8½, the Plan 9 Window System

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`rob@research.att.com`

ABSTRACT

The Plan 9 window system, 8½, is a modest-sized program of novel design. It provides ASCII I/O and bitmap graphic services to both local and remote client programs by offering a multiplexed file service to those clients. It serves traditional UNIX files like `/dev/tty` as well as more unusual ones that provide access to the mouse and the raw screen. Bitmap graphics operations are provided by serving a file called `/dev/bitblt` that interprets client messages to perform raster operations. The file service that 8½ offers its clients is identical to that it uses for its own implementation, so it is fundamentally no more than a multiplexer. This architecture has some rewarding symmetries and can be implemented compactly; indeed 8½ is considerably *smaller* than most of its clients.

Introduction

In 1989 I constructed a toy window system from only a few hundred lines of source code using a custom language and an unusual architecture involving concurrent processes [Pike89]. Although that system was rudimentary at best, it demonstrated that window systems are not inherently complicated. Last year, for the new Plan 9 distributed system [Pike90], I applied some of the lessons from that toy project to write, in C, a production-quality window system called 8½. 8½ provides, on black-and-white or grey-scale but not color displays, the services required of a modern window system, including programmability and support for remote graphics. The entire system, including the default program that runs in the window — the equivalent of `xterm` [Far89] with 'cutting and pasting' between windows — is well under 60 kilobytes of text on a Motorola 68020 processor, about half the size of the operating system kernel that supports it and a tenth the size of the X server [Sche86] *without* `xterm`.

What makes 8½ so compact? Much of the saving comes from overall simplicity: 8½ has little graphical fanciness, a concise programming interface, and a simple, fixed user interface. 8½ also makes some decisions by fiat — three-button mouse, overlapping windows, built-in terminal program and window manager, etc. — rather than trying to appeal to all tastes. Although compact, 8½ is not ascetic. It provides the fundamentals and enough extras to make them comfortable to use. The most important contributor to its small size, though, is its overall design as a file server. This structure may be applicable to window systems on traditional UNIX-like operating systems.

The small size of 8½ does not reflect reduced functionality: 8½ provides service roughly equivalent to X windows, other than the lack of support for color terminals. (The provision of that support would not dramatically increase the code size.) 8½'s clients may of course be as complex as they choose, although the tendency to mimic 8½'s design and the clean programming interface means they are not nearly as bloated as X applications.

User's Model

8½ turns the single screen, mouse, and keyboard of the terminal (in Plan 9 terminology) or workstation (in commercial terminology) into an array of independent virtual terminals that may be ASCII terminals supporting a shell and the usual suite of tools or graphical applications using the full power of the bitmap screen and mouse. The entire programming interface is provided through reading and writing files in `/dev`.

Primarily for reasons of history and familiarity, the general model and appearance of 8½ are similar to those of `mux` [Pike88]. The right button has a short menu for controlling window creation, destruction, and placement. When a window is created, it runs the default shell, `rc` [Duff90], with standard input and output directed to the window and accessible through the file `/dev/cons` ('console'), analogous to the `/dev/tty` of UNIX.® The name change represents a break with the past: Plan 9 does not provide a Teletype-style model of terminals. 8½ provides the only way most users ever access Plan 9.

Graphical applications, like ordinary programs, may be run by typing their names to the shell running in a window. This runs the application in the same window; to run the application in a new window one may use an external program, `window`, described below. For graphical applications, the virtual terminal model is extended somewhat to allow programs to perform graphical operations, access the mouse, and perform related functions by reading and writing files with suggestive names such as `/dev/mouse` and `/dev/window` multiplexed per-window much like `/dev/cons`. The implementation and semantics of these files, described below, is central to the structure of 8½.

The default program that runs in a window is familiar to users of Blit terminals [Pike83]. It is very similar to that of `mux` [Pike88], providing mouse-based editing of input and output text, the ability to scroll back to see earlier output, and so on. It also has a new feature, toggled by typing ESC, that enables the user to control when typed characters may be read by the shell or application, instead of (for example) after each newline. This feature makes the window program directly useful for many text-editing tasks such as composing mail messages before sending them.

