User-Level Window Managers for UNIX

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

ABSTRACT

Wm manages a collection of windows on a display terminal. Each window has its own shell or other interactive program, running in parallel with those in the other windows. This permits a user to conduct several interactions with the system in parallel, each in its own window. The user can move from one window to another, re-position a window, or create or delete a window at any time without losing his or her place in any of the windows. Windows can overlap or completely obscure one another; obscured windows can be "lifted" up and placed on top of the other windows.

This paper describes how such a window manager for UNIX† is implemented as a set of user processes, without modifications to the UNIX kernel. It shows how the simple, but well-chosen facilities provided by the original (Version 6) UNIX kernel are sufficient to support *wm*. In addition, subsequent versions of *wm* exploit features of the kernel introduced into newer versions of UNIX to provide faster and more sophisticated window operations, still implemented entirely at the user level.

Introduction

This paper describes the design of a display window manager for UNIX implemented entirely as a set of user processes, without modifications to the UNIX kernel. It shows how the simple facilities provided by the original (Version 6) UNIX kernel are sufficient to support such a window manager. In addition, more recent versions of the window manager exploit features of the kernel introduced into newer versions of UNIX to provide faster and more sophisticated operations in windows, still implemented entirely outside the kernel.

This window manager, *wm*, provides a UNIX user the ability to conduct several interactions in parallel, each in a different window on a text display terminal. The windows may be created, moved, and temporarily or permanently erased at any time. They may also overlap or completely obscure one another, and such hidden or partially hidden windows may be "lifted" and placed on top of the other windows as desired. Figure 1 shows a snapshot of a *wm* session in progress.

User Interface

The notion of organizing computer data spatially was propounded and exploited by Nicholas Negroponte in the Spatial Data Management System [2, 3]. In *wm*, however, spatial cues are used only to specify a context for a dialogue. Once a window is selected, further interactions within that window make use of the power and abstraction of more conventional user interface techniques. Teitelman [8] made good use of display screen windows for a collection of parallel interactions with an INTERLISP system. More recently, several personal computers and workstations have adopted this window-oriented style of dialogue as their principal mode of interaction. Other systems similar to the present one have also been provided under UNIX [4, 7, 9].

---

†UNIX is a trademark of Bell Laboratories.

Traditional user interfaces for computers that handle parallel processes place all inputs and outputs in one chronological stream, identifying the process associated with each, but interleaving the data. The Berkeley job control facilities for UNIX provide a first attempt at improving this situation [5].

By contrast, a window-based user interface enables a user to manage a collection of dialogues by associating a spatial location with each dialogue, in much the same way one organizes a desk. On a desk, all input papers on all topics are not (one hopes) placed on a single pile in chronological order, but rather they are divided into piles by topic. When input for a particular topic is received, the corresponding pile is located, lifted, and placed on top of other papers, and the necessary work is done on that pile. Each topic may thus be associated with and remembered in terms of a location on the desk. Recent empirical evidence showed that such a window-oriented user interface induced better user performance than a more traditional scrolled message interface in a particular situation involving several parallel interactions [6].

Wm conducts several concurrent dialogues with a UNIX user. Each takes the form of a UNIX shell, to which UNIX commands can be given and from which other interactive programs can be initiated. Each dialogue is conducted in a separate area of the screen or *window* designated by the user. At any moment, one of the windows is considered the current input window, and all keyboard inputs (except for *wm* commands themselves) are sent to the shell or program associated with that window. At any time (including in the middle of typing a command in a window), the designation of the current window may be changed and a different dialogue begun or resumed. Outputs resulting from these dialogues will appear in their appropriate windows as they are generated, regardless of which window is the current input window. Output destined for a portion of a window that is obscured by another window will appear whenever that portion of the window is uncovered. Windows can be "piled" on one another in any sequence.

Wm was originally designed for use in an intelligent terminal that could communicate with several computers simultaneously. Each dialogue with a different computer was associated with a window. The method is equally applicable to a collection of dialogues, all with the same computer but on different topics. Still, any or all of the present windows can run programs to conduct interactive dialogues with other computers (such as *telnet*).

