Using Student-Built Algorithm Animations as Learning Aids

John T. Stasko

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E-mail: stasko@cc.gatech.edu

Technical Report GIT-GVU-96-19
August 1996

Abstract

The typical application of algorithm animation to assist instruction involves students viewing already prepared animations. An alternative strategy is to have the students themselves construct animations of algorithms. The Samba algorithm animation tool fosters such student-built animations. Samba was used in an undergraduate algorithms course in which students constructed algorithm animations as regular class assignments. This article describes Samba and documents our experiences using it in the algorithms course. Student reaction to the animation assignments was very positive, and the students appeared to learn the pertinent algorithms extremely well.

1 Introduction

An algorithm animation is a dynamic graphical depiction of the data and operations of an algorithm. The animation's purpose is to illustrate how the algorithm functions to someone seeking to learn the algorithm, such as a student in a computer science class. Because algorithms can be challenging to learn and understand, it is hoped that the graphical depiction can make the algorithm more concrete, and easier to comprehend for the student.

A number of algorithm animation systems have been built over the last ten years. Just a few of the better known and more widely used systems include BALSA[Bro88a], Tango[Sta90], Zeus[Bro91], and AACE[Glo92]. These systems typically have been used to create animations to accompany a lecture in an electronic classroom, or to prepare animations for students to observe and interact with outside the classroom. The animations utilized for education are usually prepared a priori and shipped with a system or are prepared by the instructor(s) for the class using the animations.

Although algorithm animation systems have been utilized by many different schools as instructional aids in this manner, it is a fair to say that algorithm animation has not achieved the application and use in instructional settings hoped for by system developers. Certainly one reason behind this is the technical requirements and availability of computing equipment for presenting algorithm animations. Another reason may be reluctance on the part of instructors to adopt algorithm animations for their classes. This may stem from the time required to prepare and deploy animations, as well as questions about the pedagogical value added by the animations.

Correspondingly, recent research has been conducted to empirically evaluate the pedagogical value of students viewing and interacting with prepared algorithm animations. Results have ranged from positive[LBS94] to relatively little benefit from the animations[SBL93, BCS96]. In virtually all of these studies, the pure numbers (direction) of the results have favored the animations, however. Unfortunately, in the majority, the positive result for the animation was small and not statistically significant.

These results and prior use of algorithm animations in classes have motivated us to seek alternative ways of utilizing algorithm animations in instructional settings. We always have believed that having the students become fundamentally more involved with the animations would be beneficial. This belief led to a question: What if the students built the animations of algorithms themselves, as opposed to simply interacting with animations already prepared for them? Perhaps the students would benefit most by constructing the mapping from a new, unfamiliar algorithm to a concrete presentation of its operation.

This article describes a system we have built that fosters student creation of algorithm animations, and it describes our use of the system over the past few years. This alternative application of algorithm animation provides unique, new opportunities for instructors in a variety of computer science classes.

2 Animation System

An algorithm animation system designed to foster student construction of animations involves certain key requirements. First, the animation system must be very easy to learn and use. If building an animation is too complex and burdensome, the construction process will

overwhelm the students and subjugate learning the algorithm, the primary goal, to simply implementing the animation. A second key requirement is that developing the animation be intimately tied to the algorithm and its operations. By constructing the animation, students should uncover the fundamental attributes and characteristics of the algorithm.

We created the *Samba* algorithm animation tool to meet these two requirements. Samba is an interactive animation interpreter and generator that also can be used in a batch mode. It takes as input a sequence of ascii commands, one command per line. Each command has a number of different fields. The first field is the command type (a simple string), and it uniquely defines the number of trailing fields in the command. One set of commands create graphical objects for the animations. The two commands below, for example, create a line and circle respectively.

```
line l3 0.1 0.1 0.2 0.2 green thin
circle 27 0.8 0.7 0.1 red half
```

The second field of these commands, *l3* and *27*, specifies a unique string identifier for the graphical object being created. This is necessary for subsequent action commands that reference and modify the object. Trailing fields specify the visual attributes of the object. For example, the circle command provides center x and y coordinates, radius, color and fill style, in order. For flexibility, Samba animations are carried out in windows with floating point world coordinates (mapped to pixels by Samba) beginning with $\langle 0.0, 0.0 \rangle$ in the lower left and $\langle 1.0, 1.0 \rangle$ in the upper right. Commands exist to modify window coordinates thereby supporting panning and zooming.

A second set of commands modify already existing graphical objects. These commands provide the action of the animation. The two commands below move the line and change the color of the circle created earlier.

```
move l3 0.5 0.6
color 27 blue
```

Object movements in Samba can be discrete jumps to a new position or smooth, continuous actions. The `move` command by default interpolates 20 intermediate frames to traverse the distance specified in the command. These continuous motions assist viewers to monitor the animation and track changes in the display[Sta90].

Samba allows animation designers to utilize multiple animation views (windows) to illustrate an algorithm. A `viewdef` command with a trailing view name instantiates a new animation window. Then, when Samba encounters a subsequent `view` command with that name, all following animation commands (up to the next `view` change) are directed to that view.