Plan 9 and 8½

Plan 9 is a distributed system that provides support for UNIX-like applications in an environment built from distinct CPU servers, file servers, and terminals connected by a variety of networks [Pike90]. The terminals are comparable to modest workstations that, once connected to a file server over a medium-bandwidth network such as Ethernet, are self-sufficient computers running a full operating system. Unlike workstations, however, their role is just to provide an affordable multiplexed user interface to the rest of the system: they run the window system and support simple interactive tasks such as text editing. Thus they lie somewhere between workstations and X terminals in design, cost, performance, and function. (The terminals can be used for general computing, but in practice Plan 9 users do their computing on the CPU servers.) The Plan 9 terminal software, including 8½, was developed on a 68020-based machine called a Gnot and has been ported to the NeXTstation, the MIPS Magnum 3000, and the Sun SPARCstation SLC: all small workstations that we use as terminals.

Heavy computations such as compilation, text processing, or scientific calculation are done on the CPU servers, which are connected to the file servers by high-bandwidth networks, typically point-to-point DMA links. For interactive work, these computations can access the terminal that instantiated them. The terminal and CPU server being used by a particular user are connected to the same file server, although over different networks; Plan 9 provides a view of the file server that is independent of location in the network.

The components of Plan 9 are connected by a common protocol based on the sharing of files. All resources in the network are implemented as file servers; programs that wish to access them connect to them over the network and communicate using ordinary file operations. An unusual aspect of Plan 9 is that the *name space* of a process, the set of files that can be accessed by name (for example by an `open` system call) is not global to all processes on a machine; distinct processes may have distinct name spaces. The system provides methods by which processes may change their name spaces, such as the ability to *mount* a service upon an existing directory, making the files of the service visible in the directory. (This is a different operation from its UNIX namesake.) Multiple services may be mounted upon the same directory, allowing the files from multiple services to be accessed in the same directory. Options to the `mount` system call control the order of searching for files in such a *union directory.*

The most obvious example of a network resource is a file server, where permanent files reside. There are a number of unusual services, however, whose design in a different environment would likely not be file-based. Many are described elsewhere [Pike90]; some examples are the representation of processes for debugging, much like Killian's process files for the 8th edition [Kill84], and the implementation of the name/value pairs of the UNIX `exec` environment as files. User processes may also implement a file service and make it available to clients in the network, much like the 'mounted streams' in the 9th Edition [Pres90]. A typical example is a program that interprets an externally-defined file system such as that on a CD-ROM or a standard UNIX system and makes the contents available to Plan 9 programs. This design is used by all distributed applications in Plan 9, including 8½.

8½ serves a set of files in the conventional directory `/dev` with names like `cons`, `mouse`, and `screen`. Clients of 8½ communicate with the window system by reading and writing these files. For example, a client program, such as a shell, can print text by writing its standard output, which is automatically connected to `/dev/cons`, or it may open and write that file explicitly. Unlike files served by a traditional file server, however, the instance of `/dev/cons` served in each window by 8½ is a distinct file; the per-process name spaces of Plan 9 allow 8½ to provide a unique `/dev/cons` to each client. This mechanism is best illustrated by the creation of a new 8½ client.

When 8½ starts, it creates a full-duplex pipe to be the communication medium for the messages that implement the file service it will provide. One end will be shared by all the clients; the other end is held by 8½ to accept requests for I/O. When a user makes a new window using the mouse, 8½ allocates the window data structures and forks a child process. The child's name space, initially shared with the parent, is then duplicated so that changes the child makes to its name space will not affect the parent. The child then attaches its end of the communication pipe, `cfd`, to the directory `/dev` by doing a `mount` system call:

```
mount(cfd, "/dev", MBEFORE, buf)
```

This call attaches the service associated with the file descriptor `cfd` — the client end of the pipe — to the beginning of `/dev` so that the files in the new service take priority over existing files in the directory. This makes the new files `cons`, `mouse`, and so on, available in `/dev` in a way that hides any files with the same names already in place. The argument `buf` is a character string (null in this case), described below.

