Yet Another System For Algorithm Animation

Victor Shnayder

January 9, 2002

Abstract

Animating algorithms can often be useful for several reasons. I was unable to find any modern animation system that focused on ease of use over features, and thus built this system in an attempt to provide one. It is implemented in Java for portability, and uses ASCII input for programming language independence. Since the goal was ease of use, the feature set is intentionally fairly primitive.

1 Introduction

Animations of algorithms can be useful for several different reasons, but it has traditionally been difficult to obtain them. Education is a very important use—viewing an animation of what the algorithm does can sometimes be much more enlightening then reading the code alone. Debugging is another use—it is often hard to find bugs when an algorithm is dealing with large amounts of data, and visualizing it can often let the programmer see exactly what is going wrong. There are a fairly large number of systems out there for generating complex animations, so the focus in this system was to make something that could be used with only a few minutes training to produce animations of algorithms of no more then moderate complexity.

The main challenges involved had to do with design, not implementation. In particular, I had to find the right balance in terms of the number of features to implement, both in the input language and the User Interface (UI).

The rest of the paper is structured as follows: Section 2 describes previous work in this area, section 3 describes my approach to the problem, section 4 describes the implementation, section 5 presents my results, and section 6 concludes.

2 Previous Work

There has been a large amount of research into animating algorithms over the past twenty years or so. In this section, I will briefly describe several systems, ranging from very early