Samba also supports explicit concurrency of animation commands. It provides two commands, denoted { and }, to begin and end the concurrent initiation of any number of other animations commands. Samba will "store up" all commands found between a pair of these brackets and carry out their actions concurrently.

Samba includes many other commands, thereby supporting rich and complex animations. Nonetheless, the primary goal in designing the Samba language was to keep it as simple as possible so that it could be learned and understood quickly. We believe that Samba is successful in this regard. The Samba command set continualy evolves too. A

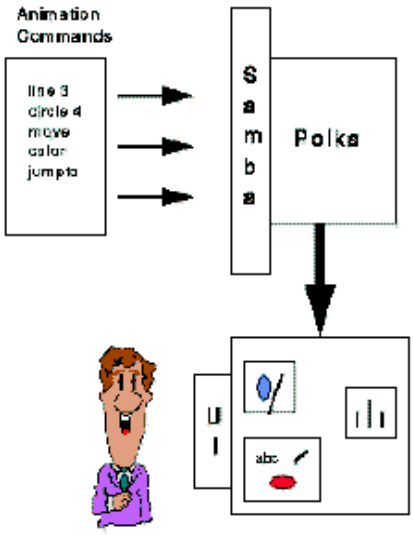

Figure 1: Overview of the architecture for Samba.

new command is added given sufficient request for some new functionality, as long as the primary goal of simplicity is not compromised.

Samba is actually just an application program of the Polka algorithm animation system and environment[SK93]. Samba provides a simpler, more accessible front-end to Polka, which requires C++ programming and has a steeper learning curve. Samba supports about 90% of the base functionality available in Polka. Figure 1 presents an overview of the architecture of the environment.

The code samples below better illustrate how Samba and Polka relate. The Samba movement command

```
move r1 0.5 0.5
```

which moves the graphical object named *r1* to coordinate $\langle 0.5, 0.5 \rangle$, has equivalent Polka code

```
int
AnimView::mover()
{
  int len;
  Loc *from,*l;
  Action *a;

  from = r1->Where(PART_C);
  l = new Loc(0.5, 0.5);
  a = new Action("MOVE", a, &l, STRAIGHT);
  len = r1->Program(a, time);
  time = Animate(time, len);
  return(len);
}
```

The actual Polka code in the Samba program for handling a `move` command must be even more general. It is shown below

```
int
InterpAnimView::moveScene(char ident[], double tox, double toy)
{
  Loc *original, *dest;
  int len;
  char *whatisit;
  AnimObject *AO;

  if (AO = VItems.Retrieverecord(ident)) {

    whatisit = (char *)AO->RetrieveData();

    if ((!strcmp(whatisit, "rect")) || (!strcmp(whatisit, "ltxt")))
      original = AO->Where(PART_SW);
    else
      original = AO->Where(PART_C);

    dest = new Loc(tox, toy);

    Action Movem("MOVE", original, dest, STRAIGHT);

    len = AO->Program(time, &Movem);

    return len;
  }
  else {
    cerr << "Warning:  Tried to MOVE nonexistant id.  (Ignoring)" << endl;
    return 0;
  }
}
```

Samba currently runs on top of the X Window System and Motif. Ports to PC Windows and Java are underway. For more details on the system functionality and all the commands, see `http://www.cc.gatech.edu/gvu/softviz/algoanim/samba.html`.

Developing Samba Animations

How do students utilize Samba to build animations of algorithms? Essentially, a student must annotate the implementation of an algorithm with "print" statements to generate the commands to drive Samba. When the program executes, these print statements will be output in an order corresponding to the execution and will comprise a trace of the program's operations. The print statment annotations correspond to the interesting events[Bro88b] often used in algorithm animation systems. This output trace is then forwarded to Samba which generates the specified animation.

For example, to animate a sorting algorithm, a student may embed two movement commands surrounded by concurrency brackets within the fundamental comparison-exchange loop of the sorting algorithm's implementation (shown here in C):

```
for (j=n-2; j>=0; --j)
    for (i=0; i<=j; ++i)
        if (a[i] > a[i+1])
           { temp = a[i];
             a[i] = a[i+1];
             a[i+1] = temp;
             printf("{ \n");
             printf("moveto %d %d\n",i,i+1);
             printf("moveto %d %d\n",i+1,i);
             printf("} \n");
             printf("swapid %d %d\n",i,i+1);
           }
```

These annotations simply make the two graphical objects (represented by string identifiers for their array indices) move to each other's position. The `swapid` command is necessary so that subsequent object references now access the correct swapped array object.

The transmission of commands to Samba can be interactive or it can be through a stored trace file. For example, if a program named `shellsort` is implemented, the program can pipe its output to Samba:

```
% shellsort | samba
```

Alternatively, the shellsort program can be run and its output stored in a file. That file can then be given to Samba as input. This latter method has the advantage that the Samba animation can be repeatedly viewed without exiting the system.

```
% shellsort > tracer
% samba tracer
```

Samba's user interface allows viewers to step one command at a time, if desired, or to pause at any point.