Design for "Vanilla" UNIX

To implement a system of this sort, it is necessary for one user process to be able to manage a collection of other user processes and to mediate all of their inputs and outputs. For the inputs, it must act as a switch, directing input from the keyboard to different programs in response to user commands. For the program outputs, it must place the output of each program in its correct position on the screen.

If adequate primitives for creating and manipulating processes and for catching their inputs and outputs are provided by the operating system, a window manager can be built entirely as a user program. The original design of UNIX, with its *pipe* and *fork* mechanisms provides a user the right primitives to design such a system in an elegant fashion without kernel modifications.

Wm initiates and manages its own collection of UNIX processes, including those run in response to entered commands. Any conventional UNIX program can be used from *wm*, provided it does not make significant assumptions about the nature of its input and output devices−that is, it should treat input and output from a pipe as equivalent to input and output from a terminal or other source.

Wm runs as *2n+2* parallel UNIX processes of four different types (where *n* is the number of windows in use). The division into processes is dictated by the fact that the original UNIX *read* call on an empty pipe or input device causes a process to block until input becomes available. Hence there is a separate process for each pipe or device that must be read asynchronously. Each such process contains a loop that reads from its particular pipe or device, processes the input, and then waits for more. Figure 2 shows the processes and pipes that comprise *wm*.

The **main** process reads from and waits for input from the keyboard. Input consisting of text is sent to the shell process associated with the current window and also to the **scrn** process, described below, for echoing. Input consisting of *wm* commands is interpreted by **main**, translated into one or more primitive commands plus arguments, and sent to **scrn** for execution. Simple changes in the input command language are thus localized in **main**. To change the name, input syntax, or prompt for a command, only the code in **main** need be modified. Since input commands are reduced to a somewhat more general set of primitive

commands, some simple new commands may be implemented entirely in **main** as aliases for specific uses or combinations of the existing primitive commands.

The **scrn** process handles all outputs to the screen. All processes that want to affect the screen must thus place requests to do so on a common pipe. **Scrn** reads these instructions from the pipe and makes appropriate modifications to the screen. *Wm* commands that affect the screen layout, such as moving a window, are placed on this pipe by **main** and handled by **scrn**. Output text characters from the individual shell processes that belong in a window are also placed on this pipe along with a window identifier and a bit indicating whether the character should be displayed immediately or just remembered for the next time the display is refreshed. **Scrn** then compares the desired configuration of the screen to a buffer containing the actual configuration and transmits the necessary updates.†

There is a **shell** process associated with each window. This is simply the standard UNIX *sh* (or any other designated program). These **shell** processes have no direct access to the terminal, but run as captives of *wm*, connected by pipes, so that their inputs and outputs can be mediated. The input to each of these processes is a pipe from **main**, since **main** knows which window is the current input window and can place the typed input text on the pipe to the corresponding **shell** process. All outputs of the **shell** processes must be sent to **scrn** to be displayed, but they must first be tagged with the name of the window in which they belong.

To do this, each window has a **shmon** process that monitors the output of the corresponding **shell** process. The output of each **shell** process is a pipe to a corresponding **shmon** process. Each time output appears on that pipe, **shmon** reads it, packages it with a header identifying its window, and then places it on the common request pipe to **scrn**.

Wm comprises about 1000 lines of C code−about 500 each for the **main** and **scrn** processes and less than 50 for the **shmon** process.

Remarks and Problems with "Vanilla" UNIX

Each window in *wm* emulates an individual glass teletype. Inputs appear in the bottom and scroll off the top of a window. Since the standard input and output for all programs run by *wm* are really pipes, all programs run under *wm* should treat their inputs and outputs simply as streams of characters, without distinctions between terminals and pipes. The fact that UNIX and most of its original programs permit a pipe to be substituted for a terminal input or output stream is an elegant aspect of UNIX that is crucial to *wm*. This obtains for most UNIX programs; they perform individual "building-block" functions and are thus intended to be equally usable individually from the terminal or as filters connected to other programs to perform more complex tasks. Programs that try to determine whether they have access to a real terminal may behave differently or even refuse to run with *wm*. For example, *stty* is meaningless when applied to a pipe rather than a terminal, *vi* will refuse to run from a pipe, and *csh* will not allow job control if it cannot access the terminal. (However, note that *wm* is really an alternate approach to controlling concurrent jobs.)