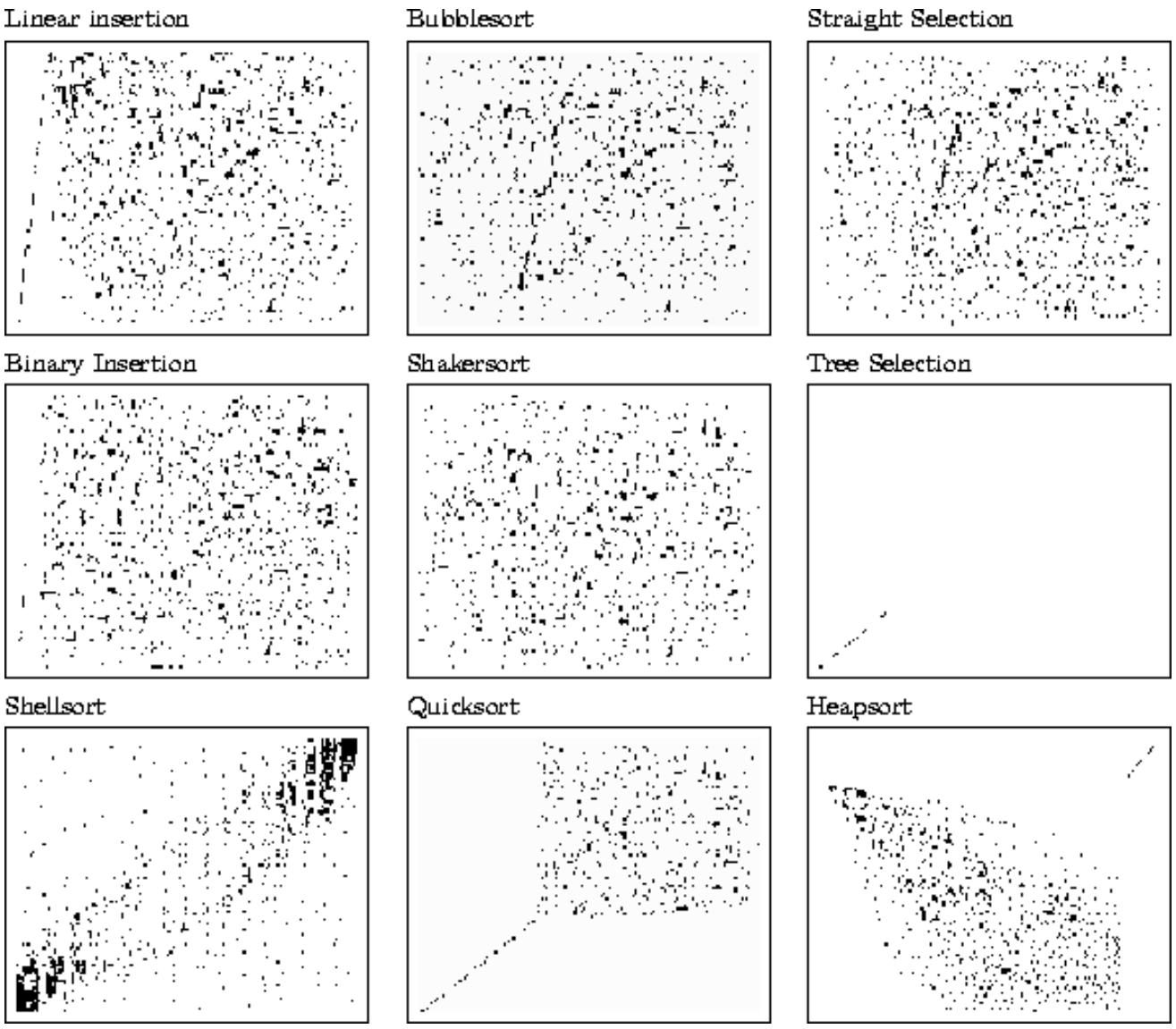

Figure 1: Sorting Out Sorting

ones to recently developed ones. A lot of the information in this section comes from Alejo Hausner and David Dobkin's "Introduction to the Animation of Geometric Algorithms" [1].

2.1 Sorting Out Sorting

Sorting Out Sorting[2] was a video, made in the early eighties, for teaching sorting algorithms. The video concludes with a comparison of nine different algorithms (See figure 1). The video was one of the first to illustrate the usefulness of algorithm animations. It also showed the need for general purpose animation systems, because the development of the custom animation software needed to produce it took several years of effort.

2.2 BALSA and Zeus

BALSA [6], developed by Marc Brown and Robert Sedgewick, was one of the first general purpose systems developed, and it has had a great deal of influence on the design of later systems. It introduced the notion of interesting *events*—steps in the algorithm that were of concern to the animation. It provided a framework that would call *renderers* at appropriate times during the execution of the program. However, the programmer was required to write the renderers himself to visualize his data, so generating animations still required a lot of work.

Zeus [7] was another system developed by Marc Brown that was in many ways similar to BALSA—it was object oriented, and provided several aids for writing renderers. Like BALSA, it is quite powerful and flexible, but as a result it still requires a large amount of effort to make animations.

2.3 A System For Algorithm Animation

This system [3] [4], developed by Jon Bentley and Brian Kernighan, took a different approach from the others that are described here. They used a text based input language instead of a function library to allow manipulation of the animation with text processing tools, and to achieve programming language independence. It was not very feature rich, but was extremely simple and easy to use. However, as it was developed about 15 years ago, the system is no longer usable in modern environments.

2.4 Various Other Systems

In the nineties, there was a surge of interest in making systems for computational geometry. Most were powerful and supported complex primitives, often including 3D support, but were fairly difficult to use, and overkill for simple animations. Some example systems are XYZ Geobench [9], Geomview [8], and GASP [11].

A number of newer systems intended for general purpose algorithm animation were also developed. These include Tango [10], POLKA[5] and others. They have feature sets of varying complexity, but the majority are implemented as libraries of function calls, and require a fair amount of effort to use.

3 My Approach

My main inspiration for this system was Bentley and Kernighan's System For Algorithm Animation. I decided to aim for the same features vs. ease of use tradeoff—providing only a limited feature set with the goal of having a novice be able to generate useful animations after only a few minutes of using the system.

For several reasons, I chose to use an ASCII input language rather than package my system as a library. First, it means that generating animations only requires putting in appropriate print statements in the algorithm implemenentation, so that any language that has text output facilities can be used. It also allows looking at the animation several times without rerunning the actual algorithm—this can be useful if the algorithm takes a long time to execute. Another reason for text input is that it makes the system more flexible because it is possible to process the animation before it gets viewed. For example, one to write a preprocessor that implements more sophisticated primitives in terms of the existing ones.

My goal with this system was to reexamine the relevant design decisions made in Bentley and Kernighan's system in light of the developments of the last 15 years, and to provide a modern implementation that would support animation of algorithms of up to moderate complexity. Some of the issues involved are the availability of color displays, the huge increases in the available CPU power and memory, and the possibility using the Internet.

4 Implementation

The main pieces that had to be created in this system were the language design, the language parser, and the actual UI implementation. I will now discuss each in turn.

4.1 Language design

The main goal of the language design was to make it easy to learn. Given that goal, I tried to keep everything as simple as possible. I made the language line based—each line is a separate statement. The statements can be divided into 3 general categories—shape definitions, actions on those shapes, and other miscellaneous statements.

Shape definitions describe a particular shape and specify various options such as color, width, fill mode, etc. Supported shapes include points, lines, rectangles, circles, and text.