A key strength of Samba is its use of simple ascii commands as input. Students do not need to learn a complex animation methodology and system. But most importantly, an algorithm implemented in *any* language can be animated because the implementation simply needs to output the ascii commands. This is critical for early undergraduate computer science courses in which the students may only know a beginning language such as Pascal.

3 Use in the Curriculum

I have used Samba and a predecessor tool, the *Animator* program of the XTango system, three times to help teach algorithms in CS 3158, Design and Analysis of Algorithms, a required course for computer science majors at Georgia Tech. The course was primarily

composed of juniors, but some sophomores and seniors also were in the class. A few non-majors filled out a typical enrollment of about 45-50 students per offering. The course material is quite theoretical and conceptual, focusing on analysis techniques, asymptotics, recurrence relations, summations, and design methodologies such as divide-and-conquer, greedy approaches, and dynamic programming. The course introduces many algorithms with a particular focus on graph algorithms. Grading has been based on weekly homework assingments and a mid-term and final examination.

A theoretical, almost mathematical, course such as this is usually not a particular favorite of students, who often struggle with the abstract nature of the material. Students in the course reacted with a bit of uncertainty and curiousity when they were told that the course would involve not only programming, but also construction of animations.

About two weeks into the ten week quarter, I introduced the Samba tool for about 20 minutes in one class. Students appeared to easily grasp the basic concepts in this short time. Students also were given the documentation (primarily command summaries), a 5 page document.

I used three animation assignments per course during in the last two times I have taught CS 3158. The first assignment was called "Flying Logos": Students built an animation that involved their names flying around amidst some creative, interesting graphics. The assignment really had no algorithm to animate; it was designed as a gentle introduction to help the students learn how Samba works before they began their first true algorithm animation. The assignment was worth only 1% of their final grade.

The next two assignments involved actual algorithm animations. The students were required to implement the algorithm in a language of their choice and to build an animation of the algorithm. Each assignment was worth 10% of their final grade, about twice the value of a weekly homework assignment, and was graded according to how well the animation illustrated the algorithm, as well as the expressiveness, insight, and creativity of the animation. The students were told to develop an animation that would help explain the algorithm to a person not familiar with it. The two algorithms animated by students were the quicksort algorithm and a minimum spanning tree (MST) algorithm. On the MST assignment, half of the class utilized Kruskal's algorithm and half utilized Prim's algorithm.

The animations created by the students varied quite a bit; many were quite clever and informative. A substantial number of the animations provided some basic imagery, with simple graphical objects representing the program's data. Some had significant embellishments, however, involving scenes such as flying spacecraft and aliens describing how the algorithm works.

Each time I taught the course, the animations produced by the students have exhibited a noticeable jump in sophistication over those from the prior offering. This is probably because the animations have become part of the local culture and students were aware of prior work. I did explicitly avoid showing the students in a class the animations of a prior class, but obviously, students did have ways of seeing other animations, if desired. In a particular class, I did make available the best animations from the prior assignment for the class to examine.

For the quicksort assignment, many students utilized the traditional blocks-view supplemented by illustrations of the indices of comparison. A frame from an example student's animation is shown in Figure 2. A particularly interesting facet of the quicksort animation

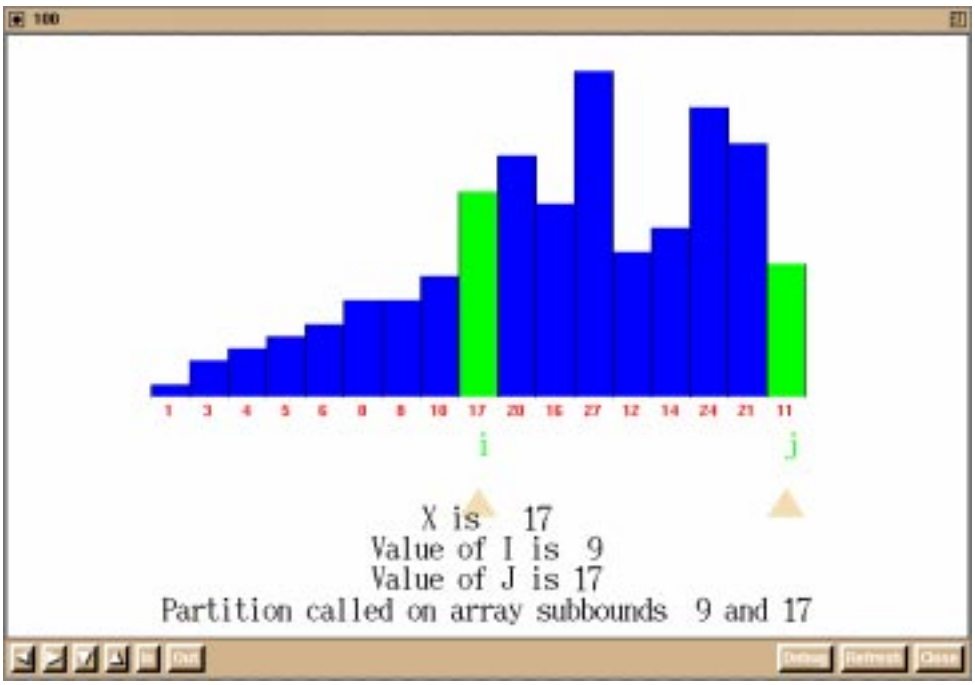

Figure 2: View of the quicksort algorithm created by a student.

was how the students illustrated the recursion of the algorithm. Noteworthy techniques included drawing boxes around the subarrays, using walls (lines) around the subarrays, or lowering subarrays in the window.

