A System for Algorithm Animation†

Jon L. Bentley and
Brian W. Kernighan
AT&T Bell Laboratories

ABSTRACT: An algorithm or a program can be animated by a movie that graphically represents its dynamic execution. A sorting algorithm, for instance, might be animated by a sequence of frames that shows a set of vertical lines of various heights being permuted into order of increasing height. Such animations are useful for developing new programs, for debugging, and for explaining how programs work. This paper describes ANIM, a basic system for algorithm animation. The output is crude, but ANIM is easy to use; a novice user can animate a program in an hour or two. ANIM currently produces movies with the X window system, among others; it also renders movies into "stills" that can be included in TROFF or TEX documents.

1. Introduction

Dynamic displays give more insight into the behavior of dynamic systems than static displays do. Consider, for instance, the experi-

† A preliminary version of this paper appeared in Bentley & Kernighan [1987A].

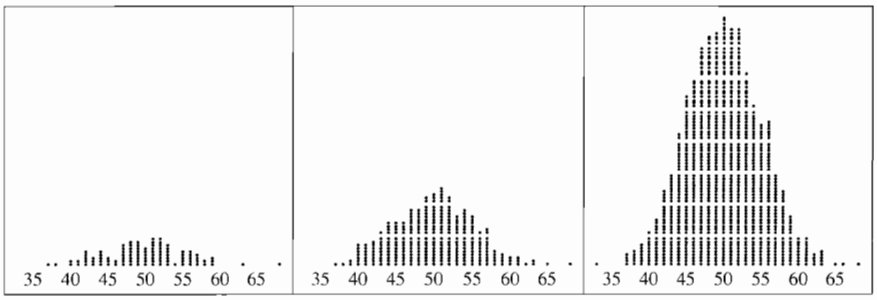

Figure 1: Three frames of a movie showing the distribution of the number of heads in 100 tosses.

ment of tossing a coin 100 times. The expected number of heads is 50, but the actual number obeys a binomial distribution. Probability theory tells us that the binomial histogram of counts will converge to the bell-shaped normal distribution. The sequence of pictures in Figure 1 is part of a movie that helps us appreciate the process more intuitively. The three snapshots were taken after 100, 300, and 1000 experiments. Every tenth vertical dot has been deleted to facilitate counting.

This paper describes the animation system that produced those pictures. A short program (a dozen lines of `awk`) performed the experiments and wrote the results to a *script* file describing the histogram's evolution through time. That file was processed by a program named `stills` to produce the pictures above, using `pic` and `troff`; we specified which frames to display and in what size and form. A `movie` program displays the same data on several kinds of terminals; the viewer can proceed forward or backward at full speed or a step at a time, and change the screen layout to emphasize certain views. The components of ANIM are shown in Figure 2.

Several systems have been developed for algorithm animation; see, for instance, Brown & Sedgewick [1985] and Stasko [1990] and the references therein. Most of those systems produce animations of very high quality; unfortunately, they are also expensive in programmer time and are sufficiently complicated that they are not suited for casual use. ANIM is at the opposite end of the spectrum: its output is primitive, but it is easy to use; a new user can animate a simple program in an hour or two by adding a few lines of code. Although

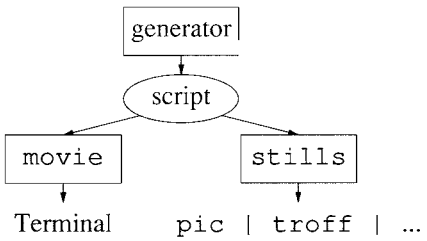

Figure 2: Components of the ANIM animation system.

ANIM was designed primarily with algorithm animation in mind, it can be useful in other domains as well.

Bentley & Kernighan [1987B] is a user's manual for ANIM; this paper omits many particulars found there. Section 2 of this paper presents the details on the animation of one algorithm. Section 3 describes ANIM, and Section 4 surveys its use. Conclusions are offered in Section 5.

2. *An Example — Bin Packing*

The first part of this section uses animations to tell the story of an algorithm; the second part then describes how we used ANIM to produce the animations.

The Algorithm.

Bin packing is a classical problem in computer science. The input is a set of weights between zero and one; we are to assign the weights to a minimal number of bins under the constraint that the sum of the weights in any bin is at most one. This problem arises in applications such as stock cutting and placing a set of files onto several floppy disks. Because the problem is NP-complete, researchers have investigated heuristics that give good, but not necessarily optimal, packings.

We will study the "First Fit Decreasing" or "FFD" heuristic, which is described in many textbooks. "Decreasing" means that the weights are considered from largest to smallest, and "First Fit" means that each weight is placed in the first (leftmost) bin into which it fits. Figure 3 shows three frames from a movie of an FFD packing

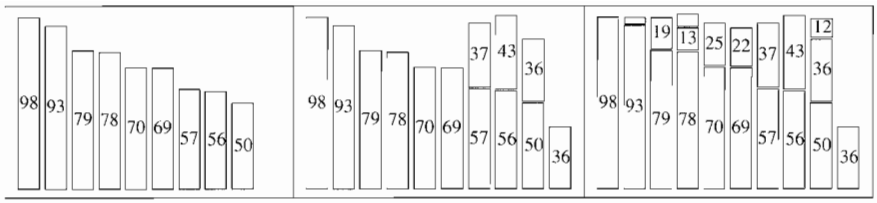

Figure 3: Three frames of an FFD packing of 20 weights.

of 20 weights chosen uniformly from [0,1]. The numbers in each rectangle are the weights multiplied by 100 and rounded; the snapshots are taken after inserting 9, 13, and 20 weights.