The client process then closes file descriptors 0, 1, and 2 and opens `/dev/cons` repeatedly to connect the standard input, output, and error files to the window's `/dev/cons`. It then does an `exec` system call to begin executing the shell in the window. This entire sequence, complete with error handling, is 28 lines of C.

The view of these events from 8½'s end of the pipe is a sequence of file protocol messages from the new client generated by the intervening operating system in response to the `mount` and `open` system calls executed by the client. The message generated by the `mount` informs 8½ that a new client has attached to the file service it provides; 8½'s response is a unique identifier kept by the operating system and passed in all messages generated by I/O on the files derived from that `mount`. This identifier is used by 8½ to distinguish the various clients so each sees a unique `/dev/cons`; most servers do not need to make this distinction.

A process unrelated to 8½ may create windows by a variant of this mechanism. When 8½ begins, it uses a Plan 9 service to 'post' the client end of the communication pipe in a public place. A process may open that pipe and `mount` it to attach to the window system, much in the way an X client may connect to a UNIX domain socket to the server bound to the file system. The final argument to `mount` is passed through uninterpreted by the operating system. It provides a way for the client and server to exchange information at the time of the `mount`. 8½ interprets it as the dimensions of the window to be created for the new client. (In the case above, the window has been created by the time the mount occurs, and `buf` carries no information.) When the `mount` returns, the process can open the files of the new window and begin I/O to use it.

Because 8½'s interface is based on files, standard system utilities can be used to control its services. For example, its method of creating windows externally is packaged in a 12-line shell script, called `window`, the core of which is just a `mount` operation that prefixes 8½'s directory to `/dev` and runs a command passed on the argument line:

```
mount -b $'8.5serv' /dev
$* < /dev/cons > /dev/cons >[2] /dev/cons &
```

The `window` program is typically employed by users to create their initial working environment when they boot the system, although it has more general possibilities.

Other basic features of the system fall out naturally from the file-based model. When the user deletes a window, 8½ sends the equivalent of a UNIX signal to the process group — the clients — in the window, removes the window from the screen, and poisons the incoming connections to the files that drive it. If a client ignores the signal and continues to write to the window, it will get I/O errors. If, on the other hand, all the processes in a window exit spontaneously, they will automatically close all connections to the window. 8½ counts references to the window's files; when none are left, it shuts down the window and removes it from the screen. As a different example, when the user hits the DEL key to generate an interrupt, 8½ writes a message to a special file, provided by Plan 9's process control interface, that interrupts all the processes in the window. In all these examples, the implementation works seamlessly across a network.

There are two valuable side effects of implementing a window system by multiplexing `/dev/cons` and other such files. First, the problem of giving a meaningful interpretation to the file `/dev/cons` (`/dev/tty`) in each window is solved automatically. To provide `/dev/cons` is the fundamental job of the window system, rather than just an awkward burden; other systems must often make special and otherwise irrelevant arrangements for `/dev/tty` to behave as expected in a window. Second, any program that can access the server, including a process on a remote machine, can access the files using standard read and write system calls to communicate with the window system, and standard open and close calls to connect to it. Again, no special arrangements need to be made for remote processes to use all the graphics facilities of 8½.

Graphical input

Of course 8½ offers more than ASCII I/O to its clients. The state of the mouse may be discovered by reading the file `/dev/mouse`, which returns a ten-byte message encoding the state of the buttons and the position of the cursor. If the mouse has not moved since the last read of `/dev/mouse`, or if the window associated with the instance of `/dev/mouse` is not the 'input focus', the read blocks.

The format of the message is:

`'m'`
1 byte of button state
4 bytes of x, low byte first
4 bytes of y, low byte first

As in all shared data structures in Plan 9, the order of every byte in the message is defined so all clients can execute the same code to unpack the message into a local data structure.

