 2001, Rodger 2002, Brown & Najork 1996). The present work describes a technique based on adding animation directives to the source code of the algorithm being animated. A toolkit of commonly used objects and methods to manipulate them is an essential part of this technique. This strategy has been used to generate a collection of animations to support a second year course in data structures and algorithms for engineers. The system described here includes direct support for a number of novel features which have been found useful to support the teaching goals: this paper's purpose is to highlight a different approach to generating algorithm animations and several other features of the underlying animation toolkit and framework.

2 Background

A full set of Web notes for a broad data structures and algorithms course which included searching and sorting as well as simple graph algorithms and hard problems preceded the animation exercise. This course was designed for software engineers rather than computer scientists[1], so the emphasis was on understanding the choice of data structure and algorithm to solve any particular problem rather than, for example, details of the code for any particular algorithm. The existing HTML notes meant that the animations should be incorporated as seamlessly as possible into the existing notes. Java applets satisfied this aim as well as providing portability. The first applets were written for this project using the original Java AWT graphics package because Java's Swing graphics package had not been released at that time. However, when Swing became available, a decision was made to ensure portability by continuing to use the original AWT package. Some minor benefits could be obtained by switching to Swing now, but in deference to users who have never seen a need to upgrade their browsers, the project continues to use AWT only. This has not significantly hampered attempts to extend the package into other areas (control, statistics, computer architecture *etc*.), so, while AWT continues to be supported by new releases of the Java Virtual Machine, it is unlikely that the original decision will be changed.
In order to allow general instructions, commentary, theoretical considerations, *etc*. to be readily available while

[1] In this context, the distinction is between users of algorithms and designers of new ones.

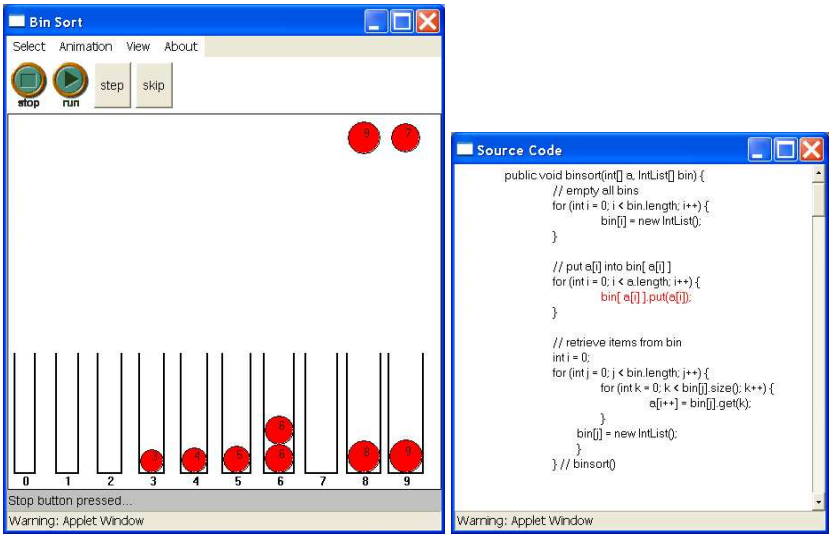

Figure 1: Bin Sort Animation: left - main drawing canvas; right - source code

the animation is running, the animation does not use up screen space in a web page: it is activated from a simple button embedded in a web page and a separate animation window is created, see figures 1, 3, 4 and 5. A standard screen format has evolved with several menu options for selecting data sets, setting animation parameters, viewing source code and the obligatory 'About' option and tape-recorder style buttons for controlling the animation. The major part of the window is the algorithm canvas. A small text area for commentary (discussed later, *cf.* Section 4.2) appears at the bottom. Using a separate frame for the animations obviated the need to provide facilities within the animation subsystem for help and documentation: web pages are an excellent medium for the text (and images if necessary) needed for this purpose. This also reduces the learning curve for an animation writer: it is assumed that he or she can already write HTML pages, so this knowledge is leveraged and no other documentation system needs to be learnt. Two threads are created - one to monitor the control buttons and a second to actually run the code (plus animation directives) of the algorithm.