There are, of course, many ways to draw pictures of bin packings. Figure 4 shows two side-by-side views of packing 40 random weights; the top snapshot shows the packing after 26 weights have been inserted, and the bottom snapshot is the final state. The left view is the same representation as in Figure 3, in which weights are rendered as rectangles (with numbers included if there is room). It is fine for small instances, but cluttered for large packings. The right view places a dot at the top of each weight; it is an effective way to depict large packings.

The FFD heuristic produces very good packings when the weights are drawn uniformly over the range [0,1]. Figure 5 shows 500 weights being packed, after 375 and 500 insertions. The FFD heuristic essentially ''folds'' the weight list over on itself. There are a few large holes here and there, but, on the whole, the heuristic is quite effective.

When the weights are chosen uniformly from [0,1/2], the FFD packings are even more efficient, and are optimal more than 80% of the time. Figure 6 shows the packing of 500 weights uniform on that range. In the first snapshot, only weights greater than 1/3 have been inserted. The first and second weights go into bin 1, the third and fourth into bin 2, etc. The second snapshot shows the weights between 1/3 and 1/4: roughly a quarter of them ''backfill'' the old gap, while the remainder create a new ''sawtooth'' that will be backfilled by later weights. The final snapshot shows that the resulting packing has a great deal of structure.

We have found pictures like these useful in several contexts:

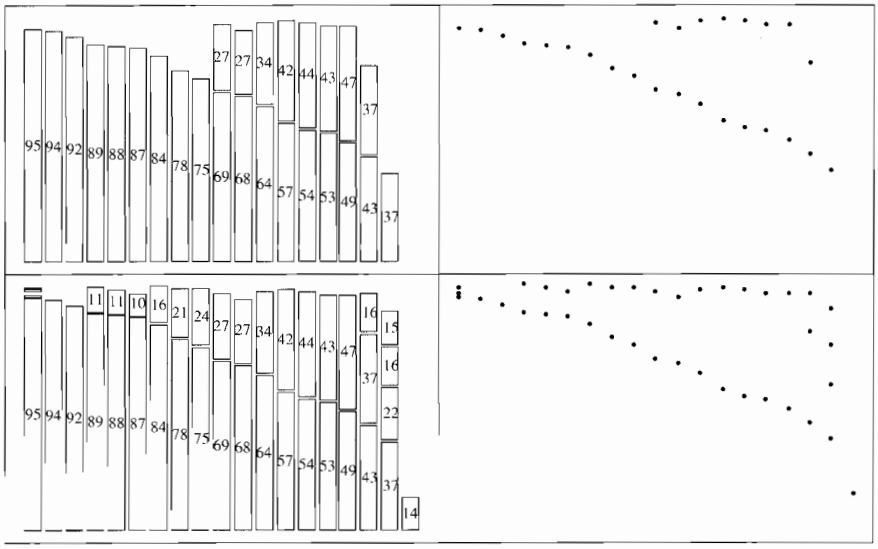

Figure 4: Two frames of two views of FFD packings.

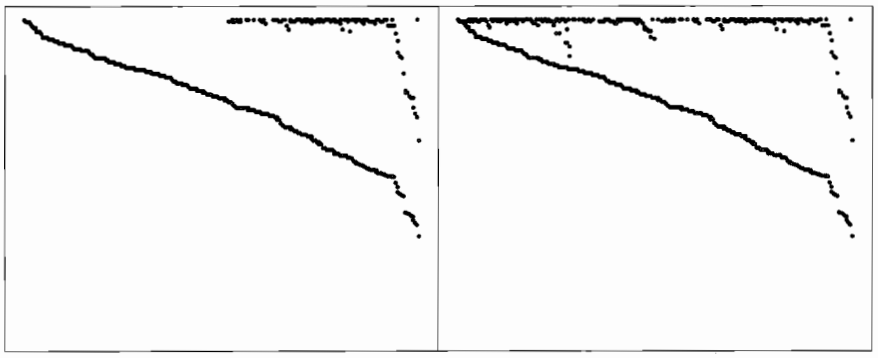

Figure 5: An FFD packing of 500 weights uniform on [0,1].

Teaching. Movies are effective classroom tools, whether stored on videotape or controlled in real time by the instructor. Stills give less insight, but they allow longer contemplation and discussion, and are much more portable.

Programming. A simple bin packing program is very short and easy to get right; we'll see one shortly. Fast FFD programs, though, require a few hundred lines of subtle code; pictures make debugging such programs fairly easy.

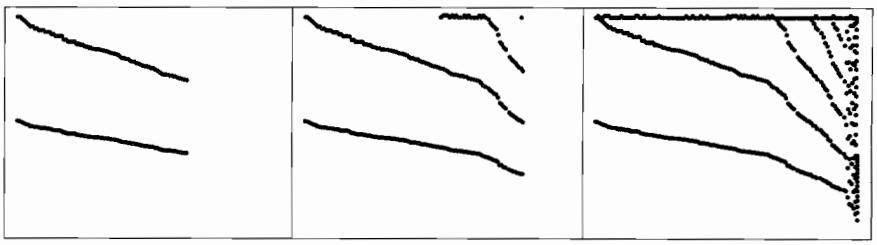

Figure 6: 500 weights uniform on [0,1/2].

Research. Our interest in algorithm animation can be traced to the summer of 1983, when one of the authors worked on the analysis of the FFD heuristic [Bentley, *et al.* 1983]. We spent roughly a week writing a program to produce bin packing animations, and it was a wise investment: the pictures led us to several surprising conjectures and proofs.

The Animation.

The first step in using ANIM is to obtain a program to animate. Program 1 is an `awk`† program that writes a history of a bin packing algorithm. We use `awk` for all examples in this paper because it is succinct and widely available, but any programming language may be used to prepare input files. All the action takes place within the `BEGIN` block (C programmers may think of it as `main()`). The first line sets from the command line the values of `n` (the number of weights) and `u` (the weights are distributed over the range [0, `u`]); the second line allows a new seed for the random number generator to be provided from the command line. The `for` loop packs each of the `n` weights; the first two lines in the loop body ensure that the weights are uniform and appear in decreasing order. The inner `for` does a sequential search for the first bin into which a weight fits; the next two statements insert the weight and write a record of the insertion.