For the MST assignment, virtually all students utilize the traditional vertex-edge graph representation in some way. A particularly ambitious student also illustrated Prim's algorithm by illustrating the priority queue (implemented as a heap) used by the algorithm in another view. A frame from this animation is show in Figure 3.

For the quicksort assignment, all of the students first implemented the algorithm, then they added output statements to drive the animation. This changed a bit on the MST assignment, however. A number of students added output statements as they were implementing the MST program. Because this algorithm was sufficiently complex, the students used the Samba animation as a form of visual debugging aid. It provided them with feedback about how their program was executing.

This brief summary provides a flavor of the animations built by the students. To truly appreciate their work, however, one must actually view the different animations they produced. Now included in the system distribution are some of the more unique animations created by past classes.

Student reaction to the animation assignments was almost universally positive. The students appeared to enjoy the change of pace in this theoretical course. Georgia Tech undergraduates also generally seem to enjoy coding, as do students elsewhere, I suspect. The animations fed this interest.

In the last two offerings of the course, the students completed a survey at course end. Responses were anonymous. Students responded to four questions with an integer from 1 to 5, with 1 being *Strongly Disagree* and 5 being *Strongly Agree*. The results of these responses

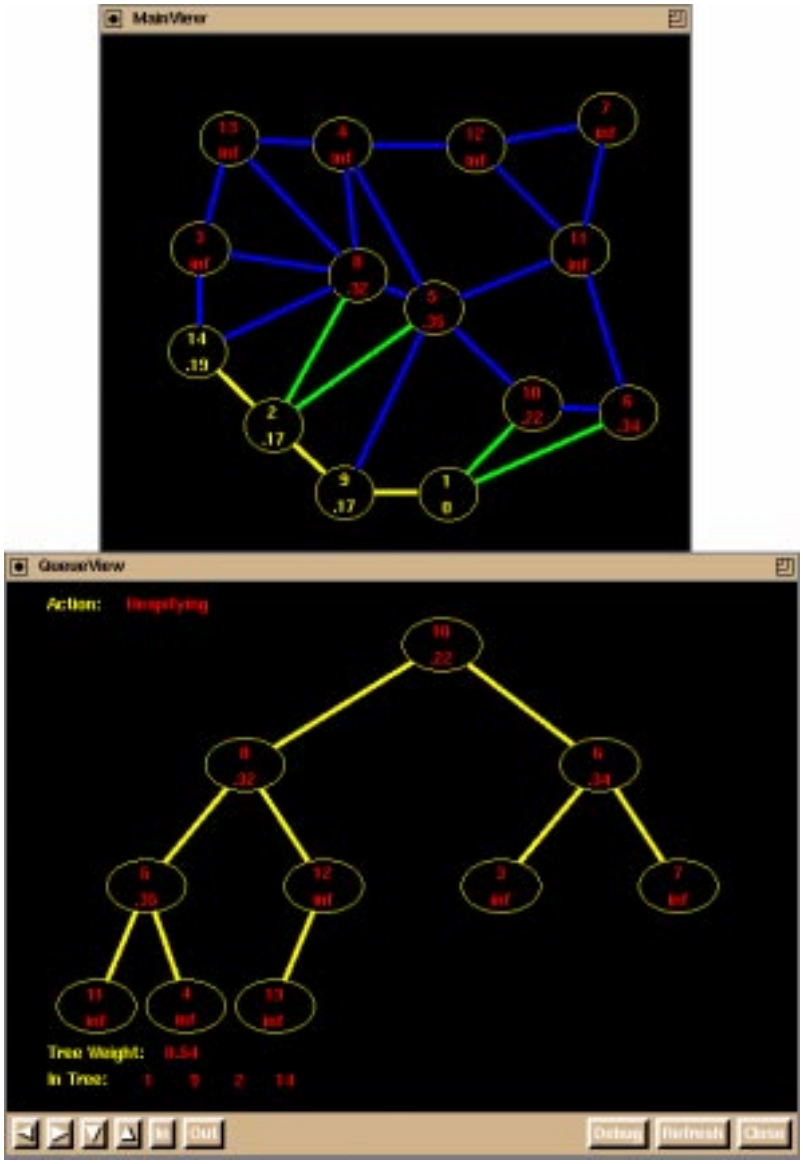

Figure 3: View of Prim's minimum spanning tree algorithm created by student.

are shown in Table 1. In general, the students felt that the animations were useful and helped them to understand the algorithms, were not difficult to create, and were fun.

	Winter '95 29 responses	Winter '96 22 responses
1. The animation assignments were a valuable learning experience.	4.34	4.05
2. The animation helped me understand the algorithms better.	4.21	4.05
3. The animation assignments were fun.	4.21	4.00
4. Creating an animation was difficult.	2.83	2.55

Table 1: Results of student surveys. Responses ranged from 1-Strongly Disagree to 5-Strongly Agree.