For keyboard input, clients can read `/dev/cons` or, if they need character-at-a-time input, `/dev/rcons` ('raw console'). There is no explicit event mechanism to help clients that need to read from multiple sources. Instead, a small (336 line) external support library can be used. It attaches a process to the various blocking input sources — mouse, keyboard, and perhaps a third user-provided file descriptor — and funnels their input into a single pipe from which may be read (three types of) events in the traditional style. This package is a compromise. As discussed in a previous paper [Pike89] I prefer to free applications from event-based programming. Unfortunately, though, I see no easy way to achieve this in single-threaded C programs, and am unwilling to require all programmers to master concurrent programming. It should be noted, though, that even this compromise results in a small and easily understood interface. An example program that uses it is given near the end of the paper.

Graphical output

The file `/dev/screen` may be read by any client to recover the contents of the entire screen, such as for printing (see Figure 1). Similarly, `/dev/window` holds the contents of the current window. These are read-only files.

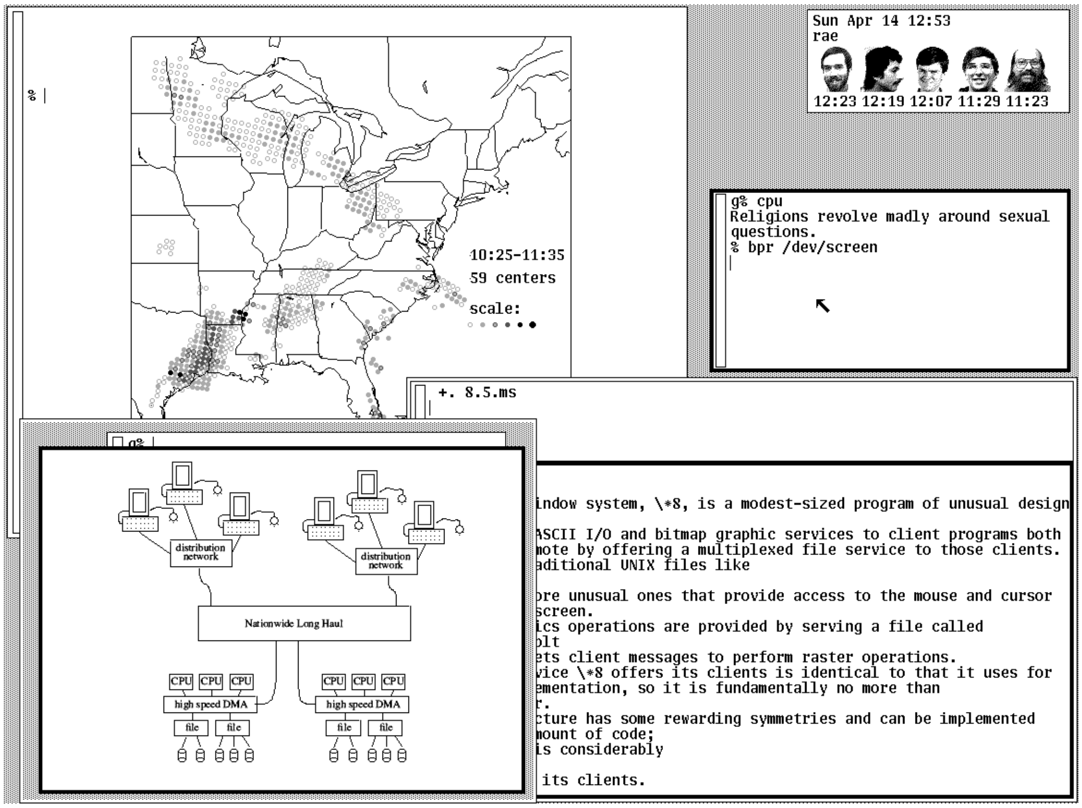

Figure 1. A representative 8½ screen, running on a NeXTstation under Plan 9 (with no NeXT software). In the upper right, a program announces the arrival of mail. In the upper left is a map displayed by the CPU server that shows the radar reflectance of precipitation in the eastern United States. In the lower right there is a screen editor, `sam` [Pike87], and in the lower left an 8½ running recursively and, inside that instantiation, a previewer for `troff` output. Underneath the faces is a small window running the command that prints the screen by passing `/dev/screen` to the bitmap printing utility.