To animate the program we replace the single print statement by several; the result is Program 2, which is named `ffd.gen`. To animate a packing of four weights chosen uniformly over the range [0,1] we invoke the program with this command:

† This program and `awk` programs later in this paper use features of the 1985 version of `awk` described in Aho, *et al.* [1988].

```
BEGIN {
    n = ARGV[1]; u = ARGV[2]; curmax = 1
    if (ARGC > 3) srand(ARGV[3])
    for (i = 1; i <= n; i++) {
        curmax *= exp(log(rand())/(n+1-i))
        tw = u * curmax
        for (b = 1; bin[b] > 1-tw; b++)
            ;
        bin[b] = neww = (oldw=0+bin[b]) + tw
        print "insert weight" , tw, "into bin",
            b, "from", oldw, "to", neww
    }
}
```

Program 1: An FFD program.

```
awk -f ffd.gen 4 1
```

That produces as output this ''script file'', printed in two columns to save space:

```
#ffd_bin_packing n=4 u=1
view dots
text 1 0
view dots
text 1 0.968228 dot
view rect
box 0.6 0.01 1.4 0.968228
text small 1 0.484114 97
click weight
view dots
text 2 0.388697 dot
view rect
box 1.6 0.01 2.4 0.388697
text small 2 0.194348 39
click weight
view dots
text 2 0.733216 dot
view rect
box 1.6 0.398697 2.4 0.733216
text small 2 0.560956 34
click weight
view dots
text 3 0.307457 dot
view rect
box 2.6 0.01 3.4 0.307457
text small 3 0.153728 31
click weight
```

This script file uses four commands: `box`, `text`, `view` and `click`. A rectangle with opposing corners at (x_1, y_1) and (x_2, y_2) is drawn by a command of the form

optional label: `box` x_1 y_1 x_2 y_2

(Literals are shown in `typewriter` font and categories are in *italics*.) Text is produced at (x,y) by a command of the form

optional label: `text` *optional size x y anything at all*

Coordinates can lie in any range; later programs will scale them appropriately.

The `view` command is used to place output in a particular view. There are two views here, `dots` and `rect`. Interesting events are marked by the `click` command. `stills` and `movie` can refer to each click with this mechanism. Labels, view names, and click names are arbitrary and unrelated to one another.

```
BEGIN {
    n = ARGV[1]; u = ARGV[2]; curmax = 1
    if (ARGC > 3) srand(ARGV[3])
    print "#ffd_bin_packing n=" n " u=" u
    print "view dots\ntext 1 0"
    for (i = 1; i <= n; i++) {
        curmax *= exp(log(rand())/(n+1-i))
        tw = u * curmax
        for (b = 1; bin[b] > 1-tw; b++)
            ;
        bin[b] = neww = (oldw=0+bin[b]) + tw
        print "view dots\ntext", b, neww, "dot"
        print "view rect\nbox", b-0.4,
                oldw+.01, b+0.4, neww
        if (tw > .1)
            print "text small", b, oldw+tw/2,
                    int(100*tw+.5)
        print "click weight"
    }
}
```

Program 2: An FFD animation program.

This script file can now be interpreted by `movie`, to see the dynamic behavior, or by `stills`, to select frames for printing.

The `movie` program is currently implemented on several different kinds of terminals, but the user interface is much the same on all. `Movie` reads the script file once (typically from disk) and loads it into local memory; during this process, the movie is played once from beginning to end. Subsequently, the viewer can examine it in greater detail with pop-up menus controlled by a mouse.

Mouse button 1 is used for ''stop'' and ''go''. Button 3 does most of the work. Selecting ''again'' repeats the movie. The speed is controlled by either doubling or halving the pause time at certain key events (the ''clicks'' mentioned above). This applies only in ''run'' mode; if one selects ''1-step'' mode with button 3, then each hit of button 1 moves to the next appropriate click. ''Backward'' and ''forward'' change the direction of play; together with ''1-step'', they make it easy to locate a key event in the movie.

Button 2 lists the names of the views and clicks in the animation. When a view name is selected, one can sweep a rectangle in which that view is to be displayed; one can delete a view by sweeping its rectangle out of the movie window. Selecting a click name turns it on or off (the ones that are on have an asterisk next to the name). Clicks that are on cause a pause in run mode and a wait in 1-step

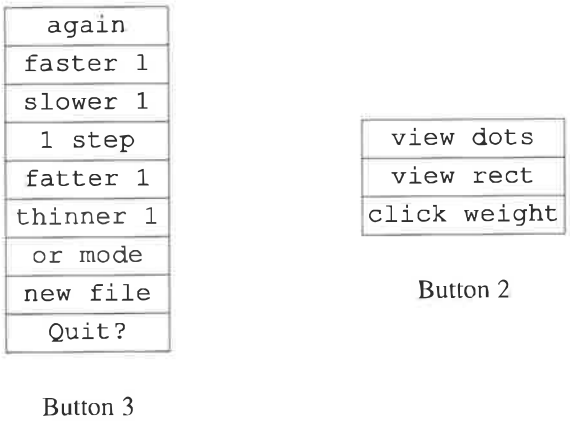

Figure 7: Menus on buttons 2 and 3.

mode. Figure 7 shows the button 3 menu and the menu on button 2 for the animation of Program 2.

Perspective.

