AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 132

A System for Algorithm Animation
Tutorial and User Manual

Jon L. Bentley
Brian W. Kernighan

August 6, 1991

A System for Algorithm Animation
Tutorial and User Manual

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A program or an algorithm can be animated by a movie that graphically represents its dynamic execution. For instance, a memory allocator might be animated by lines that appear when memory is allocated and disappear when it is freed; a sort might be animated by a randomly scrambled sequence of lines being permuted into order. Such animations are useful for debugging programs, for developing new programs, and for communicating information about how programs work. This paper describes a basic system for algorithm animation: the output is crude, but the system is easy to use; novice users can animate a program in a couple of hours. The system currently produces movies on Teletype 5620 terminals and workstations that support the X window system, and also renders movies into ‘‘stills’’ that can be included in *troff* documents. This paper is a user manual and a tutorial introduction to algorithm animation using the system.

August 6, 1991

A System for Algorithm Animation
Tutorial and User Manual

Jon L. Bentley
Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Dynamic displays are better than static displays for giving insight into the behavior of dynamic systems. The pictures in Figure 1, for instance, illustrate four equal-mass bodies moving in the plane under Newtonian attraction.

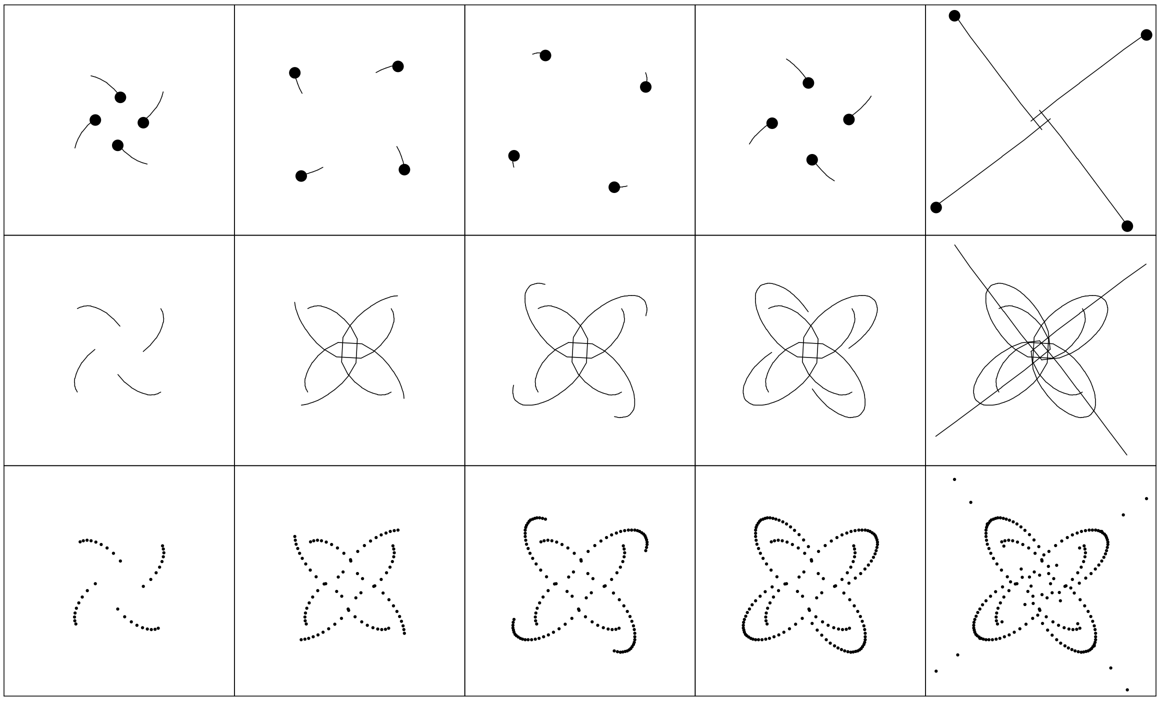


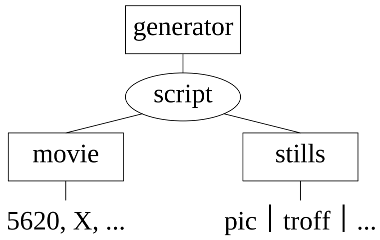
Figure 1: Bodies moving under Newtonian attraction

Time marches across the pictures left to right, at roughly equal intervals. Each column, called a *snapshot*, depicts three *views* of the bodies. The top view gives the current position as a dot; the tail formed by the last few positions hints at the velocity and the acceleration. The middle view renders the path of each body as a line. The bottom view depicts the history of velocities by dots recorded at equal time intervals: low velocities give close dots and high velocities leave the dots far apart.

The first snapshot shows the bodies starting slowly, far apart. In the second snapshot they have recently experienced a high-velocity encounter near the center. In the third snapshot the bodies are again far apart, moving slowly toward another encounter in the fourth snapshot. In the fifth snapshot the large time-step of our simulation program violates conservation of energy and sends the bodies racing away from each other.

This paper describes the animation system that produced those pictures. A sixty-line C simulation program was augmented with eight *printf* statements to generate a *script* file describing the paths of the four bodies in the three views. That file was processed by a program named *stills* to produce the pictures above, using *pic* and *troff*;

we were able to control what frames were displayed, in what size and form. A program named *movie* displays the same data on a Teletype 5620 terminal or an X workstation; the viewer can control the speed of display, proceed forward or backward through time, and change the screen layout to emphasize certain views. Those components can be depicted as:



Several systems have been developed for algorithm animation [3]. Most of those systems produce animations of very high quality; unfortunately, they are expensive in both programmer time and CPU time. Our system is at the opposite end of the spectrum: its output is primitive, but the system 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 our system was designed primarily with program animation in mind, the gravitational example shows that it can be useful in other domains as well.

Section 2 introduces the system by animating a sorting algorithm. Sections 3 through 5 describe the three primary components of the system: script files, *movie*, and *stills*. Section 6 describes the animation of several larger programs, and Section 7 discusses a few everyday matters about using the system.

2. A Simple Example — Sorting

Insertion sort is the method most card players use. As each card is dealt, it is inserted into its proper place among the existing cards. To sort the array $X[1..N]$, insertion sort maintains the sorted subset in $X[1..I]$ and increases I from 2 to N. The subarray $X[1..1]$ is sorted by definition. The first phase of the algorithm sifts $X[2]$ down so that $X[1..2]$ is sorted, the second phase sifts $X[3]$ down so that $X[1..3]$ is sorted, and so on.

Here is an implementation of insertion sort in *awk* [1]:

```
awk '
BEGIN { n = ARGV[1]     # numer of elements to generate and sort
        for (i = 1; i <= n; i++) x[i] = 1 + int(25*rand())
        for (i = 2; i <= n; i++) {
                for (j = i; j > 1 && x[j] < x[j-1]; j--)
                        swap(j-1, j)
                show()
        }
}
function swap(i, j,   t) {
        t = x[i];  x[i] = x[j];  x[j] = t
}
function show(  i) {
        for (i = 1; i <= n; i++)
                printf("%3d", x[i])
        printf("\n")
}' $*
```

The first `for` statement sprinkles `x[1..n]` with random integers in the range 1..25, and the second and third `for` statements perform the insertion sort. The function `swap` exchanges array elements; `show` prints the current state of the array.

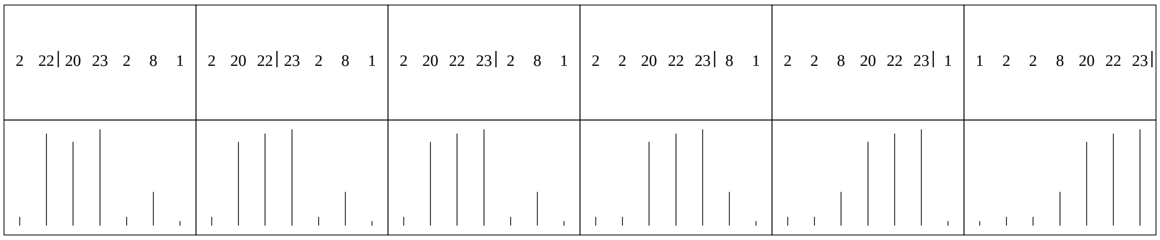
If we run this program with `n` set to 7, we get this static display of sorting a 7-element array of random integers:

```
2 22 20 23  2  8  1
2 20 22 23  2  8  1
2 20 22 23  2  8  1
2  2 20 22 23  8  1
2  2  8 20 22 23  1
1  2  2  8 20 22 23
```

The animation system provides an alternative: by adding a few more `print` statements to the program, we can produce input for the animation system, thus providing a dynamic display of the sorting process. Furthermore, we can use graphics as well as text, to give a more visual presentation of the algorithm.