A very rudimentary facility for supporting a whole-screen-oriented program is provided. It creates a special temporary window, creates a *termcap* description of a "terminal" that occupies only the corresponding area of the actual screen, and then provides that description and direct access to the terminal to the screen-oriented program until the latter exits.

Since *wm* operates with the terminal in raw mode, it must provide for itself the input line editing functions normally provided by the teletype driver.

Because of the architecture of *wm*, there are no pipes that connect one window to another, hence there is no explicit facility for communication between windows. It can be achieved, however, through the file system. A program in one window can append to a file while one in another window continuously tries to read from the end of that file.

---

†The update algorithm is less sophisticated than the optimization performed in the *curses* package [1], but *curses* was not available in Version 6 UNIX. This update algorithm is also somewhat easier to adapt to unusual terminals, as seen below.

Terminal Dependencies

While the newer version of *wm* uses *curses* to perform all terminal-dependent operations in a terminal-independent fashion, terminal dependencies can be isolated fairly easily even without *curses*. All terminal-dependent code in the original *wm* is restricted to a collection of five simple procedures. They were originally written separately for each type of terminal, but have also been written in terms of the terminal-independent interface, *termcap*, for systems that have it.

The five procedures perform the following tasks:

ttyinit	Performs any necessary initialization for the terminal.
ttyclose	Performs any necessary closing for the terminal before *wm* exits or suspends.
ttymov	Moves the terminal cursor to a given row and column.
clearscreen	Clears the terminal screen.
clearline	Clears from the cursor to end of the current line (not mandatory).

For each of several common terminals, the definitions of these procedures comprise about 15 lines of code altogether.

This approach isolates terminal dependencies sufficiently that *wm* can also be adapted for use on graphic displays by replacing the above procedures and making other minor changes. Such a version of *wm* has been written to produce output suitable for the standard UNIX plot filters (plus some added commands for raster graphic displays) and used with a Genisco frame buffer. Windows may be in various colors and may use different fonts for their text.

Design for Version 4.2 UNIX

Berkeley Version 4.2 VAX UNIX provides new features that make it possible to improve *wm* significantly. By using pseudo-terminals instead of pipes for interprocess communication, several of the problems discussed above disappear. In addition, the synchronous input/output multiplexing feature of the new UNIX makes the former division of *wm* into processes as dictated by the blocking read unnecessary. A revised version of *wm*, then, solves many of the earlier problems and runs in a single process (plus the user's shells). It is, however, less interesting and certainly less portable than the initial version. Again, the facilities are provided entirely in user-level processes, without the need for kernel modifications.

This version of *wm* reads from the keyboard and also from the pseudo-terminals associated with each window, in a round-robin, using the multiplexed read call *(select)*. Keyboard input consisting of text is sent to the pseudo-terminal associated with the current window. The pseudo-terminal driver itself handles echoing (when enabled) and intraline editing, obviating the need for *wm* to duplicate these functions. Keyboard input consisting of *wm* commands is processed directly; text input is sent to the appropriate pseudo-terminal. Output from the pseudo-terminals is read by *wm*, interpreted in terms of the cursor control commands of a simple virtual terminal defined by *wm*, and then added to the appropriate screen window for processing by the *curses* package [1].

This version of *wm* comprises about 1000 lines of C code, all in a single process. Figure 3 shows the architecture of the program.

Remarks and Problems with Version 4.2 UNIX

Since each window is implemented with a pseudo-terminal, the fact that a program is running in a window rather than on a real terminal is transparent to most programs. Specifically, most screen editors and games may be used, and *stty* may be called to change characteristics such as echoing or line editing individually for each window. For example, note that one of the windows in Figure 1 is running *vi*, which has adjusted itself to the window size. Some programs, however, assume that their output devices are of some minimum size; they will not run well in very small windows. Also, programs that attempt to manipulate the controlling terminals of process groups will not work properly under *wm*. For this reason, *csh* cannot currently be run in the individual windows instead of *sh*.

It is generally not possible to move a window while an interactive program (other than a shell) is running in it. First, this is necessary because, whenever a window is moved, *wm* sends a shell command to

change the *TERMCAP* variable for the shell in that window, to describe its new size. A more fundamental reason is that the *curses* library routines (sensibly) do not expect the terminal description to change while a program is running, and so make no provision for checking for or adapting to such changes.