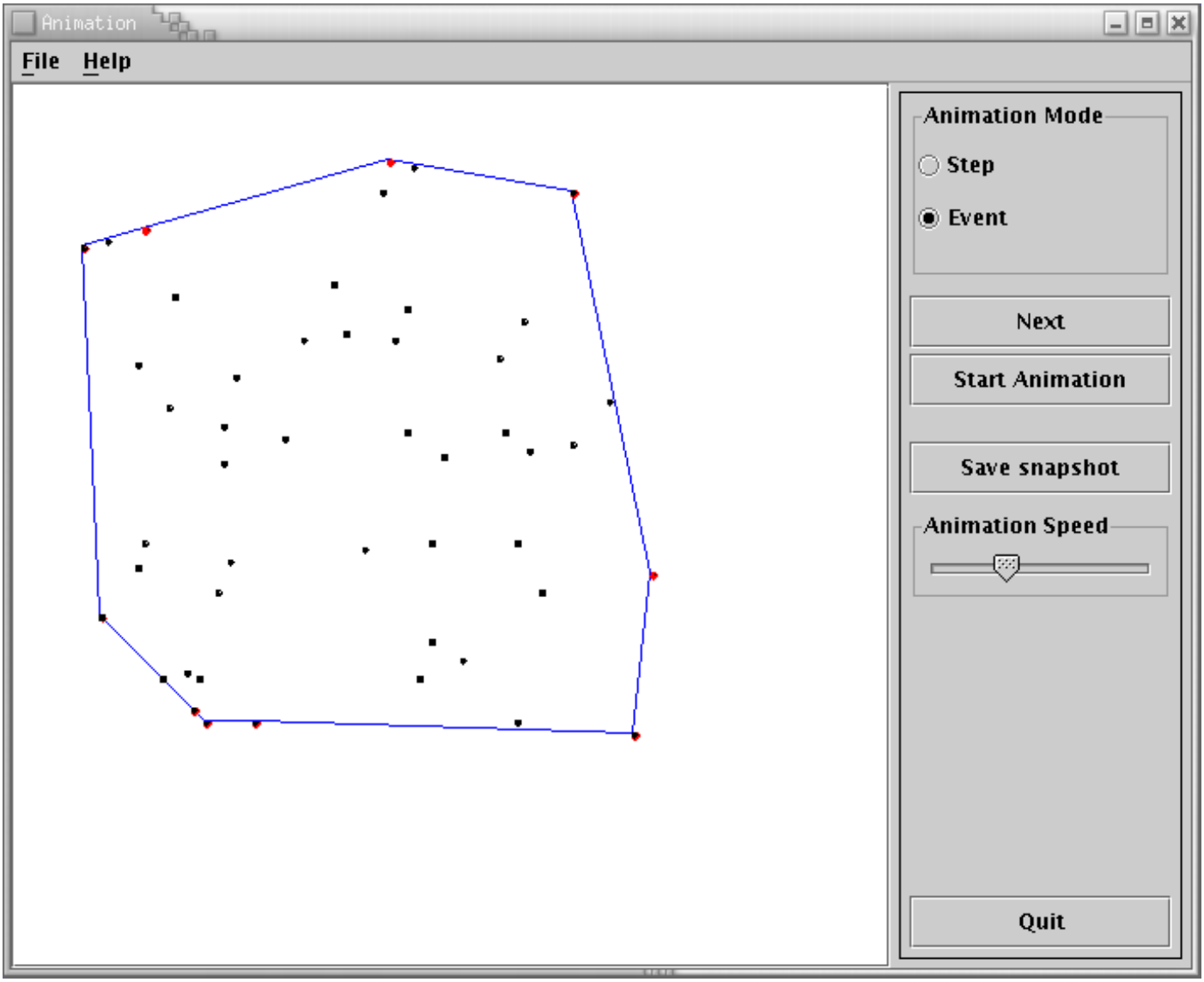

Figure 2: A screen shot of a convex hull animation.

Each shape can also be given a label that can be used to refer to it in later statements. Because the animations are meant to be generated by programs, I used a simple format that is not very easy to read for humans because all the numeric parameters are present with no description. (For example, instead of using `width=2` I just use `2`). This makes it easier to parse, and is just as easy to generate with a program. There are two ways to specify colors—mnemonic names, or an HTML like format: #RRGGBB, where RR, GG, and BB are the hex values of the red, green, and blue components of the color. The mnemonic names are loaded by the preprocessor from a file named colors.txt, which contains mappings of color names to the corresponding colors. The colors.txt file was generated from the rgb.txt file that was on my system, and has over 600 predefined colors.

