Cite as:
Greenberg, S., Peterson, M., and Witten, I. H. (1986). Issues and experiences in the design of a window management system. In Proceedings of the Canadian Information Processing Society Edmonton Conference, pp. 33-44, Edmonton, Alberta, October 21-23. Earlier version as Report 86-240-14.

ISSUES AND EXPERIENCES IN THE DESIGN OF A WINDOW MANAGEMENT SYSTEM

Saul Greenberg
Murray Peterson
Ian Witten

Man/Machine Systems Laboratory
Department of Computer Science
The University of Calgary
2500 University Drive NW
Calgary, Canada T2N 1N4

Abstract — Window systems underly many successful human-computer interfaces. But constructing them involves several fundamental design issues which are resolved in various ways by different implementations. These affect both user's and programmer's interface, and include tiling *vs* overlapping windows; program *vs* user control of windows; techniques for coping with changeable window sizes; design of transient windows; low-level *vs* high-level tools for the programmer; local *vs* distributed control; and the role of a controlling window manager process. There is no broad consensus among designers as to how best to make such decisions.

This paper illuminates the issues involved by surveying characteristics of window systems and discussing dominant design questions. The philosophy and implementation underlying the JADE window manager constructed at the University of Calgary is revealed, and the design is reevaluated from the perspective of three years' experience of using the system and programming within it.

Overview

It seems to be widely accepted that "modern" human-computer interfaces should often be built around the concept of windows. But what are the characteristics of a window system? How should it appear to the user? Who should manipulate windows — the user, the application, or both? And what about the programmer's interface? Should his access to the functional components be low-level (which increases programming complexity) or high-level (which usually limits flexibility)? And who should control window management, the system or the programmer?

This paper surveys design issues surrounding window systems, and explains the philosophy behind a particular implementation. The implementation is then reexamined in the light of experience. In order to set the stage before plunging into design details, we first review general characteristics of window systems and the reasons for basing modern human-computer interfaces on them. Next we raise questions and concerns about problems which all designers of window systems must address. There are two kinds of user — "end" users of application software which is built within the window system, and "programmers" who create such software; these are discussed in separate sections. Regrettably, it is not possible to insulate design questions completely from implementation of a window system — for one thing, design compromises must be made because execution speed is of paramount importance in many window operations. The subsequent section descibes the design philosophy of the JADE window manager, constructed at the University of Calgary as part of an environment for developing distributed systems. Finally, our decisions are reevaluated from a three-year perspective of using and programming within the system.

Introducing windows

Traditional interactive interfaces to computers are highly sequential. At least by default (and sometimes by decree), conventional command languages complete the current activity before the next can begin. Hierarchical menu structures make matters worse. Before a command can be invoked, the user must navigate from his present location in the hierarchy (the previous activity) to the desired target (the next activity). Despite this imposed sequentiality, parallelism arises naturally and pervasively in interactive computer dialogues. Documentation must be consulted, references checked, subsidiary programs run, and so on. Current activity may be unexpectedly preempted by more urgent tasks, only to be resumed later. A user generally has to map his internal conceptions, in terms of interleaved parallel activities, into a sequential stream of tasks to be presented to the computer (Bannon *et al*, 1983).

The need to sequentialize activity imposes enormous strains on the user's short-term and working memory.

> Short-term memory is used in conjunction with working memory for processing information and problem solving. ... If many facts and decisions are necessary to solve a problem, then short-term

> and working memory may become overloaded. ... [These memories] are highly volatile; disruptions cause loss of memory, and delays can require that the memory be refreshed.
>
> Shneiderman, 1984

At least as currently implemented, sequential dialogues make it difficult to store intermediate results. For example, a programmer requiring documentation of a language construct must leave the editor, invoke the online manual, find and read the required information, and finally reenter the editor. At this point he must recall his original location within the program, remember the retrieved information, and apply it. Obviously a high load is placed on short-term and working memory, and the almost inevitable failures cause frustration and inefficiency. No wonder hard copy is so popular!

Computer support for parallel activity is a more realistic framework for helping the user with his thought processes. In most interactive settings, one needs to view or interact with many different contexts simultaneously, sometimes suspending them temporarily to deal with more urgent interruptions. Admittedly, many command languages permit primitive parallelism. For example, in the UNIX *csh* (Joy, 1979), tasks can be suspended interactively and resumed on demand. But this provides strictly limited benefits, for just one viewport must serve all processes. In contrast, window interfaces are specifically designed to map an indefinite number of views on to a single physical screen. If views are regarded as virtual input/output devices, one can then communicate with different asynchronous processes in separate windows.