2.1 Algorithm code

Firstly, an animator must write or obtain working Java code for the algorithm itself. The animation tasks can now be split into several subtasks: drawing the basic data structures, deciding where to 'break' the animation and display an updated structure and determining whether auxillary structures (graphs, *etc.*) are needed.
Invocations of methods on the class modelling the data structure(s) being animated are inserted into a standard framework. Animation directives are then *added to the source code of the algorithm being animated.* A common and useful feature of algorithm animations is the simultaneous highlighting of the line(s) of code being executed as the algorithm progresses. Adding the animation directives to actual working code makes realization of this aim simple, but the animation directives are 'noise' to a student wishing to understand the implementation of the algorithm. In order to allow the executing code to be highlighted without this noise, the animation directives follow an easily parsed symbol (actually a legal comment `/*-*/`) inserted into the code: see the example in Figure 2. The module which displays source code performs a simple parse of the animated code, stripping out the directives following the special symbol and displaying the core algorithm code only.

This makes the animator's task considerably easier: there is no major animation design phase - the standard framework and the toolkit takes care of that. Once a working algorithm is coded following some simple design rules that would usually be followed by well engineered code, the animator focusses on how best to illustrate the transformations of the various structures as the algorithm progresses. This mainly involves deciding where to insert the 'update' directives (which will draw updated structures) into the code. Provision is also made for minor and major steps within the code: this allows a student who has understood some of the basic steps to skip over them (perhaps in a re-run of the same animation) until a new operation is encountered. Classes for critical structures are extended with `draw` methods so as to implement the defined `DrawingObj` interface. For some commonly used structures such as tree and graph nodes, the toolkit already provides suitable classes which may be sub-classed if necessary for a particular algorithm. For example, a basic tree node (and tree drawing methods) is provided which was sub-classed to produce nodes suitable for red-black and AVL trees.

2.2 Toolkit

The supporting toolkit is the real key to efficient and fast generation of new animations; it has grown as animations were written and now contains classes for handling trees, graphs, matrices, histograms, labels, legends, arrows, tables, *etc.*To support the dynamic visualization of structure rearrangements, it allows a `Trajectory` to be specified - allowing, for example, animation of the path 'followed' by a node being added to a binary tree. Almost all the existing animations use this class in some way.
Methods are also provided to mark step boundaries at various levels. The thread controlling the animation progress 'advances' the animation from step boundary to step boundary at a rate set by the user. A student may request a 'fast forward' to the next major step boundary to speed up animation over uninteresting or well understood segments.

3 Animation Framework

A standard framework, the `AlgAnimFrame` class, provides most of the basic drawing and animation control structures: it produces the standard window seen in figure

```
/* Rotate node x to left */
private void RotateLeft( RBNode x, Graphics g ) {
  /*-*/ Color origColour = x.getColour(); x.Highlight(g, Color.yellow);
  /*-*/ AlgAnimFrame.showText( 3, "Rotate Left about " + x.getWeight() );
  /*-*/ addLabelNear( rot_left, x ); drawRBTree(); AlgAnimFrame.pauseStep();
  RBNode y = (RBNode)x.getRight();
  /* establish x.right link */
  x.setRight(y.getLeft());
  if ((RBNode)y.getLeft() != sentinel)
    ((RBNode)y.getLeft()).SetParent(x); /*-*/ tl.reLink( y, y.getLeft(), x, y.getLeft() );
```

Figure 2: Fragment of code from red-black tree animation showing animation directives following `/*-*/` symbols

5[2]. The animation is set up by writing a specialization of the `AlgThread` class - usually by editing any one of several existing specializations, *e.g.* `RBTreeAlgThread`. This class contains lists of example data sets, invocations of constructors for the major structures and invocations of the methods being animated. In the case of interactive animations, user input is controlled by the `AlgThread` specialization.

In addition to speeding animation generation, the standard framework, with its common operation modes, avoids the need for elaborate user documentation: once operation of one animation is understood, the remainder follow the same pattern. In fact, the animations have been extremely popular despite a lack of any real effort by myself and others who wrote animations to write any user documentation at all!

4 Experience

The animations have been in use for several years now so there has been ample opportunity to observe their use and collect feedback. They have been extremely popular: there are known links to them on dozens of other web sites and many local copies of all or parts of the original UWA PLSD210 course notes are kept in universities throughout the world. Hundreds of unsolicited emails have been received from grateful students who gained from the notes and the accompanying animations.