For this example, we decided to present two views. The top view simply shows the numbers in the array as they are sorted; it is essentially the same as the textual output above, except that we have added a vertical bar: elements to the left of the bar are in order. The bottom view is graphical: the value of each element is represented by the length of the corresponding vertical line.

These six snapshots show the state after each phase of the sorting algorithm:



phase: 1 phase: 2 phase: 3 phase: 4 phase: 5 phase: 6

Although the two views represent the same information, they are useful for different tasks. The textual nature of the top view is easier for novices to follow, and the complete information is handy for debugging small examples. The more visual bottom view is better for displaying large sorts.

The Program.

Here is the complete *awk* program `is.gen` that generated the animation depicted above.

```
awk '
BEGIN { n = ARGV[1]     # number of elements to sort
        for (i = 1; i <= n; i++) x[i] = 1 + int(25*rand())
        for (i = 1; i <= n; i++) draw(i)
        for (i = 2; i <= n; i++) {
            for (j = i; j > 1 && less(j, j-1); j--)
                swap(j-1, j)
            print "view text"
            print "bar: text ljust", i, 1, "\"  |\""
            print "click phase"
        }
}
function draw(i)  {
        print "view text"
        print "a" i ": text", i, 1, x[i]
        print "view geom"
        print "a" i ": line", i, 0, i, x[i]
}
function swap(i, j,   t) {
        t = x[i];  x[i] = x[j];  x[j] = t
        draw(i); draw(j)
        print "click swap"
}
function less(i, j) {
        print "click comp"
        if (x[i] < x[j]) return 1; else return 0
}' $*
```

It is very similar to the first version, but we have added the functions `less` to make comparisons and `draw` to display elements, as well as several `print` statements. In addition to performing their sorting functions, `less` and `swap` record their actions by printing output into a script file that will serve as input to *movie* and *stills*. This animation uses four commands: `line`, `text`, `view` and `click`.

A line from (x_1, y_1) to (x_2, y_2) is drawn by a command of the form

optional_label: `line` x_1 y_1 x_2 y_2

(Throughout this paper, literals are shown in `typewriter` font and categories are in *italics*.) The coordinates of the line can lie in any range; later programs will scale them appropriately. The label on a line is optional. When a labeled object is drawn the object that previously had that label is erased. Thus for all lines that have the same label, the act of drawing one line erases its predecessor. (There is also an explicit `erase` command.)

Text is produced at (x, y) by a command of the form

optional_label: `text` x y *anything at all*

As with lines, labels are permitted; re-use of a label erases whatever object previously had that label. For example, the vertical bar is always printed with the label `bar`, so each bar erases the previous one.

The `view` command is used to place output in a particular view. There are two views here, `text` and `geom`:

```
view text
view geom
```

The `draw` function draws text in the `text` view and lines in the `geom` view, relying on implicit erasure to remove the previous object before creating a new one. Different views are independent, so a label like `a1` can be used in several views without interference.

Interesting events are marked by the `click` command:

```
click swap
click comp
click phase
```

Stills and *movie* can refer to each click with this mechanism, as we will see in more detail shortly.

Labels, view names and click names are arbitrary and unrelated to one another.

The Script File.

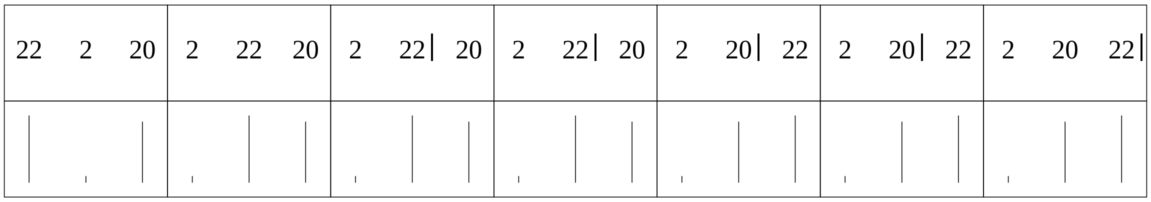
Executing the command

```
is.gen 3 >is1.s
```

produces the script file `is1.s` containing a sort of three elements. The basename `is1` is for ‘‘insertion sort 1’’; the suffix `.s` identifies it as a script file and is required by the animation system. Here is `is1.s`, printed in two columns to save space:

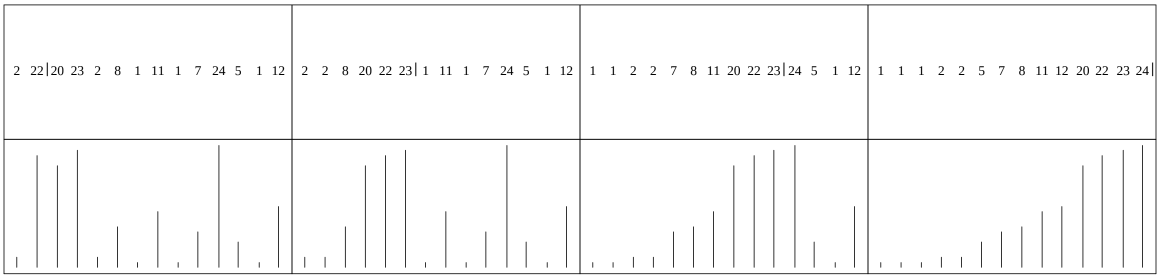
```
view text
a1: text 1 1 22
view geom
a1: line 1 0 1 22
view text
a2: text 2 1 2
view geom
a2: line 2 0 2 2
view text
a3: text 3 1 20
view geom
a3: line 3 0 3 20
click comp
view text
a1: text 1 1 2
view geom
a1: line 1 0 1 2
view text
a2: text 2 1 22
view geom
a2: line 2 0 2 22
click swap
view text
bar: text ljust 2 1 "  |"
click phase
click comp
view text
a2: text 2 1 20
view geom
a2: line 2 0 2 20
view text
a3: text 3 1 22
view geom
a3: line 3 0 3 22
click swap
click comp
view text
bar: text ljust 3 1 "  |"
click phase
```

Here is a picture of all clicks in the script file `is1.s`, produced by *stills*.



comp: 1 swap: 1 phase: 1 comp: 2 swap: 2 comp: 3 phase: 2

As before, time goes from left to right. Both views of the current state are depicted in a snapshot at each `click`; the label below each frame tells the name and number of the click. This detailed picture illustrates the animation system; here is a sparser picture that illustrates insertion sort on a 14-element array:



phase: 1 phase: 5 phase: 9 phase: 13

Making Stills.

The last picture was included in this document by this *stills* description:

```
.begin stills
view text ""            # no title on text view
view geom ""            #    or on geometry view
print phase 1 5 9 13    # print these clicks of phase
file is2.s              # name of script file
frameht 0.7             # height of frame in inches
framewid 1.5            # width of frame
medium -5               # size down 5 points for medium text
.end
```

The description is delimited by the lines `.begin stills` and `.end` (which correspond, for instance, to `.EQ` and `.EN` in `eqn` or `.TS` and `.TE` in `tbl`). Text following the sharp character # is a comment to be ignored. The `view` statements specify the empty string as the title for both views of the data. The `print` statement displays snapshots at the requested `click`s of `phase`.

The last four statements are parameter assignments of the form

parameter_name *value*

Assignment to the `file` parameter names the script file to be displayed. The next two assignments set the height and width of frames, and the final statement causes `medium` text (the default text size) to be set five points smaller than the current *troff* point size.

Viewing A Movie.

To watch a movie on the 5620, make a window of suitable size and shape and in it type the command

```
is.gen 20 | movie
```

If you want to access the script later, type instead

```
is.gen 20 >is.s
movie is.s
```

After a pause to run `is.gen` (about 15 seconds on a VAX™-750 in this case), the 5620 down-loading procedure will begin; after that (another 30 seconds), the data for the movie itself will begin to appear. When this is finished (also about 30 seconds), a message about the number of bytes sent (about 9500) appears in the upper left corner.