Action statements include moving shapes, changing their color, and deleting shapes. The general format has the command name, a label corresponding to the shape being manipulated, and then command specific parameters, such as the new color for color_change, the movement offset for move, etc.

There are other statements that change the background color, clear the display entirely, save a snapshot of the current view to an image file, and specify when "events" occur.

The language does not have any intrinsic concept of time—the steps are simply placed in the order they should be performed, and the animation program can process them at any rate.

There were various other features that I considered putting in but decided against in the end. Some of these include control of the order in which overlapping shapes get drawn, sound, and macros and/or custom shape definitions. The main reasons for leaving them out were that they are not really necessary for basic animations, and that they complicate both the language and implementation. Macros can still be added with relative ease as a preprocessing step that would output programs in the current language. Additional primitive shapes can be easily added to the actual program, or as a preprocessing filter that converts them into the available primitives.

Another interesting issue that came up was the choice of coordinate system. Several possible options were to have the user provide a value range, to automatically scale the input values to fit the window, or to use pixel coordinates in the shape definitions. There are no overwhelming arguments for any one of these, but I ended up using pixel coordinates and providing a scrolling view, so that the data does not have to fit in the window. The benefits of this are that the user can very easily tell what changing a particular value will do without having to keep track of the current scale, and that implementation is easier. Automatic scaling, for example, would either require reading through the entire program first to get the extreme values, or would have the picture jump disconcertingly whenever objects outside the current window were added, causing a change in scale.

I will now present some sample input files. The following is the input that generated figure 3:

```
text 30 150 14 black Parser
rect 15 130 80 160
line 80 145 150 145
text 165 150 14 black State
rect 150 130 220 160
line 220 145 285 145
text 300 150 14 black Graphics2D
rect 285 130 400 160
line 343 130 343 90
line 400 145 455 145
text 315 80 14 black Timer
rect 300 60 380 90
line 80 145 150 145
text 470 150 14 black Screen
rect 455 130 530 160
```

This example only illustrates the shape definition commands. An example that illustrates the most other features is:

```
t1: text 50 420 20 black Test Animation!
rect 10 10 400 400 10 gray nofill
r1: rect 25 100 75 300 blue fill
r2: rect 125 100 175 300 blue fill
r3: rect 225 100 275 300 1 blue fill
r4: rect 325 100 375 300 1 #0000FF fill
event
background gray30
c1: circle 50 75 20 20 green fill
c2: circle 150 75 20 20 green fill
c3: circle 250 75 20 20 1 #00FF00 fill
c4: circle 350 75 20 20 1 green fill
event
change_color c1 DarkOrange
change_color c2 #FF7000
change_color c3 DarkOrange
change_color c4 DarkOrange
event
move r1 0 20
move r2 0 20
move r3 0 20
move r4 0 20
```

```
event
snapshot snap.jpg
move r1 0 -20
move r2 0 -20
move r3 0 -20
move r4 0 -20
event
delete r3
delete c3
```

This example illustrates almost all the features of the language, and yet is only about 30 lines long. This is a good indication that the system is indeed easy to learn.

See the attached language reference for the exact syntax of the language.

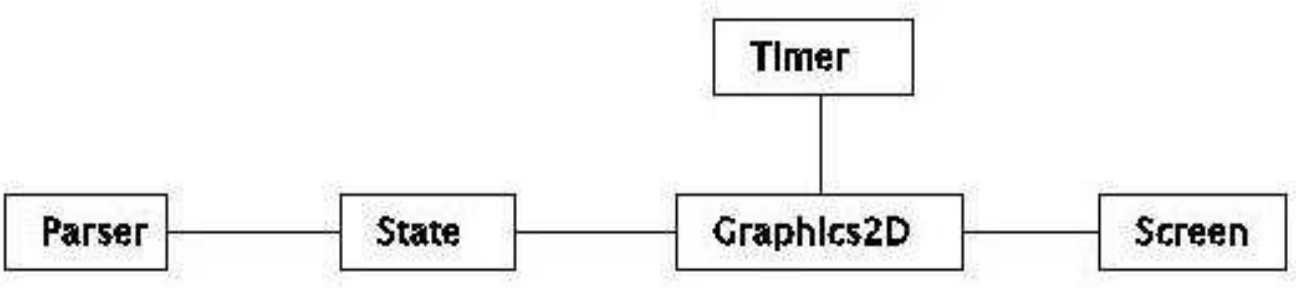

Figure 3: High level structure of the system.