Curiously, despite the large volume of email, very few suggestions for change have been received: the most persistent one is a request to allow user input of sample data sets. This was introduced as a trial into the animation of red-black trees(Morris 2002) to allow nodes to be added to or deleted from a basic tree. Although many other animations provide this capability, it was deemed of relatively low importance for the animation of computer science algorithms. *It was considered more important to ensure that all the important cases were demonstrated by test data*. The danger with user input, in common with all software testing, is that users may unwittingly input representatives of the same equivalence class each time they run an animation - and thus not see many of the cases which it is important to observe. For example, the handling of sorted data is important to understanding the properties of quick sort and the enhancements that are needed to the basic algorithm to maintain its good performance. Without prompting, how likely is it that a casual experimenter would enter already sorted data into a *sorting* algorithm? There is some probability that a curious student will try a data set in reverse order - lucky for one learning quick sort's behaviour because it has the same pathological behaviour as sorted data. However, for insertion sort, sorted and reverse sorted data have quite different performances, so this phenomenon might easily be missed. Thus the approach with the current set of animations has been to provide a selection of data sets which exercise all the cases which an algorithm may have to process. It is felt that this is more likely to result in complete understanding of an algorithm's behaviour: as long as the student tries every provided data set, then he or she will be exposed to all the nuances of an algorithm. Whilst it is highly likely that a thorough student will explore each provided data set, we would be relying on a student's fortuitous combination of luck, prior knowledge or reading ahead[3] or experience with similar problem to 'discover' all the relevant cases. With some algorithms, *e.g.* the balanced tree algorithms such as red-black trees, the 'path' to some of the interesting cases is quite difficult to discover, making it further unlikely that it will be observed through random experimentation. Whereas a single data set found in Cormen's text(Cormen, Leiserson & Rivest 1993) and incorporated in the animation will exercise all the different cases for the red-black tree algorithm. Presenting this data set as the first one that a student should try ensures that all balancing cases are seen. However, as this position is at odds with some educational theories about learning by self-discovery, it will remain contentious!

4.1 History Panels

Many algorithms involve complex rearrangements of relationships between objects in a single step, *e.g.* red-black tree re-balancing requires addition of a node, colour changes and one or more rotations. The animator has a number of choices:

1. Slow the process down so that each 'step' is represented as a new image - allowing each microstep to be displayed individually,
2. Try a composite approach in which arrows, labels, *etc.*are added to highlight each of the micro-steps,
3. Allow stepping backwards *or*
4. Present the progress of the algorithm in a number of panels - allowing the student to see 'before' and 'after' images.

Approach 1 is the 'standard' approach: most systems permit the update rate to be varied by the user by either controlling the rate in a continuous fashion or by an action which steps to the next significant event. This approach can leave a user confused when a non-obvious step was taken - an ability to 'rewind' is needed. With simple structure changes, approach 2 may solve the problem, but complex rearrangements will require too much clutter in the form of added labels, arrows, *etc.* for this to be effective.

[2] The other figures show earlier versions of the framework that has now been adopted. Those animations will be trivially updated to use the new framework as soon as any maintenance is performed on them!

[3] Feedback from students in class suggests that the animations are best presented first! The informal understanding of concepts gained from viewing the animation aids the formal study that follows.

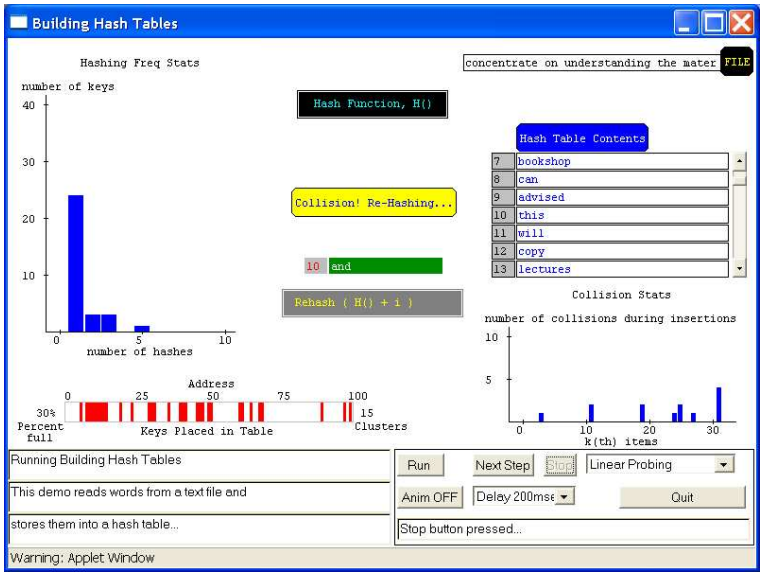

Figure 3: Hash Table Animation: A large complement of tools may be seen in this animation!