Quite a different use of windows, which is becoming common in text editors, is to provide multiple viewports into one or more documents. More generally, windows can separate different modalities of communication such as command input, data input, system state, and data output; provide semipermanent accessories such as clocks, calendars, calculators; simulate buttons, keyboards, or potentiometers for input. They can be opened, moved around, and closed, to reflect the demands of the task. Like a conventional VDU, a window system constitutes an outer shell within which other specialized user interfaces can operate.

Windows provide a ready-made metaphor for manipulating objects on the screen, such as text, bookmarks and processes. It seems more "natural" to transfer a chunk of text from one place to another by literally moving it between windows on the screen than by editing it out of the source document, saving it somewhere, reinvoking the editor, retrieving it, and placing it in the target. (Imagine, for instance, copying an address from an address list in one window to a letterhead in another.) It seems more natural to save one's place in a document by opening a separate window for a temporary or not-so-temporary excursion elsewhere, leaving the main context manifest, than by placing an invisible marker there and having to return to it by name. It seems more natural to reactivate a suspended process by pointing at it than by using a symbolic identifier as argument to a wakeup command.

The popularity of window systems is increasing rapidly because people like to use them. The Xerox Star (Smith *et al*, 1982) and Apple Macintosh (Williams, 1984) are popular systems aimed at office and home markets. Window systems are marketed by leading home computer software houses (witness for example Microsoft's "Windows," VisiCorp's "VisiON"). Innumerable systems have been developed in research laboratories to support or supplement general programming environments; well-known examples include Sun and Apollo workstations; Xerox Dandelion, Dolphin, and Dorado; and Lisp machines from a variety of manufacturers. Packages such as the Maryland system (Wood, 1982) and NUNIX (Test, 1982) offer the programmer high-level subroutines for window manipulation within the UNIX operating environment. Several non-graphical window systems are also available, including the CMU Network Window Manager (Gosling & Rosenthal, 1983) and the Waterloo Port user interface (Malcolm & Dyment, 1983). Some editors for ASCII terminals provide their own support for windows (eg *emacs*; Stallman, 1981).

But windows also increase complexity for the user. He must now physically manipulate the windows and keep track of which does what. Operations on windows generally involve a pointing device (such as a mouse), both to indicate where to place windows on the screen and to identify which one is to be manipulated. Typical operations include creating a window, binding it to a particular process, moving and resizing it, activating and destroying it, and shuffling the order of windows in a pile.

Similarly, complexity is increased for the programmer, as he can no longer consider input and output as indivisible primitives provided by the operating system. User actions such as keystrokes, pointing, menu-selection or window manipulation can occur at any time and must be anticipated explicitly by the program. Output complexity is increased too: the window may change size and position; it may be completely or partially occluded by others; or the application may have to keep a coordinated set of windows updated.

Despite their growing popularity, there has been surprisingly little human factors research on the use of windows. Although there have been some publications which address technical questions of implementation (eg Pike, 1983; Rosenthal, 1982), fundamental design decisions such as whether windows should overlap, or whether only the user should be capable of creating them, must be taken in the absence of hard experimental evidence one way or the other. (The few evaluations that have been published are reviewed below.) Moreover, the proprietary nature of most implementations makes experiments difficult to perform and alternatives difficult to evaluate.

Design issues from the user's viewpoint

Window design is not simple. Seemingly small decisions may profoundly affect how the overall system appears to the user, and dictate the style of programming within the interface. Several important issues central to all window-based software are outlined below. The questions raised do not have easy answers, and their resolution will depend critically upon application requirements, hardware and software resources, and the creativity of the designer. The choice depends too on the use to which the window is put. Possible uses include viewing greater amounts of information; accessing and combing multiple information sources; independently controlling multiple programs; using reminders to see what is available; working within special contexts; and viewing the same information in different ways (Card *et al*, 1984). Given

this variety of user tasks,

> ... attempts to understand the merits of competing window designs must be clear about the function for which the windows are being used.
>
> Card *et al*, 1984

Overlapping versus tiling of windows. Many existing systems allow windows to overlap; others do not. Figure 1 shows a representative screen with overlapped windows from the Apple Macintosh, while Figure 2, from the *emacs* text editor, is non-overlapping or "tiled". (Depending on the implementation, one can tile horizontally, vertically, or both.) Which to implement is a crucial and far-reaching design decision. Overlaps create considerable technical complications because they require the overlapped portion to be saved somewhere. In general there may be multiple overlaps, so that a tree structure of saved areas is needed. At some extra cost in storage, it may be easier to save complete windows instead of obscured fragments. However, then it takes longer to redraw when the window pile is altered, and redrawing time is usually a critical resource. There is also the question of what happens to output to a fully or partially covered window. If it is blocked, the user loses much of the attraction of running jobs in the background. If it is active, implementation complexity is increased and the system may expend considerable resources updating screen images which will never be viewed.