Perhaps even more informative, however, were student replies to free response questions on the survey. Below are a number of responses taken directly from surveys. Please note that these responses are not a few exceptions; they are representative of the cumulative set of replies.

`Q. In general, what was your opinion of the animation assignments?`

"They were cool."

"They were fun. It definitely helped me understand the algorithms more clearly when I was put in the position of having to explain it."

"Awesome–more of these and less of the homework in the future. I enjoyed doing the animations."

"They start out frustrating when one runs out of space on the screen and such, but they end up useful and informative."

"They were challenging but yet doable. After completing them, it gave you a good sense of accomplishment."

"The system was cool, but I did not think such assignments were appropriate for a theory course."

"They really make you think about the algorithm which helps to understand it."

"They were helpful to write programs to generate. That reinforced learning of the algorithms. Being graded on the creativity and utility for teaching of the animations doesn't seem relevant."

"Great! I wish more theory courses would use this approach."

"They were the only thing that helped my grade."

If a student felt that the animations were helpful, they were asked follow-on questions:

`Q. What aspects or things about the algorithms did the animations help you learn better?`

"I got a better understanding of how the pivot was chosen and how it migrated toward the middle."

"Not only did I have to know how to code it, I also had to think about how to relate it to another person—That made me really understand it in multiple dimensions."

"It helped me to visualize what is going on. I lose track if I try to trace stuff in my mind. The animations kept me from losing track."

"I didn't really understand the algorithm too well until I had to try to explain it to someone else, I used my ChemE roommate as a test dummy."

`Q. How did the animations help you understand the algorithms better?`

"I had a misconception of the recursive nature before the quicksort assignment."

"You could see exactly what was happening at each step without any abstraction."

"It is easier to understand things when you see them visually, at least it is for me."

"Having to set it so someone else will understand it puts you in a whole different frame of mind."

"They provide an overall view of the algorithm that cannot be seen on a code level."

Overall, the student responses were very positive. A number of students even requested more animation assignments. (Whenever students ask for *more* assignments, instructors take note.) Interesting among the responses above is the dissenting view in the first set of replies. One student did not feel that such assignments were appropriate for a theory course. A number of students also were unhappy about the subjectivity of the animation grading; it was a strong contrast from grading of the mathematically-oriented homework problems. A number of students noted how they felt that they understood an algorithm well, but that the animation uncovered inaccuracies and misperceptions in their understanding. Many others noted the positive benefits of having to explain (teach) through illustration how the algorithm works.

4 Related Work

As mentioned earlier, many different algorithm animation systems have been developed. Samba uses ideas and experience from all of this prior research. For a good summary of different algorithm animation and software visualization systems, see [PBS93].

In terms of similar approaches, Samba follows most closely the ANIM system by Bentley and Kernighan[BK91]. ANIM's focus contrasted with other peer systems such as BALSA and TANGO in that it provided a much more limited graphical vocabulary, but was accordingly simpler to learn and use. ANIM utilized an ascii file with simple scripting language directives as input. The basic system included `line`, `text`, `box`, `circle`, `view`, `click`, `erase`, and `clear` commands. Although this language was very basic, it was still powerful enough to be used to analyze the execution and performance of a number of UNIX system programs. Samba follows this approach and simply provides a richer graphical vocabulary including primitives for smooth animation.

The Goofy system by Ford provides an ascii command language on top of the Polka system as Samba does[For93]. Goofy is a richer, more complex language than Samba, however. It supports more of the base Polka functionality, even including timing parameters in commands. Samba does not include such functionality in order to make its language easier to understand and use.

We are not aware of similar educational applications of other algorithm animation systems as done with Samba, i.e., having students construct the algorithm animations themselves.[1] Other software visualization technology has been effectively utilized in educational settings, however. For instance, Naps utilizes the GAIGS system for laboratory visualization exercises[Nap90]. GAIGS produces views of data structures, and students use GAIGS to analyze algorithms, compare algorithm methodologies, and to suggest improvements in algorithms.

5 Discussion

After using the student-built animations for three classes, what stands out the most is how much the students enjoyed the animations. In a class often described by students as "dry" and full of "theoretical useless stuff", the animations provided a real change of pace. In a class with much analysis, the animations were able to engage students' creativity and expressiveness, something fundamental to the educational process. In fact, the students became quite competitive with each other, observing friends' animations and seeking to top their peers.

It also was clearly evident, however, that the animations helped the students to truly learn and understand the two algorithms under study. Feedback on the student surveys indicated this as students noted how they felt that they understood an algorithm well, but then discovered how their early conceptions were incorrect. Likewise, although an informal measure, student performance on final exam questions regarding the two algorithms was nearly perfect.

One possible explanation for this enhanced understanding is that students simply spent

[1] Instructors at other schools have used Samba and its predecessor system, *animator*, in this manner. Most noteworthy is use by Hartley[Har94].

a great deal of time working with these algorithms, much more so than on algorithms not given animation assignments. Certainly, that is true. But an accompanying question then is whether other assignments or activities could garner the same amount of time and attention from students and achieve the same level of enthusiasm. Would an assignment to write a report about an algorithm work equally well? The animation assignments provided a forum that encouraged diligence, study and creativity in the students, and did this in an engaging way. Is this not the fundamental objective of education?