To perform graphics operations in their windows, client programs access `/dev/bitblt`. It implements a protocol that encodes bitmap graphics operations. Most of the messages in the protocol (there are 14 messages in all) are transmissions (via a write) from the client to the window system to perform a graphical operation such as a `bitblt` [PLR85] or character-drawing operation; a few include return information (recovered via a read) to the client. As with `/dev/mouse`, the `/dev/bitblt` protocol is in a defined byte order. Here, for example, is the layout of the `bitblt` message:

`'b'`
2 bytes of destination id
2x4 bytes of destination point
2 bytes of source id
4x4 bytes of source rectangle
2 bytes of boolean function code

The message is trivially constructed from the `bitblt` subroutine in the library, defined as

```
void bitblt(Bitmap *dst, Point dp, Bitmap *src, Rectangle sr, Fcode c).
```

The ‘id’ fields in the message belie another property of 8½: the clients do not store the actual data for any of their bitmaps locally. Instead, the protocol provides a message to allocate a bitmap, to be stored in the server, and returns to the client an integer identifier, much like a UNIX file descriptor, to be used in operations on that bitmap. Bitmap number 0 is conventionally the client’s window, analogous to standard input for file I/O. In fact, no bitmap graphics operations are executed in the client at all; they are all performed on its behalf by the server. Again, using the standard remote file operations in Plan 9, this permits remote machines having no graphics capability, such as the CPU server, to run graphics applications. Analogous features of the original Andrew window system [Gos86] and of X [Sche86] require more complex mechanisms.

Nor does 8½ itself operate directly on bitmaps. Instead, it calls another server to do its graphics operations for it, using an identical protocol. The operating system for the Plan 9 terminals contains an internal server that implements that protocol, exactly as does 8½, but for a single client. That server stores the actual bytes for the bitmaps and implements the fundamental bitmap graphics operations. Thus the environment in which 8½ runs has exactly the structure it provides for its clients; 8½ reproduces the environment for its clients, multiplexing the interface to keep the clients separate.

This idea of multiplexing by simulation is applicable to more than window systems, of course, and has some side effects. Since 8½ simulates its own environment for its clients, it may run in one of its own windows (see Figure 1). A useful and common application of this technique is to connect a window to a remote machine, such as a CPU server, and run the window system there so that each subwindow is automatically on the remote machine. It is also a handy way to debug a new version of the window system or to create an environment with, for example, a different default font.

Implementation

To provide graphics to its clients, 8½ mostly just multiplexes and passes through to its own server the clients’ requests, occasionally rearranging the messages to maintain the fiction that the clients have unique screens (windows). To manage the overlapping windows it uses the layers model, which is handled by a separate library [Pike83a]. Thus it has little work to do and is a fairly simple program; it is dominated by a couple of modest-sized switch statements to interpret the bitmap and file server protocols. The built-in window program and its associated menus and text-management support are responsible for most of the code.

The operating system’s server is also compact; the version for the 68020 processor, excluding the implementation of a half dozen bitmap graphics operations, is 1153 lines of C; the graphics operations are another 1118 lines.

8½ is structured as a set of communicating coroutines, much as discussed in a 1989 paper [Pike89]. One coroutine manages the mouse, another the keyboard, and another is instantiated to manage the state of each window and associated client. When no coroutine wishes to run, 8½ reads the next file I/O request from its clients, which arrive serially on the full-duplex communication pipe. Thus 8½ is entirely synchronous.