At this point, the mouse buttons can be used to redisplay the movie. Button 3 is the main control. It has 9 menu items:

```
again
faster 1
slower 1
1 step
backward
fatter 1
thinner 1
or mode
new file
Quit?
```

To play the movie again, select `again`. (This movie takes about 6 seconds at full speed.) You can stop it at any point by pushing any button; a further push of button 1 continues it. To slow the display, select `slower`; each selection halves the speed by increasing a wait interval by a factor of two. After three selections the menu reads

```
again
faster 8
slower 8
...
```

Try selecting `again` to see the sort once at this speed, then select `faster` three times to get back to full speed.

The items labeled `fatter` and `thinner` control the thickness of lines in an analogous manner. Selecting `fatter` several times results in obese lines; you may return things to normal with `thinner`.

Three menu items control binary mode settings: `1 step` or `run`; `backward` or `forward`; and `or mode` or `xor mode`. For each, the label indicates the next state, not the current state.

Normally the movie is played from beginning to end without pause. The menu item labeled `1 step` puts it into a mode where it displays only one ‘‘step’’ each time button 1 is pushed. This allows you to inspect the sort a frame at a time. This item changes to `run` when `1 step` has been selected so you can revert to continuous action.

The `backward` item causes the steps to be taken in reverse order (time runs backwards); the menu item changes to `forward` when selected. After the sort ends, select `backward` and `again` and watch the array scramble itself before your very eyes. Judicious use of `1 step` and `backward` and `forward` make it easy to examine a few snapshots in detail.

Normally items are displayed in ‘‘exclusive OR’’ mode, which means that a bit drawn over a previous one erases it. If you are drawing numerous objects in a crowded area, this can lead to unintended erasures. The `or mode` item switches to ‘‘inclusive OR’’ for drawing objects, and erases objects by clearing. Some movies are far better in one mode than the other.

The `new file` item allows one to view a new movie without having to reload the *movie* program into the 5620. We’ll see how it works in Section 4.

The `Quit?` menu item is the way to exit. If selected, it displays a skull and crossbones to warn that its effect is irreversible. Pushing button 3 again exits; a different button avoids quitting.

Button 2 contains menu items to control the size and shape of the views on the screen and to control the meaning of a ‘‘step’’ in 1-step mode. For this sorting movie, button 2 looks like:

```
view text
view geom
click comp
click swap
click phase
```

When a view is selected, you can sweep a rectangle in which that view is to be displayed; the use is exactly like the `New` menu item in *mux*. You can arrange the views any way you like; try deleting the textual view by sweeping its rectangle out of the window.

When a `click` is selected, 1-stepping proceeds to the next occurrence of that click. So, for example, to see swaps one at a time, select `click swap` on button 2 then select `1 step` on button 3. Each hit of button 1 will pause at the next `click swap` statement in the script file, in either `forward` and `backward` mode.

Multiple clicks may be selected. Selecting both `click comp` and `click swap` will cause a pause after every comparison and every swap. Selected clicks are marked with an asterisk in the menu; selecting a click that already has an asterisk removes the asterisk and turns off the click.

In `run` mode, the movie runs at full speed until it encounters a selected click, then pauses for a time proportional to the selected speed. Selecting no clicks is equivalent to having selected an implicit click that occurs after each geometric object (text, line, circle, rectangle) is drawn or erased, so animations run faster when some clicks are turned on (with `click phase` selected, for instance, the sorting movie runs nearly twice as fast).

Summary.

This exercise illustrates the capabilities and limitations of our animation system. 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 (we would like to include a movie in this document, for instance, but paper is easier to distribute than videotape). A sophisticated animation system might require 500 lines of code to produce beautiful animations of insertion sort. Our output is unpolished by comparison, but it is adequate for many purposes and requires just a few dozen lines.

If our system 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 extra services like these:

Device Independence. A script file can be rendered as a movie on a 5620 or an X11 workstation; the system is designed to make it easy to port to additional output devices. 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 avoids much of the tedium of bookkeeping. Click names mark key events; they can be used to group related events.

Independent Views. Different simultaneous views of a process are crucial for animating algorithms. In our system, 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. Labels in different views are independent.

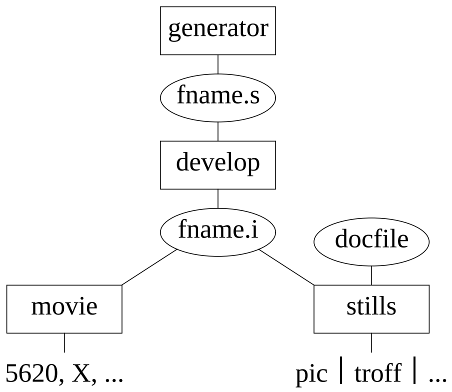
Viewer Control. Both *movie* and *stills* allow the viewer to select which views will be displayed and which clicks will be recognized. Additionally, *movie* allows the viewer to go forward or backward, in single steps or running at a selected speed.

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

Our system does not support interactive animations, however: once the script has been generated, there's no way to change it except to generate it again.

3. The Script Language

This section is a more complete description of the script language in which animations are described. A script file is processed by the heretofore unmentioned program *develop*; errors in script files are reported by that program. The output of *develop* feeds *stills* and *movie*:



The command `develop fname.s` produces the *intermediate* file `fname.i` from the script file `fname.s`, unless `fname.i` already exists and is newer than the corresponding script file. Fortunately, most users need not be concerned with intermediate files and the *develop* program; both *movie* and *stills* call *develop* implicitly. Appendix I defines the format of an intermediate file.

The script language provides commands to draw geometric objects and commands that control the pictures. A line whose first non-blank character is # is a comment; comments may not appear on the line after other commands. Blank lines are ignored.

Geometric Commands: `text`, `line`, `box`, `circle`.

Geometric commands describe text, lines, rectangles, or circles. They share the common form

optional_label: *command options x y additional parts*

If a label is present, it names the object and will implicitly erase any existing object with the same name in the same view. The options are a (possibly null) list of names, terminated by the next numeric field.

Text is placed at a position by the command

optional_label: `text` *options x y string*

The available options are

```
[center]  ljust  rjust  above  below
small  [medium]  big  bigbig
```

At most one option may be selected from each line; if none is selected, the option in brackets is used. The first line describes text position, and the second line describes text size. The text string may be quoted. If there is no leading quote, then the string starts at the first non-blank character and continues until the end of the line. If there is a leading quote, subsequent leading white space is kept and any trailing quote at the end of the line is removed; intermediate quotes are kept. Some strings, including (but not necessarily limited to) `bullet`, `dot`, `circle`, and `times`, are recognized by later processors.

A line is drawn by

optional_label: `line` *options* x_1 y_1 x_2 y_2

The available options are

```
[-]  ->  <-  <->
[solid]  fat  fatfat  dotted  dashed
```

The first line of options describe whether the line should be drawn with arrowheads; the default is without arrowheads. The option `<-` puts an arrowhead at the (x_1, y_1) end of the line, `->` puts an arrowhead at the other end, and `<->` puts them at both ends. The second line of options describes the body of the line.

A rectangle is drawn by

optional_label: `box` *options* *xmin ymin xmax ymax*

The only options are

```
[nofill]  fill
```

Under the default `nofill` only the border of the box is drawn; a `fill`ed box has a solid interior as well.

A circle is drawn by

optional_label: `circle` *options* *x y* *radius*

The radius is measured in the *x* dimension. Circles will look right only if *x* and *y* are in about the same range. As with rectangles, the options are

```
[nofill]  fill
```

Control Commands: `view`, `click`, `erase`, `clear`.

The current view is set by the statement

`view` *name*

If there are no view statements in the script file, *develop* generates a single implicit view named `def.view`. If geometric objects appear before the first view statement, they go in that view and a warning message is generated.

A click is named by

`click` *optional_name*

If no name is present, then the name `def.click` is implicitly supplied.

A labeled geometric object can be explicitly erased by the command

`erase` *label*

develop prints a warning if the object was never defined or has already been erased. The various views have distinct name spaces; the same label may be applied to two unrelated objects in two different views. All objects in the current view can be erased by the statement

```
clear
```

None of these commands may have labels.

Summary.