Using ANIM is rather like using a home camcorder and VCR. The computation cannot be interactive (e.g., you cannot type in a number and watch a binary search try to locate it in an array). Once the script has been generated, there's no way to change it except to generate it again. The *display* of a fixed computation is, however, highly interactive: the viewer can run it forward or backward, quickly or slowly or a frame at a time, etc.

ANIM would have been very useful for the experimental bin packing research we sketched earlier. Several years ago, we had to build a special-purpose animation program for the Teletype 5620 terminal; it took a week and several hundred lines of C. We can now do the job in a dozen lines of code in an hour. ANIM also provides several facilities that were not present in the original program but which would have been very useful: multiple views, stills output, and more control over presentation.

The case study in this section illustrates the capabilities and limitations of ANIM. The output of `movie` is a crude but useful animation. The output of `stills` is handy for more detailed study and for presentation in documents, like this paper.

If ANIM is so crude, why bother using it? Why not animate an algorithm simply by drawing geometric objects on the output device you happen to be using? Some of the answer lies in services like these:

Device Independence. A script file can be viewed interactively as a movie on several different kinds of terminals; a higher-quality videotape can be made on some terminals. The same script file can be incorporated into a document by `stills`.

Names. Labels allow geometric objects to be erased; implicit erasure by re-using a label is easy to use and to implement. Click names mark key events and can be used to group related events.

Independent Views. Different simultaneous views of a process are crucial for animating algorithms. In ANIM, a single statement moves from one view to another. Within a view, the user need not be concerned about the range of coordinates; the system scales automatically.

Viewer Control. Both `movie` and `stills` allow the viewer to select which views will be displayed and which clicks will be recognized. Additionally, `movie` allows the viewer to go forward or backward, in single steps or running at a selected speed.

An Interface To The World. Although writing to files takes more computer time than using the geometric primitives provided by a specific output device, those files allow complicated tasks to be easily composed out of simple software tools.

3. The System

We begin this section with the obligatory minimal movie:

```
echo text 0 0 hello, world | movie
```

This `awk` program makes a movie with real motion:

```
BEGIN { s = "hello, world"
        for (i = 1; i <= length(s); i++)
            print "text", i, 0, substr(s, i, 1)
      }
```

Having dispensed with these formalities, we turn to the more systematic view of ANIM shown in Figure 8; additional details can be found in Bentley & Kernighan [1987B]. A script file is processed by

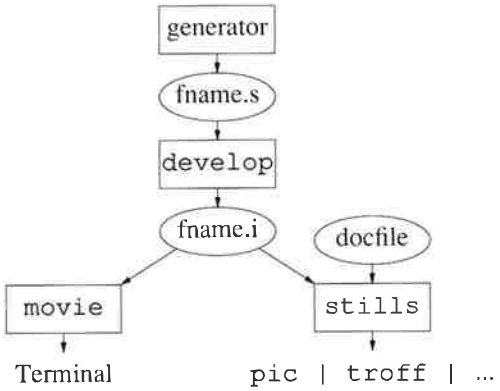

Figure 8: Components of ANIM.

the heretofore unmentioned program named `develop`. The output of `develop` is an intermediate file that feeds `stills` and `movie`.

The Script and Intermediate Languages.

The script language is summarized in this table:

optional	*label*: `line`	*options*	x_1 y_1 x_2 y_2	
optional	*label*: `text`	*options*	*x y*	*string*
optional	*label*: `box`	*options*	*xmin ymin*	*xmax ymax*
optional	*label*: `circle`	*options*	*x y*	*radius*
`view`	*name*			
`click`	*optional*	*name*		
`erase`	*label*			
`clear`				
`#`	*comment: any text*			

A line whose first non-blank character is # is a comment; blank lines are ignored.

Labels are optional. If a label is present on a geometric object, it names the object and implicitly erases any existing object with the same name in the same view.

Each object type has a small set of valid options. Lines may have arrowheads on either or both ends, and may be of several styles and thicknesses. Text may be centered or left- or right-justified, in one of several sizes. Boxes and circles may be filled. The options are a (possibly null) list of names, terminated by the next numeric field. For instance, a script file might contain the command

```
A117: line <-> fat 0 234.021 1 234.087
```

to draw a heavy line with arrows at both ends.

The `view` statement places subsequent objects in the named view, and `click` denotes an interesting event.

A labeled geometric object can be explicitly erased by the command

`erase` *label*

The various views have distinct name spaces; the same label may be applied to two unrelated objects in two different views. The `clear` statement erases all objects in the current view.

The intermediate language can be viewed as the ''assembly code'' output of the `develop` program. The program scales all numeric values into the range 0..9999, translates symbolic labels into numbers, makes implicit erasures explicit, and translates options into a standard form. The resulting file is easy for the subsequent `movie` and `stills` to process; more details are in Bentley & Kernighan [1987B]. The `develop` program began life as a 150-line `awk` program, but is now about 1000 lines of C.

The Movie Programs.

The original `movie` program runs on the Teletype 5620 and contains roughly 1500 lines of C. Movie production, as with most 5620 programs, uses a host process and a terminal process. The host sends the intermediate file produced by `develop` in a compact form to the terminal, which stores it in a form suited for forward or backward display. Considerable effort was expended in making the internal representation compact, since memory was at a premium in the 5620. Fortunately, newer terminals and workstations have much more memory, so this is no longer an issue.

The X version of ANIM has about 1100 lines of C in a single process for animation, but requires 1800 further lines to convert the simple graphics and mouse interfaces of our local window system into X calls.

We also implemented a version of `movie` on the SGI IRIS workstation, for producing videotapes suitable for classroom use. In some ways it is less powerful: it runs only in the forward direction and does not have single stepping. In other ways it is more powerful: the

viewer has greater control over the positioning of views and the time spent pausing at clicks, and we have added colors as options on any geometric object. In any case, the programs are different: the original `movie` is controlled by a mouse, while the IRIS version has textual input. This version took only a few days; it is about 500 lines long.