Since pseudo-terminals are a system-wide resource and are usually fixed in number, the total number of windows that can be in use by all users at any one time is limited to the number of pseudo-terminals made available to *wm*.

A facility for communicating between windows is now easy to provide. Since each window uses a pseudo-terminal, any data sent to its slave pseudo-terminal will appear in the window; and pseudo-terminals are in the name space of the UNIX file system and thus available to other processes. To simplify use of this feature, when a window is created and a pseudo-terminal obtained for it, a link to the name of its slave pseudo-terminal is created in the user's current directory. Any program inside or outside *wm* can then write to or read from that file name without prearrangement.

Program Versions

These programs are written in C for use with UNIX. There are three principal versions: **wm.v6**, **wm.v7**, and **wm.v42**. The first, as described above, runs under unmodified Version 6 UNIX on a PDP-11. The code for this version was frozen and abandoned several years ago, but it is still available. **Wm.v7** runs under Version 7 UNIX, and the same code also runs on Berkeley 2.8 and also on a VAX on Berkeley 4.1 and 4.2. No changes in the source code are required between the PDP-11 and VAX, except that constants for the maximum number and size of windows are limited by the available memory on a PDP-11. This version is similar in design to **wm.v6**, which was described above, but has a number of improvements. The newest version, **wm.v42**, runs only under Berkeley 4.2 on a VAX, as described in this paper. It uses the *select* synchronous input/output multiplexing call, which is unique to 4.2, and also other features that are found in some, but not all, versions of UNIX, such as pseudo-terminals and *curses*. At this writing, this version is not yet thoroughly tested on 4.2. An intermediate version for use with Versions 2.8 or 4.1 can also be constructed by adapting some of the features of **wm.v42** to **wm.v7**. For example, the use of *curses* can certainly be adapted to 2.8; pseudo-terminals are available on some versions of 4.1; and some versions of 4.1 can also simulate a non-blocking read on a pseudo-terminal or a short time-out.

Availability

Three versions of *wm* are available to interested researchers.

wm.v6 For Version 6 UNIX.

wm.v7 For Version 7 UNIX, also runs on Berkeley 2.8, 4.1, and 4.2.

wm.v42 For Berkeley 4.2 UNIX only (but has some features than can be retrofitted to **wm.v7**).

The code can be obtained over the Arpanet by sending a request to jacob@nrl-css. The author can also be reached via uucp at ...!decvax!linus!nrl-css!jacob.

Conclusions

It is demonstrably feasible to provide a useful and efficient display window management facility in UNIX at the user level, without support from kernel modifications. Such a facility can even be provided for the original Version 6 UNIX, although some improvements are obtainable by exploiting features provided by more recent versions of UNIX.

Acknowledgments

I would like to thank Mark Cornwell, Rudy Krutar, Alan Parker, and Mark Weiser for helpful discussions of this work.

References

1. K. Arnold, "Screen Updating and Cursor Movement Optimization," University of California, Berkeley (1980).

2. R. Bolt, “Spatial Data Management,” Technical Report, Architecture Machine Group, Massachusetts Institute of Technology (1979).

3. C.F. Herot, R. Carling, M. Friedell, and D. Kramlich, “A Prototype Spatial Data Management System,” *Computer Graphics* **14**(3), pp. 63-70 (1980).

4. M. Horton, personal communication (September 8, 1982).

5. W. Joy, “An Introduction to the C Shell,” University of California, Berkeley (November 1980).

6. S. Murrel, “Computer Communication System Design Affects Group Decision Making,” *Proc. Human Factors in Computer Systems Conference*, pp. 63-67 (1983).

7. R. Pike, “Graphics in Overlapping Bitmap Layers,” *ACM Transactions on Graphics* **2**(2) (1983).

8. W. Teitelman, “A Display Oriented Programmer’s Assistant,” *International Journal of Man-Machine Studies* **11**, pp. 157-187 (1979).

9. M. Weiser, C. Torek, R. Trigg, and R. Wood, “The Maryland Window System,” Technical Report 1271, Computer Science Department, University of Maryland (1983).