4.2 Parser and User Interface Implementation

For the actual implementation, I decided to use Java for portability reasons and because it has built in UI support. It turned out that the supported primitives mapped extremely well onto the Graphics2D library, so there were very few problems in terms of the actual animation code. I simply create a Collection that contains all the currently active shapes and display them as necessary. In that respect, this entire project can almost be seen as putting a simplifying layer on top of the Graphics2D API. For this reason, adding other features that are supported by the API is fairly trivial in terms of an implementation standpoint, so expansion is certainly possible. However, the problem with Graphics2D is

that it is overkill for simple animations, so I chose to expose only a small part of the API for now.

To perform animation, a timer calls an update function several times per second. This function updates the state of the animation to include the next step or event, and updates the display by drawing all the shapes in an arbitrary order. The speed of the timer can be adjusted by using a slider (See Figure 2 for a screenshot).

The other main part of the implementation is the parser. I initially started out writing this in Java as well, but soon found that while it was certainly possible, it was not very easy, especially since at this point I was still in the process of making frequent changes to the language. To make it easier to experiment with the language design, I introduced an Intermediate Representation (IR) that was much more rigid and thus much easier to parse, and wrote a small (about 300 lines) Perl program to convert the actual language to the IR. This has the problem of hurting the portability of the program somewhat, but since Perl is very portable itself, I did not see it as being too much of a problem. Now that the language design is fairly stable, it is also possible to write the converter as a wrapper around the Java InputStream class. If my system is actually used and the Perl preprocessor becomes a problem, I will implement the Java version. It will also be necessary to change the parser if I were to convert the system into an applet for use on the web. However, since there are is a fairly significant number of other issues that would have to be resolved to do that, it is unlikely that it will be done unless there is significant interest from users. Therefore, I have left it as is for now because it makes it much easier to make adjustments to the language.

5 Results

This section discusses the results of my project. These can be generally partitioned into performance measurements as well as general usefulness conclusions.

5.1 Performance

The following tests were all performed on a 1 Ghz AMD Athlon system, with 256MB of SDRAM running Mandrake Linux 8.0, and using the Java SDK version 1.4.0-beta3 and Perl 5.6.0.

Unlike some applications, this system does not need to be extremely fast because its output is meant to be viewed in real time by humans, so if all the processing is done between frames, no further improvement can be seen. To test performance, I made two input generators: one generates random points, and the other generates random shapes. Each also has an option to add events after every N primitives. These tests are fairly representative of general animations that might be used, because the locations of the primitives do not affect the performance in any way, so randomized positioning should have similar performance to an animation generated by an algorithm.

Number of Points	Event Spacing				
	10	50	100	500	1000
100	1	1	1	1	1
1000	8	2	1	1	1
2000	20	5	3	2	1
10000	367	68	39	10	6

Table 1: Random points: table entries are time(s) to run the animation.

Number of Shapes	Event Spacing		
	10	50	100
100	1	1	1
1000	30	7	4
2000	110	23	12

Table 2: Random shapes: table entries are time(s) to run the animation.

The data I got is summarized in Tables 1 and 2. The system was set to event mode and the nominal frame rate was set to 15. The frame rate is nominal, not actual, because as the number of objects increases, the time to display each frame starts to exceed the nominal per-frame time, and the animation slows down. This effect is clearly seen in the large increase in time from the 2000 point to the 10000 point test, and from the 1000 to the 2000 shape test. The effect is also clearly visible when viewing the animation—as the number of objects to be displayed grows to beyond a thousand or so, the animation noticibly slows down.

Looking at the data in the other direction—increasing the distance between events rather than the number of points, we can conclude that the time to actually read in the data and update the internal data structures is negligible compared to the drawing time. While it is not shown in the above data, this is also true of the action commands such as color_change, move, etc. When the number of objects is small, everything finishes in less then the per-frame time, and as the number of objects increases, the drawing time dominates all other factors.

Another thing to notice is that displaying random shapes is slower then displaying points by a factor of 3 or so. This is not surprising, and should not cause problems because since general shapes take up more space in the view, which makes it unlikely that there will be as many displayed at a time.

Unfortunately, I have not been able to think of any way to seriously improve the performance without major changes. Fortunately, at least on the test machine, the performance

is already quite sufficient for the types of animations for which the system was intended, as visualizing algorithms that operate on many more then 1000 shapes at once will likely be quite difficult given the minimal feature set.