The script language contains the following commands; options are indented on a subsequent line, with defaults in brackets:

```
# comment
optional_label: line   options   x1 y1 x2 y2
        [-]  ->  <-  <->
        [solid]  fat  fatfat  dotted  dashed
optional_label: text   options   x y   string
        [center]  ljust  rjust  above  below
        small  [medium]  big  bigbig
optional_label: box   options   xmin ymin xmax ymax
        [nofill]  fill
optional_label: circle   options   x y   radius
        [nofill]  fill
view   name
click   optional_name
erase   label
clear
```

The shell command `develop fname.s` makes the intermediate file `fname.i` from the script file `fname.s`, if `fname.i` is out of date. The purpose of the intermediate file is to trade increased space (for storing the intermediate file) for reduced run time (a script file is developed just once, not each time it is used). The *movie* and *stills* shell scripts could be rewritten to pipe their inputs through *fdevelop*, a filter form of *develop*. The *fdevelop* program can handle script files with at most 20,000 lines; the argument `-l`*n* changes the upper bound to *n* instead. It can similarly handle at most 10,000 pieces of geometry active at any time; the argument `-s`*n* (for ‘‘slots’’) changes that upper bound. Error messages tell when these bounds need to be increased. After any `-l` and `-s` arguments, *fdevelop* can have an optional file name. If there is a name, that is the input file; otherwise, the standard input is used. The output is written on the standard output.

4. The Movie Program

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. As the file is shipped, the line number in the intermediate file is displayed in the upper-left corner of the window every 100 lines. Afterwards, the total number of bytes stored is displayed in that location. The terminal process allocates 80,000 bytes (typically 5-10,000 objects) for the picture; the argument `-m`*n* sets the allocation to *n* instead.

The button 3 menu was sketched in Section 2. In general, drawing can be interrupted at any point by pushing any button, then resumed by pushing button 1.

Four menu items control two variables:

```
faster [speed]
slower [speed]
```

decrease (halve) and increase (double) the pause at selected clicks, and

```
thinner [line width]
fatter [line width]
```

alter the width of lines. (If the line thickness is *n*, then `solid` lines are $2n-1$ bits wide; `fat` and `fatfat` lines are larger.) Three menu items control binary attributes:

```
backward    forward
or mode     xor mode
1 step      run
```

The mode displayed on the menu is the next state, not the current one. If the program is currently in `or mode`, for instance, then `xor mode` is displayed.

The `new file` item allows one to view a new movie without downloading the *movie* program again. After selecting that item, text in the upper left of the window asks for the name of the intermediate file to be processed. The *movie* program does not call *develop* to make the intermediate (`.i`) file from the script (`.s`) file; that is the responsibility of the user, typically in a separate window.

Button 2 lists views and clicks. Selecting a view results in an icon for sweeping a rectangle, as in *mux*. Views may be positioned anywhere; portions positioned outside the window will not be shown. Initially, views have a 5 percent margin at each edge; this margin is zero for views that have been reshaped. If the window itself is reshaped, all views revert to the default position and margin.

Normally, in 1-step mode, the display pauses after each primitive object (line, text, etc.) has been drawn or erased. If any clicks are defined and turned on by button 2, however, then the display pauses only at those points. Any number of clicks may be turned on. Clicks that are turned on are marked with an asterisk; they may be turned off by selecting them again.

As it is for the 5620, so it is for X workstations, although the exigencies of the X window system have forced us to curtail some features. To keep the code relatively portable, there are again two processes, so the window in which one starts the animation clones another window of uncontrolled size, shape and position where the animation itself occurs.

The current terminal programs support many, but not all, text size and line mode options.

If *movie* has a single argument, it must be either a `.s` or `.i` file; *movie develop*s a `.s` file. If it has no arguments, then it will pipe the standard input through *fdevelop*; no intermediate file is created. For more exotic situations, use *fmovie*: with no arguments, it projects the intermediate file from the standard input; with a single argument, it projects that intermediate file.

5. The Stills Language

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
```

There are three classes of statements in a *stills* description: `print`, `view`, and parameter assignments. Only two statements are mandatory in a particular description: an assignment to the `file` parameter and a `print` statement. Text following the sharp symbol # is discarded as a comment; blank lines are ignored.

There may be any number of `print` statements of any combination of the following forms:

`print all`
`print final`
`print` *clickname* `all`
`print` *clickname number number number* `...`

The first statement causes a snapshot to be drawn at each `click` statement in the script file; the second draws one at the end of the file. The third form prints all `click`s of the designated name, and the fourth prints only the clicks enumerated in the list of numbers.

View statements select which views are to be printed in snapshots and assign titles to views.

`view` *name optional title*

The views appear in the order they are named, either top-to-bottom if time goes `across` the page or left-to-right if time goes `down` the page. If there are no `view` statements, each snapshot depicts all views in the script. If the title is enclosed in quote marks, they are stripped and leading space is kept. If no title is given for a particular view, the view name itself is used as a title; thus an empty title is needed to turn off printing.

Parameters can be set by assignment statements of the form

parameter_name value

All parameters are reset to their default values at each `.begin` *stills* statement. Numeric values may be of the form `n`, `+n`, `-n`, or absent (zero default).

Filename parameter: `file`. The script file is named by assigning to the `file` parameter with a statement of the form `file basename.s`. The *stills* program *develop*s that script file and then reads `basename.i`.

Text sizes: `small medium big bigbig`. The assignment `small -2` causes text with the `small` option to be printed two point sizes smaller than the current *troff* point size. Assigning `+5` or `5` to `bigbig` increases the point size by 5 for `bigbig` text. All changes are relative; there is no way to set absolute point size.

Line widths: `solid fat fatfat`. Relative size changes for line widths, exactly as for text sizes.

Direction: `across down`. By default, snapshots proceed across the page in time. The assignment `down 5` causes time to proceed down the page for 5 snapshots before starting a new column; `down 0` or `down` yields as many snapshots as will fit on an 8-inch page. The assignment `across 7` gives 7 snapshots in a row before starting a new row; `across 0` or `across` adapts to a 6-inch width.

Frame parameters: `frameht framewid margin`. The height and width of frames are given in inches; defaults are 1.5. The margin of a frame is the white space surrounding the data; the default is 0.05, or a five percent border.

Optional parts: `frames times`. Frames are the solid borders around pictures; times are the click name and number that triggered a snapshot. Both have the default value `vis` and are shown; they may be suppressed by assigning them the value `invis`.

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. The parameter names in the right column may appear on the left side of a name/value assignment:

Filename	`file`
Text sizes	`small medium big bigbig`
Line widths	`solid fat fatfat`
Direction	`across down`
Frame parameters	`frameht framewid margin`
Optional parts	`frames times`

6. Larger Animations

The examples in this section illustrate algorithms on several classes of data structures, including arrays, trees and graphs. The graphical style is simple; this is easy for the animator and effective for the viewer. The programs in this section are not presented as paradigms of good programming style; rather, they show how succinct programs can yield useful animations.

In the paper cited earlier, Brown and Sedgewick employ several conceptual levels in animating an algorithm.

Execution. At one extreme is the program to be animated, executing on the data of interest.

History of ''Interesting Events.'' What events in a computation should be depicted in the animation? The simple animation of insertion sort in Section 2 used augmented `less` and `swap` routines to capture two primitives of most sorting programs: comparisons and data movements. It also explicitly recorded information associated with the interesting event of finishing a phase.

Geometric interpretation of events. One must next decide how to represent the interesting events pictorially. An array of integers, for instance, might be represented by a sequence of numbers, a scatterplot of bullets, or a sequence of vertical bars.

Rendering on a device. In our system, this is the job of *movie* and *stills*.

The sample program `is.gen` identified interesting events and gave them a geometric interpretation in separate procedures within a single program. Alternatively, it may be more convenient to implement the various tasks as a pipeline of two programs: a generator program writes interesting events that are given a geometric interpretation by a program that writes a script file.

One usually prints a `click` statement immediately after the event it marks. Sometimes, though, one draws additional objects to highlight the event, which are erased immediately after the `click`. In that case, one uses a sequence like

draw event E
highlight E
`click E`
remove highlights

Sorting, Again.

We will return to the subject of sorting with a more interesting animation: a race of insertion sort versus quicksort. We will use the same insertion sort we saw earlier, and a quicksort described in Section 10.2 of [2]. The two sorting routines and their supporting functions are contained in this *awk* program:

```
awk '
BEGIN { n = 50
        for (i = 1; i <= n; i++) x[i] = int(100*rand())
        for (i = 1; i <= n; i++) draw(i)
        qsort(1, n) # or isort()
}
function draw(i)  {
    print "a" i ": line", i, -3, i, x[i]
}
function swap(i, j,   t) {
    t = x[i];  x[i] = x[j];  x[j] = t
    draw(i); draw(j)
    print "click swap"
}
function less(i, j) {
    print "click comp"
    if (x[i] < x[j]) return 1; else return 0
}
function isort(   i, j) {
    for (i = 2; i <= n; i++)
        for (j = i; j > 1 && less(j, j-1); j--)
            swap(j-1, j)
}
function qsort(l, u,   i, m) {
    if (l >= u) return
    swap(m = l, l + int((u-l+1)*rand()))
    for (i = l+1; i <= u; i++)
        if (less(i, l)) swap(++m, i)
    swap(l, m)
    qsort(l, m-1)
    qsort(m+1, u)
}'
```

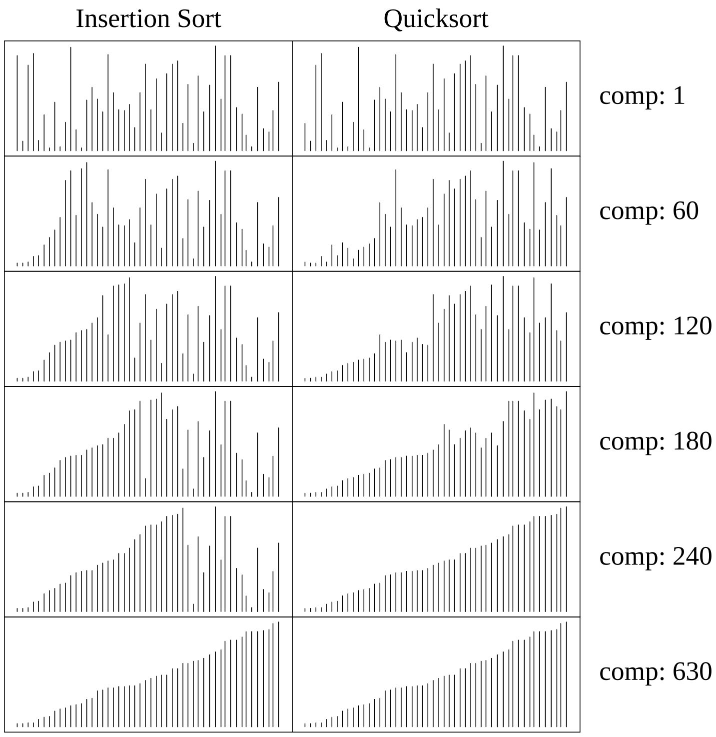
Quicksort is called as `qsort(l,u)` to sort the subarray `x[l..u]`. The function `draw` represents the numbers to be sorted as vertical lines. The `BEGIN` block sets `n`, initializes the array, draws the initial representation, and then calls a sort routine. As it stands, the code depicts a 50-element quicksort.

Some algorithm animation systems present races of programs by implementing a simple form of time slicing. To create a race in our system, we first ran the program into the file `qs.s`. We then changed the final line in the `BEGIN` section to call `isort`, and ran that into the file `is.s`. Finally, we merged the two scripts into one with this *awk* program:

```
awk '  # merge two sort animations
BEGIN { # file name   label                status of last getline
    f1 = "is.s";      l1 = "view insert";  s1 = 1
    f2 = "qs.s";      l2 = "view quick";   s2 = 1
    sep = "click comp"
    while (s1 && s2) {
        print l1; while ((s1 = getline <f1) > 0 && $0 != sep) print
        print l2; while ((s2 = getline <f2) > 0 && $0 != sep) print
        print sep
    }
    if (s1 > 0) { print l1; while (getline <f1) print }
    if (s2 > 0) { print l2; while (getline <f2) print }
}' >race.s
```

The loop copies the script of insertion sort until it encounters a `click comp` statement; it then copies quicksort until it encounters the corresponding `click`, at which points it prints the `click` statement. The variables `s1` and `s2` store the status of the most recent `getline`s of the two files; the status is one if a record was found and zero if an end-of-file was encountered. When one file is exhausted, the remainder of the other is copied. (A production splicer should have a more graceful interface for naming the files and views.)

The resulting script file is an execution of the sorts in two parallel views, synchronized by comparisons:

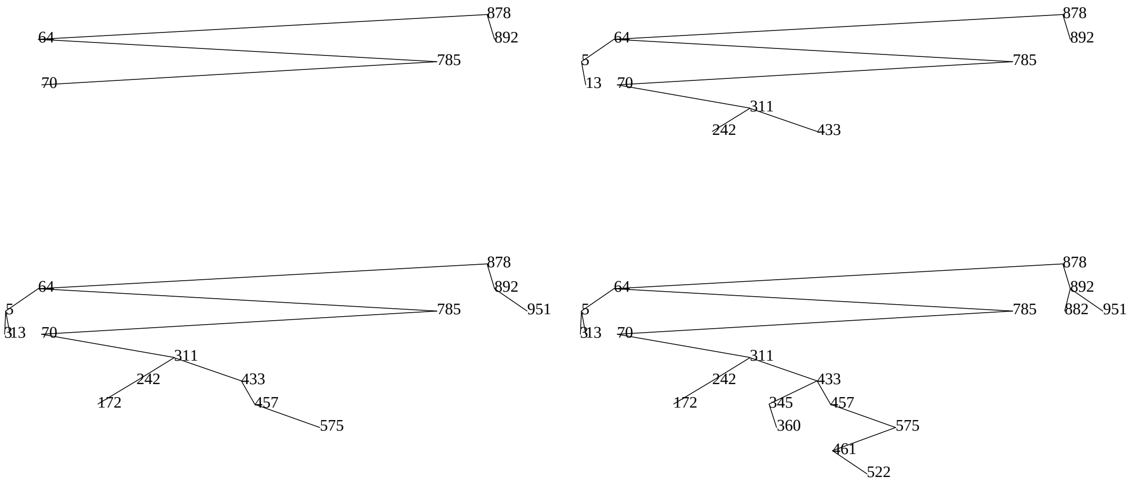


As before, insertion sort sifts each element into place in turn. This picture only hints at the operation of quicksort; insight into that algorithm requires the identification of interesting events beyond comparisons and swaps. Quicksort finishes after 240 comparisons, while insertion sort takes almost three times as long. For completeness, here is the *stills* input that produced the picture:

```
.begin stills
down
file race.s
frameht .6
framewid 1.5
view insert "Insertion Sort"
view quick "Quicksort"
print comp 1 60 120 180 240 630
.end
```

Trees.

Here are pictures of a (nonbalanced) binary search tree after inserting 5, 10, 15 and 20 random integers in the range 0..999:



In the last two frames, the labels for node 3 and 13 are squeezed too close together. The script was generated by this *awk* program:

```
awk '
BEGIN { n = 20; root = null = -1
        for (i = 1; i <= n; i++)
                root = insert(root, null, int(1000*rand()), 0)
}
function insert(p, pp, x, d) { # node p, parent pp, value x, depth d
        if (p == null) {
                val[p = ++nodecount] = x
                lson[p] = rson[p] = null
                if (pp != null) print "line", val[pp], 1-d, val[p], -d
                print "text ljust", val[p], -d, "\"" x "\""
                print "click insert"
        } else if (x < val[p])
                lson[p] = insert(lson[p], p, x, d+1)
        else if (x > val[p])
                rson[p] = insert(rson[p], p, x, d+1)
        return p
}
'
```

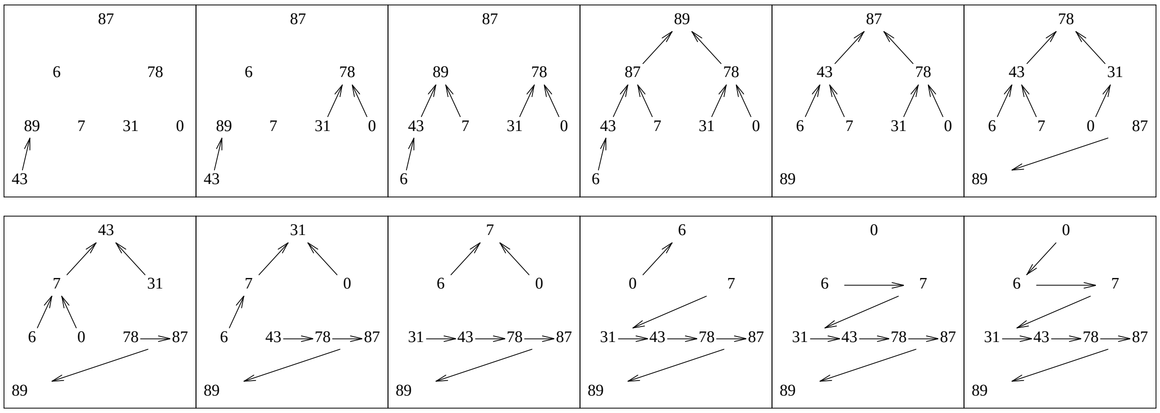
The animation code consists of just three lines in the `insert` procedure.

The pictures give insight into random binary search trees in spite of being rather ugly. The trees have two distinct failings: the depiction of individual nodes, and the layout of the entire tree.

A node is represented by its numeric value; each node is connected to its parent. If the shape of the tree is more important than the values it contains, one can delete the values entirely. We will shortly see a tree with more graceful edges.

The other aesthetic issue is the layout of the tree. The *y*-value is the depth of the node in the tree, which is a very robust choice (one could also use the time at which the node was inserted). The *x*-value in this example is simply the randomly generated value itself; there are many alternative choices. One could instead use the number of the node in an inorder traversal, which involves a multiple-pass algorithm: the tree is first built, then traversed and numbered, and the insertions are then reported with knowledge of the numbers. For this representation, it is crucial to separate the interesting events from their geometric representation; it is convenient to calculate these in two filters in a pipeline.

This animation of heapsort uses an alternative representation of trees. It was generated by a 50-line *awk* program.



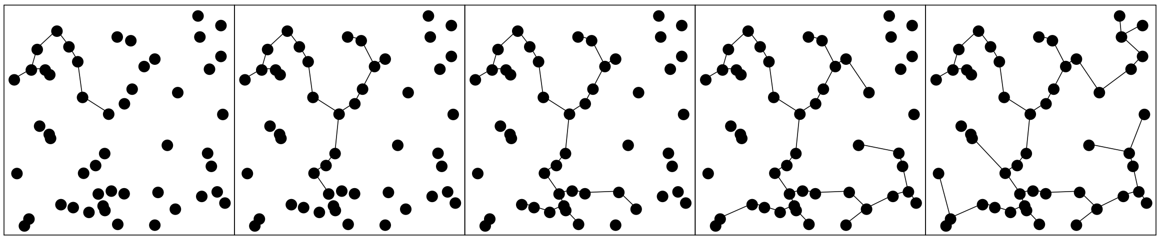
Each snapshot shows the result of a `sift` operation. The first four `sift`s build the heap; subsequent `sift`s maintain the unsorted elements as a heap. Arrows point from lesser elements to greater elements.

A node in the heap is represented by its value; lines between nodes have a single arrowhead and are chopped by twenty percent at each end. The root of the heap is at $(1/2,\ -1)$, its two children are at $(1/4,\ -2)$ and $(3/4,\ -2)$, their four children are at $(1/8,\ -3)$, $(3/8,\ -3)$, $(5/8,\ -3)$, $(7/8,\ -3)$, etc.

Other tree layouts proved useful in animating two algorithms dealing with parse trees. In both cases, a node's *x*-value was the minimum of the *x*-values among the node's descendants (equivalently, the *x*-value of its leftmost child); all edges were therefore either vertical or slanting down to the right. A random sentence generator built the tree left-to-right and top-down; just as in the binary search tree, the height of a node was one less than its parent. A parser built the tree in postorder and bottom-up: the height of a node was one greater than the maximum of the heights of its children.

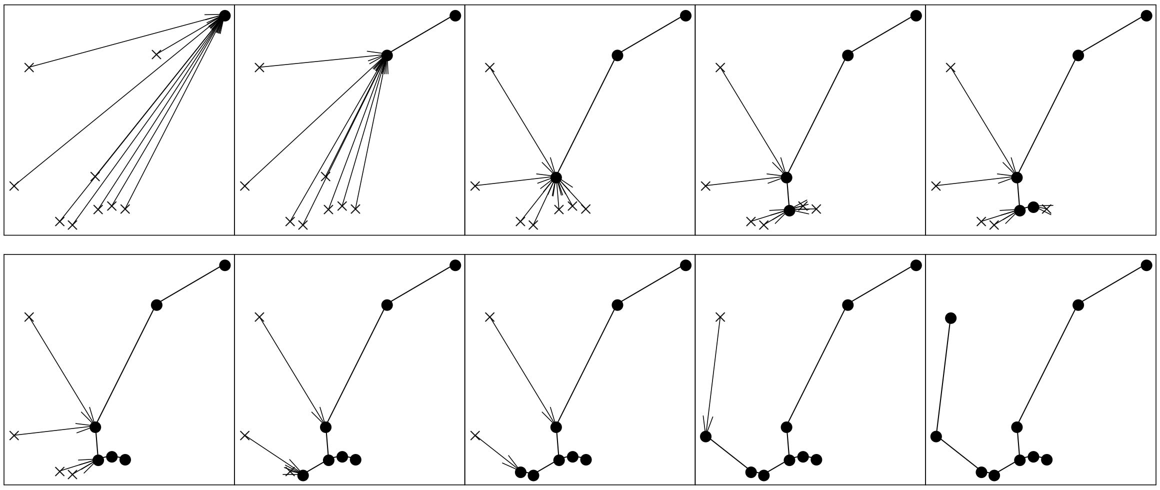
A Graph Algorithm.

Prim's algorithm for computing the minimum spanning tree (MST) of a graph starts with a fragment consisting of a single vertex. It increases the fragment by adding the nearest vertex until all vertices are in the fragment, at which point it is the MST of the entire graph. This picture shows Prim's algorithm on the complete graph induced by a set of 50 planar points; the weight of an edge between two points is defined to be their Euclidean distance.



The snapshots are taken every ten stages.

The obvious implementation of Prim's algorithm on an N-point set requires time proportional to N^3. Dijkstra discovered an elegant implementation of the algorithm with running time proportional to N^2: every point not in the fragment keeps a pointer to its nearest neighbor in the fragment. Here is Dijkstra's implementation on a ten-node planar graph:



In this animation, nodes in the fragment are bullets, nodes not in the fragment are crosses, edges in the MST are `fat` lines, and the nearest neighbors are pointed to by arrows.

These two sequences illustrate two styles of drawing graphs with our system. Dots and lines are sufficient for simple algorithms, while various symbols and line options can depict subtle processes.

The geometric nature of the above graphs made them easy to lay out. Laying out a general graph is very hard. If the graphs in your applications are specialized (such as trees), you might exploit that structure to compute an effective layout. A graph of N vertices can be easily represented by an $N \times N$ matrix in which the i, j-th element represents the edge from vertex i to vertex j; that is useful for insight into some graph algorithms. If you have access to a program that produces good layouts of general graphs, you might use that program to compute positions of vertices, and then feed those into our system.

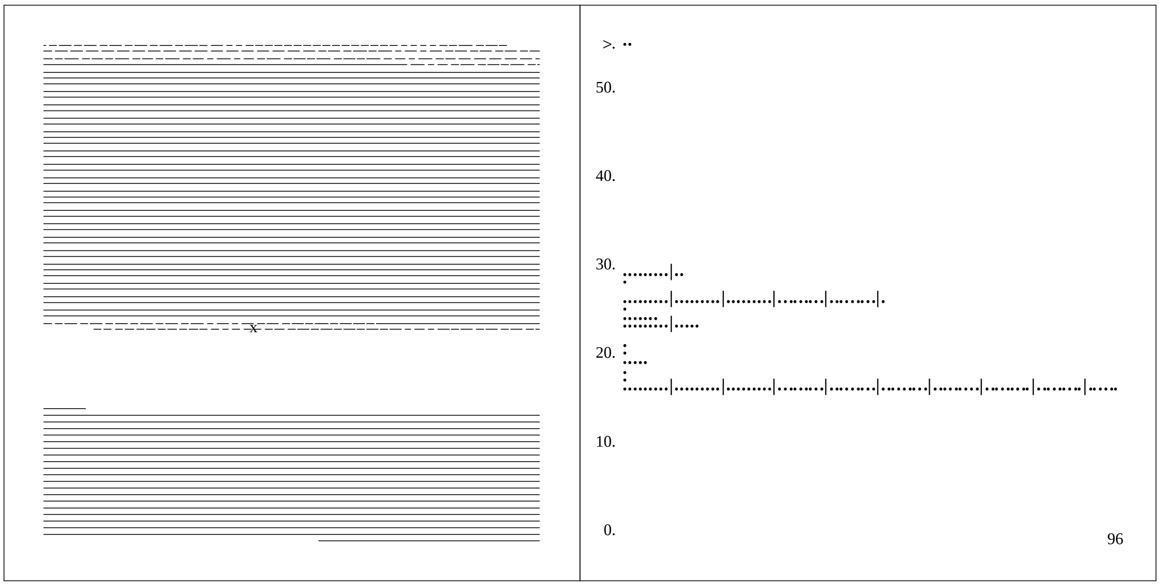
A Memory Allocator.

The *develop* program uses the *malloc* memory allocator for several of its data structures. We augmented *develop*'s calls to the allocator with data gathering routines to produce an animation; here is the final snapshot.
The left frame is the arena of storage from which memory is allocated. Memory blocks are represented by lines, low addresses are at the bottom of the picture, and the picture is 1024 bytes wide. The right frame is a histogram of the sizes of memory currently allocated; there is a dot for each element, a vertical bar every ten positions to help counting, and the maximum value is in the lower right corner.

This snapshot was taken at the end of execution of *fdevelop*. The histogram shows two large blocks of 20,000 and 40,000 bytes, 96 blocks of size 16, and 96 slightly larger blocks (the blocks of size 16 are symbol table records; each points to an allocated string). The arena shows the two huge pieces (the larger is higher) and a gap above the smaller (memory allocated by procedures that didn't call our augmented *malloc*). The remainder of the arena is allocated efficiently.

A movie like this helped us find a bug in *fdevelop*: an early version allocated the symbol table nodes but did not *free* them. This problem manifested itself in an overloaded arena and a huge spike in the histogram at size 16.

The *fdevelop* program interacts with the storage allocator only through the two routines *emalloc* and *efree*. We animated the storage allocator by modifying those routines:




```
int     mallocinit = 0;
FILE    *mallocfp;

char *emalloc(n)
int n;
{       char *p;
        if (mallocinit == 0) {
                mallocinit = 1;
                mallocfp = fopen("/tmp/malloc.hist", "w");
        }
        p = malloc((unsigned) n);
        if (p == NULL) fatal("out of memory");
        fprintf(mallocfp, "m\t%d\t%d\n", (int) p, n);
        return p;
}

efree(p)
char *p;
{       fprintf(mallocfp, "f\t%d\n", (int) p);
        free(p);
}
```

They write on the named file output lines of two types:

`m` *address length*
`f` *address*

The first line denotes that a *malloc* of the given length returned the given address; the second marks a *free*.

The resulting history file contains the interesting events; they are given a geometric interpretation by a subsequent program. Here is a simple program that generates only the arena view from the script file:

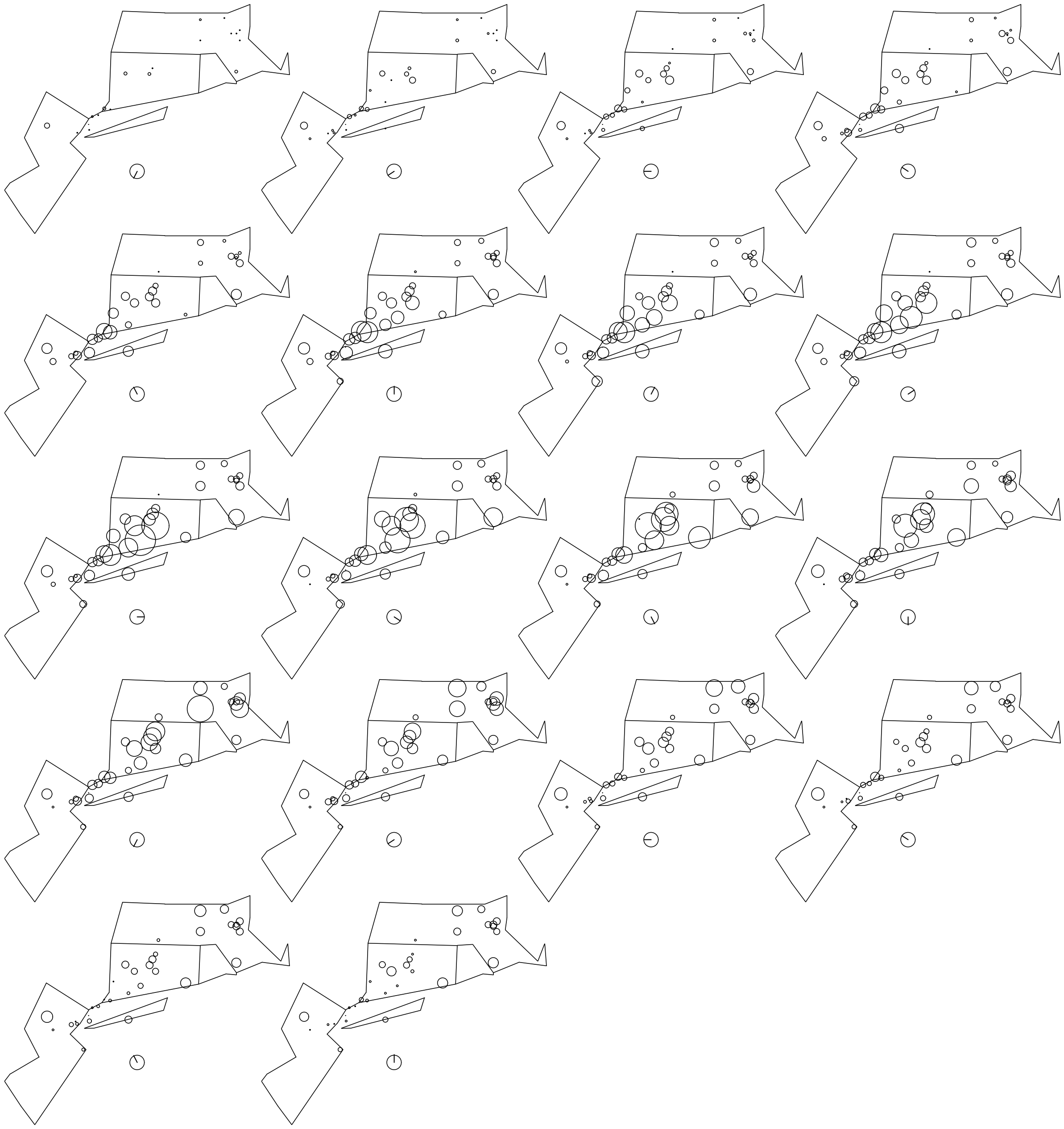
```
awk '
BEGIN          { OFS = "\t"; w = 1024 }
$1 == "m"      { s = $2
                 ehist[s] = e = s + $3
                 sx = s % w; sy = int(s / w)
                 ex = e % w; ey = int(e / w)
                 if (sy == ey) print "a" s ": line", sx, sy, ex, sy
                 else {  print "a" s ": line " sx " " sy " " w " " sy
                         for (i = sy+1; i <= ey-1; i++)
                                 print "b" i ": line", 0, i, w, i
                         print "c" ey ": line", 0, ey, ex, ey
                 }
               }
$1 == "f"      { s = $2
                 e = ehist[s]
                 sx = s % w; sy = int(s / w)
                 ex = e % w; ey = int(e / w)
                 if (sy == ey) print "erase a" s
                 else {  print "erase a" s
                         for (i = sy+1; i <= ey-1; i++)
                                 print "erase b" i
                         print "erase c" ey
                 }
               }
               { print "click call" }
' $*
```

The `BEGIN` block initializes variables, the actions for `m` lines draw memory, and those for `f` lines erase memory. The variable `s` is the starting byte of a block; `e` is the ending block. The variables `sx` and `sy` are the *x* and *y* positions of the starting byte, and similarly for `ex` and `ey`. If the block fits on one line, then only a single line fragment need be drawn; otherwise, fragments of three types are needed.

The complete program for generating scripts from history files is 56 lines of *awk*. To ease tracing the action in the arena, it marks a *malloc* with an `x` and a *free* with an `o`. The histogram view is embellished with bars and the maximum value. There is room for further elaborations; one might, for instance, want to put either or both of the axes of the histogram on logarithmic scales.

Dynamic Statistical Displays.

Rick Becker constructed this display of air pollution in the Northeast United States:



The data was gathered hourly on a single summer day in the early 1970's. The radius of each circle is proportional to the ozone reading (a common measurement of air pollution) at one of 32 stations in New Jersey, New York, Connecticut, and Massachusetts. The radius of the clock denotes the maximum ozone level prescribed by the Environmental Protection Agency; its hour hand goes from 7:00 AM to midnight.

This display shows how New York City's air pollution is blown to the northeast. Becker and his colleagues prepared a similar movie in the early 1970's using the technology of the day; it required several weeks of programming time, then a weekend with a movie camera to make the final film. He built this display from the same data in a couple of hours, then prepared a video tape of the resulting movie in under thirty minutes. The original movie was a bit nicer (it drew the background map and the circles in two different colors), but the new movie is just as useful, and provides stills for free.

7. Living With The System

The system that we have described is the bare bones of an animation environment. We have found that the most fruitful way of enhancing the environment is not by modifying the primary programs, but rather by using small filters that interact with the various files in the system.

We showed earlier, for instance, a race of two sorting algorithms. While other animation systems implement races with a general mechanism for time sharing, we did the job with a small *awk* program that merges two files. Our system does not have a facility for counting clicks; rather, we use filters such as

```
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.