The Stills Program.

The `stills` program is a typical `troff` preprocessor. Portions of its input bracketed by `.begin stills` and `.end` are translated into `pic` commands, and the rest of the input is passed through untouched. A paper containing `stills` input is typically compiled by a command like

```
stills paper | pic | troff >paper.out
```

For instance Figure 4 was produced by this description:

```
.begin stills
file     ffd2.s
view     rect ""
view     dots  ""
print    weight 26 40
frameht  1.4
framewid 2.25
down
times    invis
small    -6
.end
```

The first line names the script file, and the next two lines select views for display and give them null titles. The `print` statement causes snapshots at the selected times of the click `weight`. The five remaining lines are name-value pairs: the height and width are in inches, `down` causes time to go down the page, and `small` text is rendered six points smaller than usual.

In summary, `stills` input consists of these commands:

```
print all
print final
print clickname all
print clickname number number number ...
view  name  optional title
parameter-name  value
```

At least one `print` statement and a `file` assignment are mandatory; other statements are optional.

4. *Uses of The System*

This section describes several animations produced by ANIM, and some supporting tools.

Sorting.

Sorting algorithms provide one of the most fertile domains for algorithm animation. Indeed, Ronald Baecker's movie "Sorting out Sorting" [1981] has provided for many students their first (and frequently best) exposure to algorithm animation. One of the authors recently faced the problem of giving a 50-minute undergraduate lecture about sorting. There wasn't time for the 25-minute "Sorting out Sorting" (and the algorithms covered didn't quite match the syllabus), so we used ANIM to produce a simpler and shorter substitute.

Figure 9 shows four frames from an animation of selection sort on a 15-element array. The vertical lines represent the elements to be sorted; in the initial frame they are in a random order, and in the final frame they are sorted in increasing order. Each comparison between a pair of elements is denoted by a horizontal line below the pair. Selection sort works by first selecting the smallest element and placing it in the first position of the array, then selecting the smallest remaining element for the second position, etc.

Figure 9 was produced by Program 3. The `BEGIN` block generates, draws, then sorts the array elements. Function `randint` generates random integers from a specified range; it is used to create the

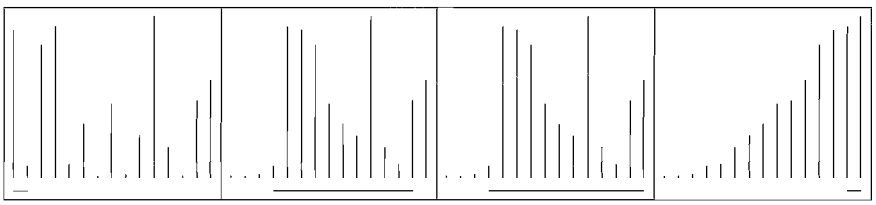

Figure 9: Selection sorting a 15-element array.

```
BEGIN {
    n = 15
    for (i = 1; i <= n; i++) {
        x[i] = randint(1, 100)
        draw(i)
    }
    selectsort()
}
function randint(l, u) {
    return l + int((u-l+1)*rand())
}
function draw(i)  {
        print "a" i ": line", i, 0, i, x[i]
}
function swap(i, j) {
        t = x[i];  x[i] = x[j];  x[j] = t
        draw(i); draw(j)
        print "click swaps"
}
function less(i, j) {
        print "compline: line", i, -8, j, -8
        print "click comps"
        if (x[i] < x[j]) return 1; else return 0;
}
function selectsort(   i, j) {  # Sedgewick, p.96
    for (i = 1; i <= n-1; i++)
        for (j = i+1; j <= n; j++)
            if (less(j, i)) swap(i, j)
}
```

Program 3: A simple program for animating sorting.

elements (and is also used by Quicksort). Function `draw` is the main animation primitive. Functions `swap` and `less` are the two fundamental operations of all later sorting algorithms; they are augmented to produce animations as a by-product. Function `selectsort` implements selection sort, using `less` and `swap` as primitives.

Program 4 contains four additional sorting algorithms. The page numbers in the comments are from Sedgewick [1988]. The heap sort algorithm has been deleted from this program to conserve space (18 lines). We modify Program 3 to animate the various algorithms by changing the final line in the `BEGIN` block.

The complete movie for the classroom lecture animated five sorting algorithms in 74 lines of `awk`. Starting from a program like Program 3, the task required two hours to write the sorts, one hour to experiment with representations as seen through a home camcorder, and one hour to shoot the five-minute movie onto videotape.

```
function insertsort(   i, j) {  # Sedgewick, p.98
    for (i = 2; i <= n; i++)
        for (j = i; j > 1 && less(j, j-1); j--)
            swap(j-1, j)
}
function bubblesort(   i, j) {  # Sedgewick, p.100
    for (i = n; i >= 1; i--)
        for (j = 2; j <= i; j++)
            if (less(j, j-1)) swap(j, j-1)
}
function shellsort(   i, j, h) {  # Sedgewick, p.108
    for (h = 1; h <= n; h = 3*h + 1) ;
    for (h = int(h/3); h >= 1; h = int(h/3))
        for (i = h+1; i <= n; i++)
            for (j = i; j > h && less(j, j-h); j -= h)
                swap(j-h, j)
}
function quicksort(l, u,   i, m) {  # Sedgewick, p.115
        if (l >= u) return
        m = l
        swap(m, randint(l, u))
        for (i = l+1; i <= u; i++)
                if (less(i, l)) swap(++m, i)
        swap(l, m)
        quicksort(l, m-1)
        quicksort(m+1, u)
}
```

Program 4: Additional sorting algorithms.