5.2 Usability

Given adequate performance, it remains to examine whether the system achieves the original goals of being useful and easy to use. This is much harder to measure quantitavely, but I will attempt to argue convincingly that it has.

First, ease of use. The language currently has only 12 statements: 5 primitive definitions, 5 action commands, and 2 miscellaneous commands. The syntax for all of them is very simple, so it seems fair to say that even a complete novice to programming could learn to use the system in less then an hour.

Next, usefulness. This is quite hard to measure, but I believe that a reasonable way is to examine some possible algorithms and see whether they can be animated naturally using the system. Here is an incomplete list of algorithms and/or categories of algorithms that I belive can be easily animated:

- Sorting Algorithms
- Bin Packing
- Memory Allocation
- Hash Tables
- Simple Graph Algorithms = MST, shortest path, etc.
- Convex Hull
- Voronoi Diagrams
- Text Processing Algorithms (substring, etc.)
- Any other algorithm that does not require complex primitives or large data sets

Algorithms that this system is not well suited for include:

- 3D algorithms
- Anything that requires complex curves
- Any algorithm (even one of the above), when trying to visualize large amounts of data

Note that actually animating some of these algorithms still requires a lot of algorithm specific work to decide exactly how to visualize and lay out the appropriate components. There is not really much that a general purpose system can do to help that, and it was not one of my goals to try. Having said that, the text based nature of the input language allows others to write preprocessors to make a specific task easier. An example that one potential user mentioned was a tool to automatically layout graphs. While this is certainly not something that my system does, if someone wrote a program to do the layout, and output a description of the graph as a sequence of point, line, and text statements, my system could be used without any modification to display it.

5.3 Possible Future Work

The system as it stands is functional and useful, but there are some more features that could be useful to have. I have not implemented them in the first version because none of them are essensial for basic functionality, they would make the language definition more complex, and in some cases would also make implementation harder. That being said, here are some possibilities for future enhancements:

- Allow different line, fill styles for the existing primitives.
- Implement more primitives, such as polygons, bezier curves, etc.
- Add the capability to do smooth transformations.
- Add more transforms—rotate, scale, shear, morph, etc.
- Allow the user to have several concurrent views, and improve the UI to naturally support them.
- Rewrite the parser in Java.
- Make an applet and make it available online.

I plan to implement some of these in the near future because I want them for my own use. The rest may be added if there are requests from other users.

6 Conclusion

This paper presented Yet Another System For Algorithm Animation. The main features of the system are extreme simplicity and ease of use, and text based input. The Java based inplementation was described and analyzed, with the conclusion that the system's feature set is flexible enough to animate a fairly wide variety of algorithms, and that the performance is good enough that it causes no problems for moderately small data sets.

Acknowledgments

I would like to thank Brian Kernighan for agreeing to advise my project, and for providing many helpful comments and suggestions throughout the semester. I would also like to thank Kaijen Hsiao for her help with revising this paper.

References

[1] A. Hausner, D. Dobkin, "Making Geometry Visible: An Introduction to the Animation of Geometric Algorithms", http://www.cs.princeton.edu/ ah/publications/animation/index.html

[2] R.M. Baecker, "Sorting out Sorting" (video), in Siggraph Video Review 7, 1981.

[3] J.L. Bentley and B.W. Kernighan, "A System for Algorithm Animation", Computing Systems (Winter 1991) 4(1) pp. 5-31.

[4] J.L. Bentley and B.W. Kernighan, "A System for Algorithm Animation: Tutorial and User Manual", Technical Report 132, AT&T Bell Labs, 1991.

[5] http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html

[6] M.H. Brown and R. Sedgewick, "A System for Algorithm Animation", ACM Computer Graphics (July 1984) 18(3) pp. 177-186.

[7] M.H. Brown, "Zeus: A System for Algorithm Animation and Multi-View Editing", Proceedings, 1991 IEEE Workshop on Visual Languages pp 4-9.

[8] http://www.geom.umn.edu/software/download/geomview.html

[9] P. Schorn, "The XYZ GeoBench for the Experimental Evaluation of Geometric Algorithms", in Computational Support for Discrete Mathematics: DIMACS Workshop (March 1992), pp. 137-152.

[10] J.T. Stasko, "Tango: A Framework and System for Algorithm Animation", IEEE Computer (September 1990), pp. 27-39.

[11] A.Y. Tal and D.P. Dobkin, "Visualization of Geometric Algorithms", IEEE Transactions on Visualization and Computer Graphics (TVCG) Volume 1, Number 2.