The program source is small and compiles in about 10 seconds in our Plan 9 environment. There are ten source files and one `makefile` totaling 3860 lines. This includes the source for the window management process, the cut-and-paste terminal program, the window/file server itself, and a small coroutine library (`proc.c`). It does not include the layer library (another 610 lines) or the library to handle the cutting and pasting of text displayed in a window (898 lines), or the general graphics support library that manages all the non-drawing aspects of graphics — arithmetic on points and rectangles, memory management, error handling, clipping, — plus events and non-primitive drawing operations such as circles and ellipses (a final 1797 lines). Not all the pieces of these libraries are used by 8½ itself; a large part of the graphics library in particular is used only by clients. Thus it is somewhat unfair to 8½ just to sum these numbers, including the 2271 lines of support in the kernel, and arrive at a total implementation size of 9436 lines of source to implement all of 8½ from the lowest levels to the highest. But that number gives a fair measure of the complexity of the overall system.

The implementation is also efficient. 8½'s performance is competitive to X windows'. Compared using Dunwoody's and Linton's gbench benchmarks on the 68020, distributed with the ''X Test Suite'', circles and arcs are drawn about half as fast in 8½ as in X11 release 4 compiled with gcc for equivalent hardware, probably because they are currently implemented in a user library by calls to the point primitive. Line drawing speed is about equal between the two systems. Text is drawn about 20% faster in 8½ and the bitblt test is about four times faster. These numbers vary enough to caution against drawing sweeping conclusions, but they suggest that 8½'s architecture does not penalize its performance. Finally, 8½ boots in under a second and creates a new window apparently instantaneously.

An example

Here is a complete program that runs under 8½. It prints the string "hello world" wherever the left mouse button is depressed, and exits when the right mouse button is depressed. It also prints the string in the center of its window, and maintains that string when the window is resized.

```
#include <u.h>
#include <libc.h>
#include <libg.h>

void
ereshaped(Rectangle r)
{
        Point p;

        screen.r = r;
        bitblt(&screen, screen.r.min, &screen, r, Zero); /* clear window */
        p.x = screen.r.min.x + Dx(screen.r)/2;
        p.y = screen.r.min.y + Dy(screen.r)/2;
        p = sub(p, div(strsize(font, "hello world"), 2));
        string(&screen, p, font, "hello world", S);
}

main(void)
{
        Mouse m;

        binit(0, 0);    /* initialize graphics library */
        einit(Emouse);  /* initialize event library */
        ereshaped(screen.r);
        for(;;){
                m = emouse();
                if(m.buttons & RIGHTB)
                        break;
                if(m.buttons & LEFTB){
                        string(&screen, m.xy, font, "hello world", S);
                        /* wait for release of button */
                        do; while(emouse().buttons & LEFTB);
                }
        }
}
```

The complete loaded binary is a little over 18K bytes on a 68020. This program should be compared to the similar ones in the excellent paper by Rosenthal [Rose88]. (The current program does more: it also employs the mouse.) The clumsiest part is ereshaped, a function with a known name that is called from the event library whenever the window is reshaped or moved, as is discovered inelegantly but adequately by a special case of a mouse message. (Simple so-called expose events are not events at all in 8½; the layer library takes care of them transparently.) The lesson of this program, in deference to Rosenthal, is that if the window system is cleanly designed a toolkit should be unnecessary for simple tasks.

Status

8½ is in regular daily use by almost all the 60 people in our research center. Some of those people use it to access Plan 9 itself; others use it as a front end to remote UNIX systems, much as one would use an X terminal. It has been fairly stable for almost a year.

Some things about 8½ may change. It would be nice if its capabilities were more easily accessible from the shell. A companion to this paper [Pike91] proposes one way to do this, but that does not include any graphics functionality. Perhaps an ASCII version of the `/dev/bitblt` file is a way to proceed; that would allow, for example, `awk` programs to draw graphs directly. Finally, 8½ offers no 'iconization' of its clients, and probably never will. But it would be helpful for it to provide some quick mechanism for managing a cluttered screen, and some ideas are being considered. By the time this paper appears, 8½ will probably address this issue. The code is unlikely to grow substantially as a result, however, and clients will be unaffected by the changes.