As entertaining as it is to watch a fast runner, the real glory of track meets is a race among many runners. Bentley & Kernighan [1987B] contains a merge program that allows us to splice together runs of the various sorting algorithms into a race. While some algorithm animation systems implement races with a general mechanism for time sharing, we do the job with a dozen-line `awk` program.

ANIM is also able to produce more sophisticated sorting animations. Figure 10 shows three frames of the history of Quicksort on a 50-element array. In the top view, the dots represent the elements to be sorted (x is position in the array and y is value), the horizontal lines represent a recursive call of Quicksort (width is the subarray and height is the partitioning value), and vertical lines show two pointers used by the partitioning code. The horizontal lines in the bottom view give the history of the recursive calls, so the final view represents the call tree of the function. This animation was produced by a 55-line `awk` program.

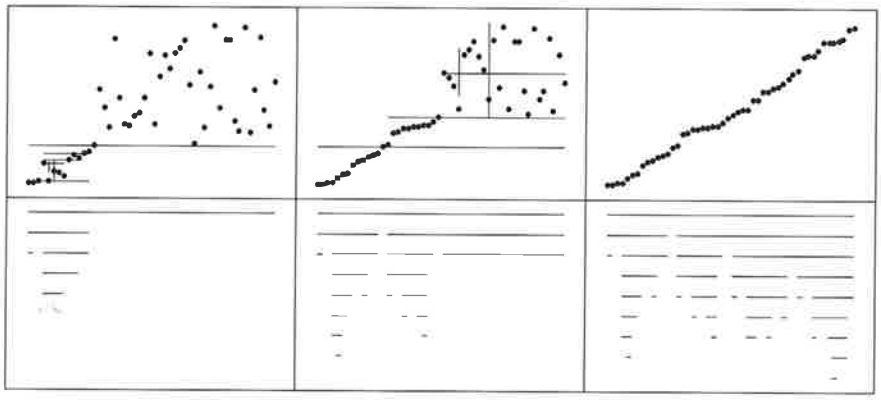

Figure 10: Quicksorting a 50-element array.

Three Dimensions.

Many processes to be animated naturally take place in three dimensions. In this section we will sketch a simple preprocessor that allows 3-d scripts to be viewed through stereo viewers. Figure 11, for instance, shows a minimum spanning tree of 40 points distributed uniformly over the unit cube; some readers will be able to view it by crossing their eyes. The program we'll describe translates a 3-d script into a standard script that contains two 2-d views (for left and right eyes).

Figure 11 was produced by Program 5. That simple version of a stereo program handles five kinds of input lines: `lines` and `text` are now in three dimensions, while `view`, `click` and comments are

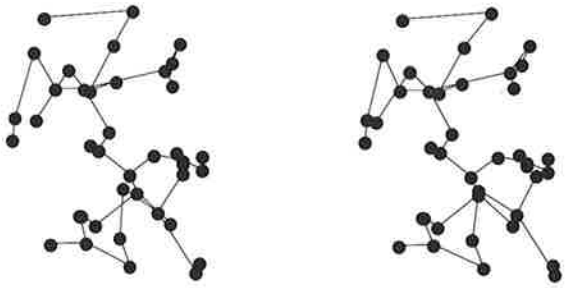

Figure 11: A 3-d minimum spanning tree.

```
BEGIN {
    lpicx = 2;   rpicx = 0
    leyex = .44; reyex = 1-leyex
    gs = 3.8
    planez = -.5
    eyey = .5;   eyez = -1
    OFS = "\t"
  }
$1 == "line" {
    ax = $2; ay  = $3; az = $4
    bx = $5; bby = $6; bz = $7
    sfa = gs * (planez-eyez) / (az-eyez)
    sfb = gs * (planez-eyez) / (bz-eyez)
    print "line", lpicx + sfa*(ax-leyex), sfa*(ay-eyey),
                  lpicx + sfb*(bx-leyex), sfb*(bby-eyey)
    print "line", rpicx + sfa*(ax-reyex), sfa*(ay-eyey),
                  rpicx + sfb*(bx-reyex), sfb*(bby-eyey)
    print "click stereo"
    next
  }
$1 == "text"  {
    tx = $2; ty = $3; tz = $4
    sf = gs * (planez-eyez) / (tz-eyez)
    print "text", lpicx + sf*(tx-leyex), sf*(ty-eyey), $5
    print "text", rpicx + sf*(tx-reyex), sf*(ty-eyey), $5
    print "click stereo"
    next
  }
  { print }
```

Program 5: A simple stereo program.

unchanged. The transformation for mapping a 3-d point into two 2-d views assumes that all input is in the unit cube. Note that the two images for left and right eyes are implemented as a single view in the resulting script file.

Program 5 is for educational purposes only. The complete stereo program is implemented in 150 lines of `awk`. It supports a more complete 3-d script language: `lines` and `text` may have labels (and subsequently be `erased`), a `frame` statement draws the 3-d bounding box of the region, and `view`, `click`, `clear`, and comments are supported as well. The larger program provides better error checking, and no longer assumes that the input is contained in the unit cube (the first pass of the now two-pass program scales the input). A command-line option allows the stereograms to be viewed either by crossing one's eyes or by using a stereo viewer.

Our first application of 3-d stereo movies was frivolous: we watched equal-mass bodies moving through 3-space under

Newtonian attraction (Bentley & Kernighan [1987B] presents a 2-d version). Our first serious application was for a biophysicist colleague who was studying the structure of a molecule with a few hundred atoms. The molecular graphics systems available to her did not support the operations she desired, so we made our own versions with a few simple programs. For instance, a 35-line `awk` program rotated its input by an angle given on the command line, and an 8-line shell script called the rotation program to spin the molecule. We have also used the complete stereo program to debug 3-d geometric algorithms.

A Survey of Applications.