Reflecting on the animation assignments allows us to hypothesize about why the assingments were helpful. Building an animation of an algorithm forces a student to identify the fundamental operations of the algorithm to be portrayed visually. The student must think about the data manipulated by the algorithm, the interactions occurring in the algorithm, and how the important high-level design concepts are manifested. This forces a student to move beyond worrying about "where the semi-colons go" to a more conceptual, fundamental reasoning. After that, however, the student must make these concepts concrete through some graphical depiction. The student constructs the algorithm-to-animation mapping and determines what is unique to the algorithm and deserves to be communicated visually. Fundamentally, building an algorithm animation forces a student to become the teacher, if only for a brief time. Instructors certainly know that the best way to learn a concept is to have to teach it.

Our experiences in using student-built animations in courses have been such that if the animations promoted absolutely no learning in the students, the assignments would still be valuable due to the enthusiasm and interest sparked in students. Happily, the animations did appear to foster learning and comprehension also, making them doubly useful.

One important follow-on needed now is formal empirical study to better identify the learning benefits from building algorithm animations, and to further characterize how and why those benefits occur. Would similar results hold if fully implemented programs were provided and the students were simply required to do animations? Does creating a "bare bones" animation provide the same benefits as creating an elaborate, intricate animation? What types of learners/students most benefit from this activity?[2] Can the animation design process be adapted to other domains such as physics, engineering and biology? The answers to these and other questions will help educators best utilize this new technology to maximal advantage in instructional settings.

Acknowledgments

Irwin Coleman assisted in implementing Samba. His work was greatly appreciated.

References

[BCS96] Michael D. Byrne, Richard Catrambone, and John T. Stasko. Do algorithm animations aid learning? Technical Report GIT-GVU-96-18, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, August 1996.

[2] An interesting note from our experiences is that the "best" students in class via traditional measures usually create the "best" animations.

[BK91] Jon L. Bentley and Brian W. Kernighan. A system for algorithm animation. *Computing Systems*, 4(1), Winter 1991.

[Bro88a] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[Bro88b] Marc H. Brown. Perspectives on algorithm animation. In *Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems*, pages 33–38, Washington D.C., May 1988.

[Bro91] Marc H. Brown. ZEUS: A system for algorithm animation and multi-view editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, Kobe Japan, October 1991.

[For93] Lindsey Ford. Goofy animation specification. Unpublished Report from the University of Exeter, Exeter, U.K., 1993.

[Glo92] Peter A. Gloor. AACE - algorithm animation for computer science education. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 25–31, Seattle, WA, September 1992.

[Har94] Stephen J. Hartley. Animating operating systems algorithms with XTANGO. In *Proceedings of the 1994 ACM SIGCSE Technical Symposium*, pages 344–348, Phoenix, AZ, March 1994.

[LBS94] Andrea W. Lawrence, Albert M. Badre, and John T. Stasko. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages 48–54, St. Louis, October 1994.

[Nap90] Thomas L. Naps. Algorithm visualization in computer science laboratories. In *Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education*, pages 105–110, Washington, DC, February 1990.

[PBS93] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.

[SBL93] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, pages 61–66, Amsterdam, Netherlands, April 1993.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Sta90] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

Appendix

This Appendix is the actual documentation about the Samba system that is given to students.

SAMBA Animation Designer's Package

This document describes the Samba program which provides an interpreted, interactive animation front-end to POLKA. Samba simply reads an ascii file, one command per line, in order to acquire its directions for creating an animation. This is beneficial because you can have the output of any program, be it Pascal, C, Modula-2, etc., drive an animation. I have used this tool in an undergraduate algorithms class. In addition to implementing an algorithm, students can develop an animation of it just by judiciously placing print statements into their program.

Animations developed with this system will be carried out in windows with a real-valued coordinate system that originally runs from 0.0 to 1.0 from left-to-right and from 0.0 to 1.0 from bottom-to-top. Note, however, that the coordinate system is infinite in all directions. You will create and place graphical objects within the coordinate system, and then move them, change their color, visibility, fill, etc., in order to depict the operations and actions of a computer algorithm.

The format for the individual commands is described later. You can run this program interactively by piping the output of your program to it, e.g.,

```
% yourprog | samba
```

If the textual output of your program was saved to a file (perhaps even on another computer system), you can simply have Samba read that file as input via a command line argument, e.g.,

```
% samba outfile
```

By using this method, you will be able to run your animation repeatedly without exiting Samba.

In order to build an algorithm animation, you need to augment your implementation of the algorithm under study by a set of output (e.g., printf, writeln, cout) statements. The statements should be placed in the program at the appropriate positions to provide a trace or depiction of what your program is doing.

Below we summarize the different commands that exist within the system. If there's a command you would like added, just ask. We are always looking to improve the system. To begin this section, we describe the commands in general.