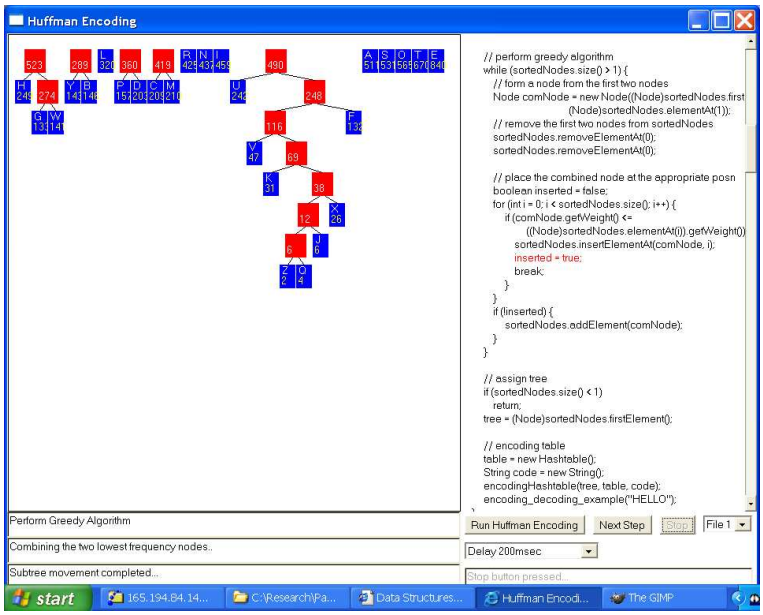

Figure 4: Huffman Encoding Animation - probably the most popular in this set!

ANIMAL's player provides the ability to return to the previous step or to start again(Rößling & Naps 2002). Again, this approach will work with simple changes: but as soon as a major processing step is composed of several smaller steps, the animator is faced with a difficult design decision: skip over smaller (or amalgamate) small steps so that the major step is visualized or focus on the micro-steps at the expense of visualizing the major step. The approach taken here allows the animator to specify the number of history panels - each displaying a previous step in the algorithm - to be displayed, see Figure 5. This allows the animator to arrange displayed steps so that the left panel shows the state before a major step commenced, intermediate panels show the micro-steps and the right panel shows the state after the major step has completed.

4.2 Commentary

Initially, a text area with an adjustable number of lines was placed below the animation canvas to provide a running commentary: original commentary is still visible in Figure 3, 4 and 5. User feedback soon showed that this was not effective: it was difficult to associate a new text message at the bottom of the screen with events on the main canvas while the animation was running. Thus a `ShadowLabel` class was written to allow text comments to be placed on the animation canvas - right next to the object to which the comment referred, see Figure 3 and 5. An `addLabelNear` method avoids the need for the animator to deal with the painful and trivial tasks of adjusting the fine position of these labels - they follow the object with which they're associated as needed.

This seemingly simple change has had a dramatic effect on the on the usefulness of animations built once the problem was discovered and was a valuable lesson in animation design.

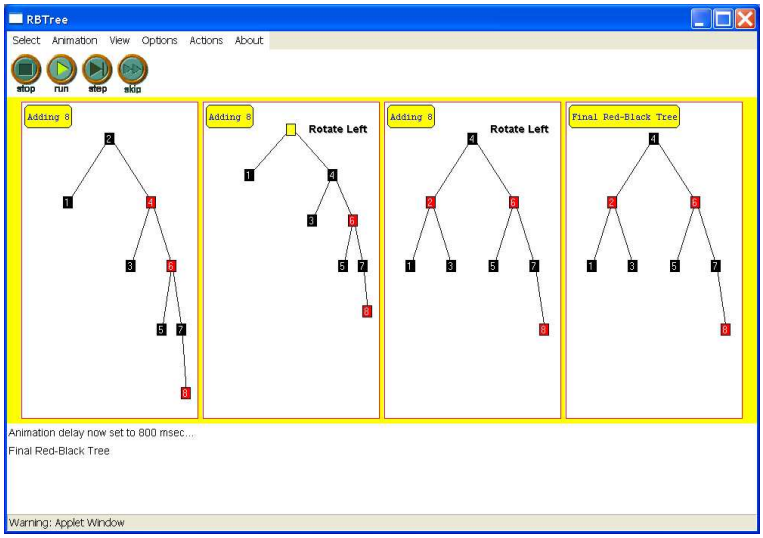

Figure 5: Red-Black Tree Animation showing a set of history panels which enable the student to study complex transformations of relationships between individual objects.

5 Other Domains

The basic framework and toolkit has been used in several projects to produce animations outside the original domain of computer science algorithms: these have included - animation of control algorithms, animation of data set descriptors (for statistics teaching) and processor simulation. In all cases, the basic framework was used with no alteration: animation programmers simply replaced the animation thread itself. Various display objects from the existing toolkit (labels, trajectory animation, *etc.*) were used as the individual animations required. New classes were written in standard Java as necessary and added to the toolkit if they were considered to have some general applicability: thus the power of the system has grown incrementally with new capabilities being readily added. One advantage of the common framework is that new capabilities such as the 'history' panels are readily added to older animations.