ANIM provides only a few geometric primitives: lines, boxes, circles and text. Nevertheless, they appear to be sufficient for making a variety of interesting movies.

Set algorithms provide an interesting domain for algorithm animation; we saw several sorting algorithms earlier. Bentley & Kernighan [1987B] contains animations of binary search trees and heaps, along with hints on how to lay out trees.

Figure 12 shows a randomly generated parse tree. It was produced by Program 6, which reads a grammar with productions including these:

```
Sentence -> Nounphrase Verbphrase
Verbphrase -> Verb Modlist Adverb
Modlist -> very Modlist
```

Program 6 is a slightly modified version of a program in Section 5.1 of Aho, *et al.* [1988]. The animation represents each node in the tree by a bullet, left-justified text, and a line to its parent. As with more general graph algorithms, the hard part of drawing a tree is placing the nodes. In this case, the y-value of a node is its depth in the tree and the x-value is the index of a terminal node in the sentence or the minimum of x-values among a non-terminal's descendants. An animated recursive descent parser for arithmetic expressions requires about 100 lines of `awk`. One user reports that he uses ANIM to show lambda calculus expressions in parse-tree form.

We'll now consider the domain of graph algorithms. Figure 13 shows the operation of Christofides' heuristic [1976] for constructing approximate traveling salesman tours. The left panel shows the

```
func gen(sym, depth,   i, j, origx) {
    origx = globalx
    print "text", origx, -depth, "bullet"
    print "text ljust", origx, -depth, "\"  " sym "\""
    if (sym in lhsct) {
        i = int(lhsct[sym] * rand()) + 1
        for (j = 1; j <= rhsct[sym, i]; j++) {
            print "line", origx, -depth, globalx, -(depth+1)
            gen(rhslist[sym, i, j], depth+1)
        }
    } else
        globalx++
}

  { if (NR == 1) start = $1
    i = ++lhsct[$1]
    rhsct[$1, i] = NF-2
    for (j = 3; j <= NF; j++)
        rhslist[$1, i, j-2] = $j
  }

END { globalx = 0; srand(); gen(start, 0) }
```

Program 6: A random sentence generation program.

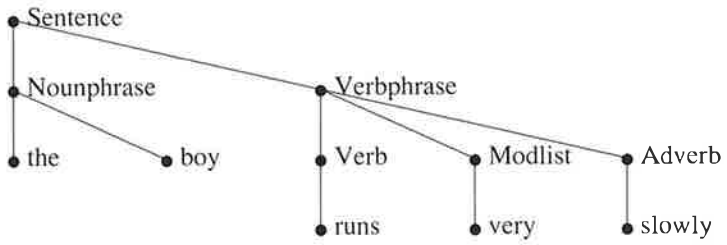

Figure 12: A parse tree for a sentence.

minimum spanning tree of a point set, the center panel shows an (approximate) matching of the odd-degree vertices in the tree, and the right panel shows the (approximate) tour constructed by an Eulerian traversal through the sum of the two previous graphs. Given a good geometric placement of the vertices, it is easy to animate many graph algorithms (though finding good layouts for nongeometric graphs can be difficult — see, for instance, Gansner, *et al.* [1988] and the references therein). Bentley & Kernighan [1987B] contains a detailed animation of Dijkstra's implementation of Prim's minimum spanning tree algorithm.

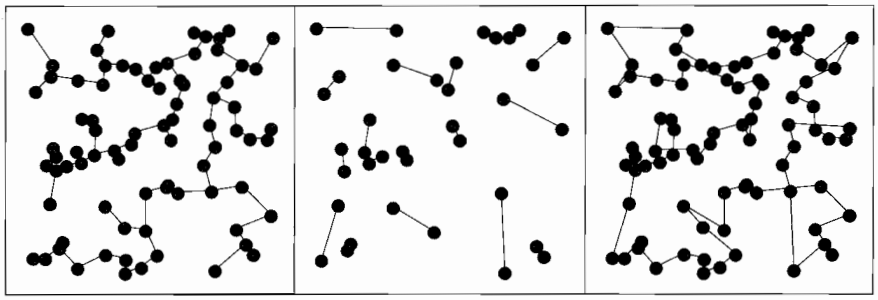

Figure 13: Christofides' TSP heuristic.

ANIM has found applications in numerical analysis tasks including the display of two-variable functions and adaptive meshes changing over time. The stars moving under Newtonian attraction in Bentley & Kernighan [1987B] can be viewed as solving simultaneous differential equations. One user writes:

> I have used ANIM for debugging a recursive algorithm that was part of my master's thesis. I had constructed some algorithms for reordering elimination trees (used for doing parallel Cholesky decomposition on sparse matrices). My algorithms were recursive and worked on large data sets, which made it difficult to use standard debugging tools. Instead I animated the tree for each reordering step and was able to step through the program and see when it was doing something wrong. Once the program was working, animation also made it possible for me to show how a faster algorithm was producing a poorer result.

Computational geometry is a natural domain for algorithm animation. Users have animated geometric programs for tasks such as triangulating simple polygons, finding intersections in sets of line segments, and computing the maxima and convex hulls of point sets. Bentley [1990] describes a 6000-line C++ program for performing experiments on geometric algorithms (including nearest neighbor searching, minimum spanning trees, and a wide variety of traveling salesman heuristics); about 200 lines of animation code proved indispensable for debugging the programs and experimenting with options in the heuristics.

Several users of ANIM have animated parallel algorithms, ranging from communication networks to tightly coupled systems to neural networks. Another user writes:

> I used ANIM with a simulator for a highly-parallel functional machine, specifically when debugging and trying to understand the subtleties of the interprocessor communication. The simulator spit out ANIM scripts, so that I could watch packets move around on a stick diagram of the machine architecture. When a deadlock occurred, I would single-step the last few packets to see what hole I'd gotten myself into; this was far better than the previous method, which amounted to reconstructing this information on a sheet of paper by hand.