Each command begins with a unique one word string. Make sure that you spell the strings correctly. Each graphical object that you create should be designated by a unique string id. You will need to use that id in subsequent commands that move, color, alter, etc., the object. In essence, the id is a handle onto the object. Most of the commands and their parameters should be self-explanatory. Arguments named `steps` and `centered` are of integer type. Arguments named `xpos, ypos, xsize, ysize, radius, lx, by, rx, ty` are real or floating point numbers. Make sure that they include a decimal point (typing 0 instead of 0.0 will cause an error). The argument `fillval` should be one of the following strings: `outline`, `half` or `solid`. The argument `widthval` should be one of the following strings: `thin`, `medthick`, or `thick`. The parameter `colorval` can be any color string name from the file `/usr/lib/X11/rgb.txt` (one word, no blanks allowed).

Note that any line with a per cent character (%) in column 1 is interpreted as an instructional comment—no command is carried out.

If you have used the Animator for the XTango system, you will find that the Samba system is very similar. In fact, Samba has been designed to be backward compatible with XTango Animator trace files. Samba simply adds new features to the base XTango Animator. What are these features? For one, all objects can be referred to by arbitrary string identifiers, instead of only integers. Samba also provides the option of multiple Views. In other words, users may have several animation windows open and running at one time, all with their own graphical objects and events. Samba also provides for explicit concurrency of animation events. Blocks of commands can be defined so that they all begin at the same point in time. The events need not end at the same time. Also, some additional objects have been added, which are detailed below.

Samba will print warning messages if you try to do erroneous actions like reuse an ID, try to delete or move an invalid ID, and so on. You can turn these warnings off by using the -O command line flag.

`comment A trailing string`

This command simply prints out any text following the "comment" identifier to the shell in which the animator was invoked.

`viewdef id <xsize> <ysize>`

This command defines a single view. It must be used to create multiple new views in an animation. The xsize and ysize parameters are optional specifications for the size of the window in pixels. View definitions must precede all other commands in a trace file (except for %). If no view definitions are found, then Samba will create default Samba view and perform all subsequent animation actions in it.

`view id`

This command sets the current view in which objects will be created or manipulated. That is, all trailing animation actions occur in this view until another `view` command is found. Note that objects are bound to the the view that was active at the time of their creation. Trying to manipulate objects from a particular view while another view is active will result in an error. The bottom line is: Change the view with this command before creating an object to put in that view, and change to that view to manipulate the object.

{

Begin a block of concurrent commands. All commands in the block will start at the same time index. Nesting of blocks are not allowed, but referencing different views within one block is allowed.

}

End a block of concurrent commands. Every { must have a matching }, and nesting is not allowed.

`bg colorval`

Change the background to the given color. The default starter is white.

`coords lx by rx ty`

Change the displayed coordinates (real valued) to the given values. You can use repeated applications of this command to achieve interesting panning or zooming effects in an animation view.

`delay steps`

Generate the given number of animation frames with no changes in them.

`line id xpos ypos xsize ysize colorval widthval`

Create a line with one endpoint at the given position and of the given size.

`pointline id xpos1 ypos1 xpos2 ypos2 colorval widthval`

This creates a line like the `line` command except that you provide the two endpoints of the line here rather than the line's size.

`rectangle id xpos ypos xsize ysize colorval fillval`

Create a rectangle with lower left corner at the given position and of the given size (size must be positive).

`circle id xpos ypos radius colorval fillval`

Create a circle centered at the given position.

`triangle id v1x v1y v2x v2y v3x v3y colorval fillval`

Create a triangle whose three vertices are located at the given three coordinates. Note that triangles are moved (for move, jump, and exchange commands) relative to the center of their bounding box.

`polygon id numsides colorval fillval xpos ypos vx1 vy1 ... vx7 vy7`

Create a polygon starting at (xpos, ypos) with up to seven other vertices specified by (vx1, vy1) ... (vx7, vy7). Specify any number of sides between three and eight with `numsides`.

`text id xpos ypos centered colorval string`

Create text with lower left corner at the given position if `centered` is 0. If `centered` is 1, the position arguments denote the place where the center of the text is put. The text string is allowed to have blank spaces included in it but you should make sure it includes at least one non-blank character.

`bigtext id xpos ypos centered colorval string`

This works just like the text command except that this text is in a much larger font.

`flextext id xpos ypos centered colorval fontname string`

This is the flexible text command, and it works just like the text command except that you can explicitly specify the font (string name) that you want to use. Any valid X11 font is allowable.

`set id num id1 id2 ...`

Create a set of already existing objects that can be manipulated as a group through a new ID. Specify the number of items in the set with the num parameter, and list the identifiers of the member objects in a space-delimited list. All manipulations of the set object affect the members of the set as well. All objects in the set must be in the same view. The set acts like a mathematical set, so duplicate identifiers in id1 on will be ignored.

`move id xpos ypos`

Smoothly move, via a sequence of intermediate steps, the object with the given id to the specified position.

`moverelative id xdelta ydelta`

Smoothly move, via a sequence of intermediate steps, the object with the given id by the given relative distance.