Can this style of window system be built on other operating systems? A major part of the design of 8½ depends on its structure as a file server. In principle this could be done for any system that supports user processes that serve files, such as any system running NFS or AFS [Sun89, Kaza87]. One requirement, however, is 8½'s need to respond to its clients' requests out of order: if one client reads `/dev/cons` in a window with no characters to be read, other clients should be able to perform I/O in their windows, or even the same window. Another constraint is that the 8½ files are like devices, and must not be cached by the client. NFS cannot honor these requirements; AFS may be able to. Of course, other interprocess communication mechanisms such as sockets could be used as a basis for a window system. One may even argue that X's model fits into this overall scheme. It may prove easy and worthwhile to write a small 8½-like system for commercial UNIX systems to demonstrate that its merits can be won in systems other than Plan 9.

Conclusion

In conclusion, 8½ uses an unusual architecture in concert with the file-oriented interprocess communication of Plan 9 to provide network-based interactive graphics to client programs. It demonstrates that even production-quality window systems are not inherently large or complicated and may be simple to use and to program.

Acknowledgements

Helpful comments on early drafts of this paper were made by Doug Blewett, Stu Feldman, Chris Fraser, Brian Kernighan, Dennis Ritchie, and Phil Winterbottom. Many of the ideas leading to 8½ were tried out in earlier, sometimes less successful, programs. I would like to thank those users who suffered through some of my previous 7½ window systems.

References

[Duff90] Tom Duff, ''Rc - A Shell for Plan 9 and UNIX systems'', Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33

[Far89] Far too many people, XTERM(1), Massachusetts Institute of Technology, 1989

[Gos86] James Gosling and David Rosenthal, ''A window manager for bitmapped displays and UNIX'', in Methodology of Window Management, edited by F.R.A. Hopgood et al., Springer, 1986

[Kaza87] Mike Kazar, ''Synchronization and Caching issues in the Andrew File System'', Tech. Rept. CMU-ITC-058, Information Technology Center, Carnegie Mellon University, June, 1987

[Kill84] Tom Killian, ''Processes as Files'', USENIX Summer Conf. Proc., Salt Lake City June, 1984

[Pike83] Rob Pike, ''The Blit: A Multiplexed Graphics Terminal'', Bell Labs Tech. J., V63, #8, part 2, pp. 1607-1631

[Pike83a] Rob Pike, ''Graphics in Overlapping Bitmap Layers'', Trans. on Graph., Vol 2, #2, 135-160, reprinted in Proc. SIGGRAPH '83, pp. 331-356

[Pike87] Rob Pike, ''The Text Editor `sam`'', Softw. - Prac. and Exp., Nov 1987, Vol 17 #11, pp. 813-845

[Pike88] Rob Pike, ''Window Systems Should Be Transparent'', Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296

[Pike89] Rob Pike, ‘‘A Concurrent Window System’’, Comp. Sys., Spring 1989, Vol 2 #2, pp. 133-153

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Plan 9 from Bell Labs’’, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

[Pike91] Rob Pike, ‘‘A Minimalist Global User Interface’’, USENIX Summer Conf. Proc., Nashville, June, 1991, this volume

[PLR85] Rob Pike, Bart Locanthi and John Reiser, ‘‘Hardware/Software Tradeoffs for Bitmap Graphics on the Blit’’, Softw. - Prac. and Exp., Feb 1985, Vol 15 #2, pp. 131-152

[Pres90] David L. Presotto and Dennis M. Ritchie, ‘‘Interprocess Communication in the Ninth Edition Unix System’’, Softw. - Prac. and Exp., June 1990, Vol 20 #S1, pp. S1/3-S1/17

[Rose88] David Rosenthal, ‘‘A Simple X11 Client Program -or- How hard can it really be to write ‘‘Hello, World’’?’’, USENIX Winter Conf. Proc., Dallas, Jan, 1988, pp. 229-242

[Sche86] Robert W. Scheifler and Jim Gettys, ‘‘The X Window System’’, ACM Trans. on Graph., Vol 5 #2, pp. 79-109

[Sun89] Sun Microsystems, NFS: Network file system protocol specification, RFC 1094, Network Information Center, SRI International, March, 1989.