Bentley & Kernighan [1987B] contains a program to animate the `malloc` storage allocator, and a picture it produced. Several users have found bugs in their use of the storage allocator by examining the progress of movies.

An easy way to learn any new system is to play games with it. Towers of Hanoi and Conway's Game of Life are popular victims for animation — each task requires about 30 lines of `awk`.

Dynamic graphical displays are frequently used by statisticians [Cleveland 1988]; they go far beyond the simple histogram in Figure 1. Clark and Pregibon [1990A] have used ANIM to provide an animation facility in the S system [Becker, *et al.* 1988], essentially using the S language instead of `awk` as the script generator. They have used the facility to implement prototypes of a wide variety of dynamic statistical graphics [Clark & Pregibon 1990B], including point cloud rotation, scatterplot linking, scrolling time series, and time series maps. An example of an animated time series, Rick Becker's movie of air pollution transport in the Northeastern United States, appears in Bentley & Kernighan [1987B].

Supporting Programs.

ANIM provides the bare bones of an animation environment. In the spirit of UNIX, we have enhanced the environment not by modifying the primary programs, but rather by using small filters to manipulate input and output files. We have already mentioned a program

for merging several animations into a race, and the stereo program and programs for rotation and spinning for 3-d animations.

Bentley & Kernighan [1987B] describes several other supporting programs. The `view.clicks` program summarizes the views and clicks in a script file. The `show.clicks` program creates a new script file with a new view in which all clicks are counted. Another program processes lines in the script file of the form

`#var` *name value*

The output script file has a new view named `variables`; it contains the name of each variable mentioned and its current value.

ANIM does not have a facility for counting clicks; rather, we use a command like

```
grep 'click comps' | wc
```

to see how many comparisons were made. We will even admit to using text editors to make minor changes to both script and intermediate files in times of need.

5. *Conclusions*

We believe that ANIM demonstrates that there is a role for an animation system that trades quality of output for ease and simplicity of use. ANIM has been used for a variety of applications, some significantly outside the algorithm animation area that was the original target, and it has been used by many people besides its authors.

There are a few features of ANIM that have proven especially useful, so much so that we feel they ought to be available in any animation system.

Independent views provide a way to see the same thing from several perspectives at the same time, or to see different things concurrently.

The use of named objects, and the implicit erasure of an object by re-drawing something with the same name, makes many kinds of animation trivial; merely drawing an object at a sequence of positions causes animation to happen ''for free.''

Movies are nice, but stills are much easier to distribute widely; the `stills` language has been heavily used to capture and present

relevant frames from movies in situations where the movie itself cannot be shown.

There are some obvious places where ANIM could be improved without compromising the fundamental goal of ease of use. It would be desirable to add more options, especially color and shading. A few more primitive geometric objects would be desirable; for example, ellipses could be added at essentially no cost. We have also had some requests for composite objects that could be drawn and erased as a unit. The `stills` language needs more ability to control layout; Sedgewick [1988] shows the kind of elaborate layout that is possible. Some applications call for a more sophisticated view of time and motion, such as "slide this collection of objects smoothly from here at this time to there at that time."

In conclusion, we believe that a simple animation system is useful for teaching, research, and, perhaps least obvious, just plain programming. Such a system need not be elaborate, nor does it need to produce superb output. Ease of use and wide availability are much more important.

Acknowledgements

We are deeply indebted to Howard Trickey, who has made ANIM work and kept it working for the X window system, a task far harder than writing ANIM itself. We are also grateful to Rick Becker, Linda Clark, Doug McIlroy, Peter Nelson, Daryl Pregibon, Howard Trickey, Chris Van Wyk, and an anonymous referee for helpful comments on this paper.

References

A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, MA, 1988.

R. Baecker, *Sorting out Sorting*, University of Toronto, 1981. 25 minute color sound film.

R. A. Becker, J. M. Chambers, and A. R. Wilks, *The New S Language*, Wadsworth, Pacific Grove, CA, 1988.

J. L. Bentley, D. S. Johnson, T. Leighton, and C. C. McGeoch, An Experimental Study of Bin Packing, *Proc. of 21st Annual Allerton Conf. on Communication, Control, and Computing*, pages 51-60, October 1983.

J. L. Bentley and B. W. Kernighan, A System for Algorithm Animation, *Fourth UNIX Computer Graphics Workshop*, Cambridge, MA, October 1987.

J. L. Bentley and B. W. Kernighan, A System for Algorithm Animation (Tutorial and User Manual), *AT&T Bell Laboratories Computing Science Technical Report 132*, 1987.

J. Bentley, Tools for Experiments on Algorithms, *Proc. CMU 25th Anniversary Symposium*, Pittsburgh, PA, September 1990.

M. Brown and R. Sedgewick, Techniques for Algorithm Animation, *IEEE Software* 2(1):28-39, January 1985.

N. Christofides, Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem, *Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University*, 1976.

L. A. Clark and D. Pregibon, An Animation Device Driver for S, *Proc. Stat. Graphics ASA*, August 1990.

L. A. Clark and D. Pregibon, Prototyping Dynamic Graphics Functions in S, *Proc. COMPSTAT 90*, September 1990.

W. S. Cleveland, *Dynamic Graphics*, Wadsworth, Belmont, CA, 1988.

E. R. Gansner, S. C. North, and K. P. Vo, DAG — A Program that Draws Directed Graphs, *Software—Practice & Experience* 18(11):1047-1062, November 1988.

R. Sedgewick, *Algorithms, Second Edition*, Addison-Wesley, Reading, MA, 1988.

J. T. Stasko, Tango: A Framework and System for Algorithm Animation, *IEEE Computer* 23(9):27-39, September 1990.