`moveto id id`

Smoothly move, via a sequence of intermediate steps, the object with the first id to the current position of the object with the second id.

`jump id xpos ypos`

Move the object with the given id to the designated position in a one frame jump.

`jumprelative id xdelta ydelta`

Move the object with the given id by the provided relative distance in one jump.

`jumpto id id`

Move the object with the given id to the current position of the object with the second id in a one frame jump.

`color id colorval`

Change the color of the object with the given id to the specified color value.

`alter id newstring`

Change the string presented for the given text object to `newstring`.

`delete id`

Permanently remove the object with the given id from the display, and remove any association of this id number with the object.

`fill id fillval`

Change the object with the given id to the designated fill value. This has no effect on lines and text.

`width id widthval`

Change the width value of the line with the given id to the designated fill value. This will not work with any other graphic object.

`vis id`

Toggle the visibility of the object with the given id.

`lower id`

Push the object with the given id backward to the viewing plane farthest from the viewer.

`raise id`

Pop the object with the given id forward to the viewing plane closest to the viewer.

`exchangepos id id`

Make the two objects specified by the given ids smoothly exchange positions.

`switchpos id id`

Make the two objects specified by the given ids exchange positions in one instantaneous jump.

`swapid id id`

Exchange the ids used to designate the two given objects.

The next three pages illustrate three sample input files to Samba. The first shows basic features (from the old xtango animator), the second illustrates some new Samba commands (basic ones) and the third illustrates multiple views and explicit concurrency.

```
% This example illustrates the more basic commands
% Note that it uses only integers as IDs, but in general,
% arbitrary character strings can be used
comment This is a sample animation script
circle 1 0.8 0.8 0.1 red half
line 2 0.1 0.1 0.2 0.2 green thin
rectangle 3 0.1 0.9 0.1 0.1 blue solid
text 4 0.0 0.0 0 black Hello
text 5 0.5 0.5 1 black RealLongStringandThenSomeAndEvenMore
circle 6 0.3 0.3 0.2 wheat solid
triangle 7 0.5 1.0 0.6 0.8 0.4 0.9 cyan solid
bigtext 8 0.2 0.2 0 black Some Big Text
moveto 1 6
moverelative 3 0.05 -0.4
jumprelative 4 0.4 0.4
raise 1
lower 1
color 6 blue
move 3 0.5 0.5
jump 3 0.9 0.9
jumpto 3 6
raise 3
fill 3 half
fill 3 outline
fill 3 half
vis 3
vis 3
vis 6
delete 8
line 200 0.9 0.2 0.0 0.6 black thick
bigtext 8  0.6 0.2 0   DeepPink    More Big Text
flextext 88  0.4 0.3 0   magenta  8x13bold    Flex Text 8x13bold
rectangle 12 0.7 0.7 0.1 0.1 green solid
exchangepos 12 3
exchangepos 12 3
exchangepos 12 3
exchangepos 12 3
switchpos 12 3
circle 99 0.8 0.8 0.15 black outline
exchangepos 1 99
bg pink
bg LemonChiffon2
coords -0.5 -0.5 1.5 1.5
```

```
% This example illustrates features not in xtango's animator

% Add a polygon
polygon poly1 4 purple solid 0.1 0.0 0.5 0.5 0.7 0.5 0.5 0.1 0.0 0.0

% Add a pointline
pointline liner 0.2 0.8 0.7 0.8 blue thin

% Add some text
text 12 0.6 0.6 1 black A text string

% Move the line down
move liner 0.08 0.3

% Change the polygon's color to cyan
color poly1 cyan

% Make the line thicker
width liner thick

% Change the text string
alter 12 Has just changed

% Make a set out of the polygon and line
set newset 2 poly1 liner

% Move them together
moverelative newset 0.3 0.3
```

```
% A simple animation for advanced feature demonstration.
% Initialize the views
viewdef MainView 600 600
viewdef SecondView 400 400
% Start animation
comment Switch to MainView.
view MainView
comment Draw a line there.
line thinblackline 0.2 0.2 0.5 0.0 black thin
comment Switch to the SecondView.
view SecondView
comment Draw a rectangle here.
rectangle redrectangle 0.5 0.5 0.3 0.2 red half
view MainView
comment Let's move the rectangle and the line at the same time.
{
view SecondView
move redrectangle 0.3 0.3
view MainView
move thinblackline 0.5 0.5
}
delay 15
comment Now, let's move them separately.
moverelative thinblackline -0.2 -0.2
view SecondView
moverelative redrectangle 0.2 0.2
comment Let's put in a polygon in SecondView.
polygon greenpoly 4 green solid 0.0 0.0 0.5 0.5 0.7 0.5 0.5 0.1 0.0 0.0
comment Let's alter the fill for the rectangle.
fill redrectangle solid
comment Let's get rid of that polygon.
delete greenpoly
view MainView
comment Let's make that thin line thick.
width thinblackline thick
view SecondView
text boxlabel 0.6 0.6 1 black Rectangle
comment Let's make the rectangle and its label a set.
set boxset 2 redrectangle boxlabel
comment Now we can move the set around like a single object.
moverelative boxset 0.5 0.5
```