6 Conclusion

The benefits of algorithm animations as an adjunct to more traditional teaching methods has been clearly established(Naps, Rossling, Almstrum, Dunn, Fleischer, Hundhausen, Korhonen, Malmi, McNally, Rodger & Velazquez-Iturbide 2003). The algorithms produced by this system have proved extremely popular with students around the world: entering 'data structures and algorithms' into a search engine routinely returns links to the pages in which they are embedded at the top of the hit list. (It also generates a large number of requests to solve homework problems from this international class!)
Three factors in the design of the system have been key to the rapid production of a family of animations:

- Driving the animations from source code of the algorithm itself makes designing the animation straightforward and fast.
- The toolkit approach provides a low learning curve for an animator: there is no new language or system to learn.
- The standard framework allows a programmer to produce a prototype rapidly.

The last point is a significant one: ideas about presentation of the algorithms evolved over the life of the project (and will almost certainly continue to evolve). Evolutions fall into two groups. The first is applicable to animations generally: for example the original (obvious!?) requirement for a commentary area turned out to be a bad idea and, although it is still available, has tended to be used less and less. Similarly, adjustment of the rate of animation and the provision of a step capability were found to be inadequate for effective presentation of more complex scenarios and the history panels have been added. The second group is algorithm specific: for example the original quick sort animation used a conventional partition-in-place algorithm, which has a confusing display. A discussion with Jeff Rohl led to the much more easily comprehended dataflow-style partition into low and high groups in the current quicksort animation(Ang & Morris 1997).
The basic framework and toolkit has also shown its versatility by being used in some domains outside its original target: control systems, statistics and processor simulation.

7 Acknowledgements

John Morris was supported by the Foreign Professors Invitation Program of The Korean IT Industry Promotion Agency at Chung Ang University in 2003-4.
A large number of the animations currently available on the web were programmed by Woi Ang who also wrote much of the current infrastructure. Many students have now contributed to individual animations: their names appear in the Web pages(Morris 2004) beside the buttons activating animations to which they have made contributions.

References

Ang, W. & Morris, J. (1997), *Quick sort animation*, `www.cs.auckland.ac.nz/software/AlgAnim/qsort.html`.

Brown, M. H. & Najork, M. A. (1996), Collaborative active textbooks: A web-based algorithm animation system for an electronic classroom, *in* 'Proceedings of the IEEE Symposium on Visual Lan-

guages', IEEE Computer Society Press, Washington, pp. 266–275.

Burgiel, H. & Raymond, M. (2004), *Simple N-gon Counter*, `www.math.uiuc.edu/~burgiel/Java/Ngons/`, (accessed Nov 2004).

Cormen, T. H., Leiserson, C. E. & Rivest, R. L. (1993), *An Introduction to Algorithms*, The MIT Press.

Kitchen, A. (2004), *Sorting algorithms*, `www.cs.rit.edu/~atk/Java/Sorting/sorting.html` (accessed Nov 2004).

Mitra, S. (1999), *Animation of Sorting Algorithms*, `www.cs.brockport.edu/cs/javasort.html` (accessed Nov 2004).

Morris, J. (2002), *Red-black Tree Animation*, `www.cs.auckland.ac.nz/software/AlgAnim/red_black.html`.

Morris, J. (2004), *Algorithm Animations*, `www.cs.auckland.ac.nz/software/AlgAnim/alg_anim.html`.

Naps, T. L., Rossling, G., Almstrum, V., Dunn, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Velazquez-Iturbide, J. A. (2003), Exploring the role of visualization and engagement in computer science education, *in* 'inroads - Paving the Way Towards Excellence in Computing Education', ACM Press, pp. 131–152.

Rodger, S. H. (2002), Using hands-on visualizations to teach computer science from beginning courses to advanced courses, *in* 'Second Program Visualization Workshop'.

Roessling, G. & Freisleben, B. (2001), ANIMALSCRIPT: An extensible scripting language for algorithm animation, *in* 'Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Sciense Education (SIGCSE-01)', Vol. 33 of *ACM Sigcse Bulletin*, ACMPress, New York, pp. 70–74.

Rößling, G. & Naps, T. L. (2002), A testbed for pedagogical requirements in algorithm visualizations, *in* D. Finkel, ed., 'Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE-02)', Vol. 34 of *SIGCSE Bulletin*, ACM Press, New York, pp. 96–100.