We have built several useful filters in addition to *merge*. The program *view.clicks* prints a summary of the views and clicks used in a script file; it is helpful as one is preparing a *stills* file. Its implementation uses the intermediate file described in Appendix I:

```
develop $1.s
awk '  # view.clicks:  print views and clicks from intermediate file
$1=="d" && $2=="v"    { printf "view %-12s x == [%g,%g], y == [%g,%g]\n",
                                $4 ":", $5, $7, $6, $8
                      }
$1=="d" && $2=="c"    { cstring = cstring "  " $4 }
$1!="d"               { print "clicks:" cstring; exit }
' $1.i
```

The first step of the shell script is to develop the script file; the subsequent *awk* program then reads the resulting intermediate file. The first pattern/action pair prints for each view its name and the *x* and *y* ranges in the script file. The second pair builds a string of all clicks from the define click statements, and the third pair prints that string and exits at the first non-define statement.

The program *show.clicks* takes a script file as input; its output is a new script file containing all information in the input and, in addition, a new view named `click.count` in which the various clicks are counted. This is useful for preparing *stills* files and for debugging. Its input is a script file, either named explicitly or as the standard input. Here is the code:

```
awk ' # show.clicks:  make a new view counting clicks
NR==1 && $1!="view"   { print "view", (oldview = "def.view") }
$1 == "view"          { oldview = $2 }
$1 == "click"         { if ((cname = $2) == "") cname = "def.click"
                        print "view click.count"
                        if ((n=cnum[cname]) == 0) {
                                n = cnum[cname] = ++clicks
                                print "l" n ": text rjust 0", -n, cname ": "
                        }
                        print "v" n ": text ljust 0", -n, "\" " ++ccnt[n]
                        print "last: text ljust 0", -n, "bullet"
                        print "view " oldview
                      }
                      { print $0 }
' $*
```

The second pattern/action pair stores the name of the current view, and the fourth action copies each line onto the output file. The first pattern/action pair supplies a default view name, if needed.

The work is done when the third pattern recognizes a `click` statement. The `if` statement puts the click name in `cname`, and the `print` statement switches to the new view. The second `if` statement is executed when a new `click` statement is seen; it assigns it a number and prints out the click name, once and for all. The two following statements rewrite the appropriate count and place a bullet next to it. The final statement returns to the current view.

One can use the ideas in *show.clicks* to process lines in the script file of the form *name* = *value*

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

Larger filters have also proven useful. For instance, we built a set of tools to render animations of three-dimensional lines and text (circles and rectangles were not supported). The primary program translated a three-dimensional script into a standard script that contained two two-dimensional views for each three-dimensional view; the resulting *movie* and *stills* were suitable for viewing with standard stereo viewers. Support programs included a filter for rotating a view around a given line.

The *movie* and *develop* programs are in fact simple shell scripts that call filter versions named *fmovie* and *fdevelop*. You may find it convenient to rewrite those shell scripts for your environment.

Acknowledgements

We are deeply indebted to Howard Trickey; he gave us invaluable advice for getting a minimal animation facility working in the Sun environment, then finished the job properly. He subsequently made it all work under the X window system. Andrew Hume and Jane Elliott made possible our first experiments with animation. Our early users, Rick Becker and Chris Van Wyk, gave us bug reports and suggestions for improvements. Eric Grosse, John Linderman, Doug McIlroy, Steve Mahaney, Howard Trickey, and Chris Van Wyk made helpful comments on this paper.

References

1. Aho, A.V., Kernighan, B.W., and Weinberger, P.J. *The AWK Programming Language*. Addison-Wesley, Reading, Massachusetts, 1988.
2. Bentley, J. *Programming Pearls*. Addison-Wesley, 1986.
3. Brown, M. and Sedgewick, R. Techniques for Algorithm Animation. *IEEE Software* (January 1985), 28-39.

Appendix I — Intermediate Files

This appendix defines the intermediate files produced by *develop*. The files are easier to process than the corresponding script files. For instance, names are converted to small integers, floating point numbers are scaled to integers in 0..9999, and commands are abbreviated to single letters.

A line whose first non-blank character is # is a comment.

An intermediate file begins with define statements that give the names of the views and the clicks:

`d v` *vnum viewname minx miny maxx maxy*
`d c` *cnum clickname*
`d p` *any text*
`d p e`

Fields are separated by tab characters. Both views and clicks are numbered 0, 1, 2, ... The four final numbers on a view line tell the range of the coordinates in the original script file. A line that begins with `d p` is a ‘‘pragma’’; both *movie* and *stills* currently ignore all such lines. The defines appear at the front of the file in the order views, clicks, pragmas, then the ‘‘end of defines’’ pragma `d p e`.

The geometric commands for lines, boxes, circles and text are mapped to the following:

`g` *slotnum* `l` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `b` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `c` *vnum opts x y rad*
`g` *slotnum* `t` *vnum opts x y text string*

Objects that are never erased have slot number 0; other objects are placed in ‘‘slots’’ that can hold at most one object. The third field is the type of geometric object; the fourth field is the view number. All *x* and *y* values are normalized to integers in the range 0..9999. There is a single separating tab before the (unquoted) text string. Options are given as a string of characters, whose length and interpretation are summarized as:

```
OBJECT    POS       NAME      ABBREV
text      1         center    c
                    ljust     l
                    rjust     r
                    above     a
                    below     b
          2         medium    m
                    small     s
                    big       b
                    bigbig    B
line      1         solid     s
                    fat       f
                    fatfat    F
                    dotted    o
                    dashed    a
          2         -         -
                    ->        >
                    <-        <
                    <->       x
box       1         nofill    n
                    fill      f
circle    1         nofill    n
                    fill      f
```

For instance, the options for a text string are described by two characters giving its position and its size; `small center` text has the option string `cs`. Further options may be added at the right end of the string; subsequent programs should ignore letters they don't expect.

A click statement is represented as

`c` *cnum*

An erase statement is translated into

`e` *line repeated here*, *except* `e` *for leading* `g`

A processor may choose to implement this statement using either the geometric description or the slot number.

A `clear` statement (which erases all objects in a view) is translated into a pair of starting and ending ‘‘blank’’ commands that bracket a sequence of erase statements:

```
b s vnum
b e vnum
```

The erase statements together clear the view. A processor may choose to implement the `clear` either by ignoring the `b` commands and letting the erase statements take their course or by explicitly processing the start command and then ignoring `e` commands until encountering the `b e` line.

As an example, running `develop` on this trivial script file

```
line 1 2 3 4
text small above 5 6 "Hello, world."
click stage
clear
```

produces this intermediate file:

```
d	v	0	def.view	1	2	5	6
d	c	0	stage
d	p	e
g	1	l	0	s-	0	0	4999	4999
g	2	t	0	as	9999	9999	Hello, world.
c	0
b	s	0
e	1	l	0	s-	0	0	4999	4999
e	2	t	0	as	9999	9999	Hello, world.
b	e	0
```

Here is a summary of the commands in the intermediate language:

\# *comment*
`b s` *vnum*
`b e` *vnum*
`c` *cnum*
`d v` *vnum viewname minx miny maxx maxy*
`d c` *cnum clickname*
`d p` *any text*
`d p e`
`e` *line repeated here*, *except* `e` *for leading* `g`
`g` *slotnum* `l` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `b` *vnum opts* x_1 y_1 x_2 y_2
`g` *slotnum* `c` *vnum opts* x y *rad*
`g` *slotnum* `t` *vnum opts* x y *text string*