One of the few works evaluating tiled and overlapping windows indicates, not surprisingly, that the choice of technique depends on the situation (Bly and Rosenberg, 1986). Overlapped windows are preferred when much window manipulation is required, for example when a user requires maximum visibility of the contents of one or several windows. Tiled windows are appropriate for those situations requiring little manipulation, for example when a user requires a balance of visibility for all windows. In spite of experimental results confirming the expectations above, Bly and Rosenberg (1986) note that most of their subjects preferred overlapping windows. Indeed, overlaps seem to be fashionable rather than demonstrably beneficial. Shneiderman *et al* (1985) report user complaints that overlapped windows are confusing — hidden windows can be forgotten and an inordinate amount of time spent shuffling and redrawing the screen. Some users have been observed to expend considerable energy in making their overlap displays resemble the tiling illustrated in Figure 2. On the other hand, users also experience frustration with tiled windows, particularly when the screen is small. Each new window causes others to be reduced in size, obscuring information at least as readily as overlapping. Much effort may be consumed in adjusting window sizes to approximate an optimal fit, only to be ruined when a new window is opened — a ready soruce of frustration. Although automatic sizing of windows based upon constraints may alleviate the problem somewhat, techniques for doing so are still in the research stage (Cohen *et al*, 1985).

The choice between overlapping and tiling may depend on the relationship between screen size and expected number of simultaneously visible windows. Large screens with a handful of related windows may be best tiled, whereas overlapping minimizes the inevitable limitations of a small screen. One compromise is to have a number of small physical screens, each supporting a single or tiled view (Shneiderman *et al*, 1985). Also, careful design may alleviate some of the problems which beset overlapping windows. The system may supply a default strategy for tiling which is overridden manually only rarely (Cohen *et al*, 1985). Or perhaps a fuller understanding of the ways people use windows will provide a firmer foundation to window system design, minimizing the user's overhead and feelings of frustration (Norman *et al*, 1985; Card *et al*, 1984).

Program versus user control of windows. Who has the power to create, position, and destroy windows? This is a very important design consideration. At one extreme the user may be given absolute control over his screen, so that he — and he alone — can manipulate windows. Then a process which needs a window must ask for one to be created and assigned to it. This runs the risk of causing the user considerable bother in doing simple things. For each process, the user must open the window, place it on the screen and size it. When the process completes its actions, the user is left with a "dead" window which must then be destroyed.

At the other extreme, the user may have no direct control over creating, placing, or destroying windows. A program which wants a window opens one; perhaps the window manager at least attempts to place it sensibly on the screen. Unfortunately this can be very disruptive, disturbing the user's context and breaking his train of thought. Indeed, windows may be created whose function (and the appropriate response) is a mystery to the user.

Many compromises are possible between these two extremes. One simple solution allows both program and user to manipulate windows. But this raises the spectre of conflict between the programmer's and the user's desire. The user may feel, for instance, that a large overhead is continually involved in repositioning windows that are ruined by insensitive automatic placement. As another example, while a user may destroy any window in the INTERLISP environment, certain windows will insist on recreating themselves (Teitelman and Masinter, 1981). Another scheme which seems attractive is to allow each program to command the window manager to create sub-windows within any areas of the screen that are allocated to it. At its simplest, this could involve splitting windows into non-overlapping "panes". However, for the sake of uniformity it may be preferable if the same window format and overlap possibilities are available to a program within its window as the user has within his screen. This raises the issue of whether a user is allowed to move a program-created window outside its parent window, and if so doing changes its status as belonging to the program that created it (for purposes of enlargement, deletion, and so on).

Panning and auto-formatting. An important difference from the conventional VDU environment is that programs executed within a system with user-controlled windows do not know the size of their output device until run time (and even then it may change!). This causes difficulty because the best output format may vary substantially with radically different page sizes (for example, handheld computers use their very small displays in quite a different fashion from normal-screen personal computers). Generally, an application program will either allow any window size with no guarantee that it will be usable, or impose a range of size restrictions on windows within which it can run effectively (even if this violates a policy of user control over windows).

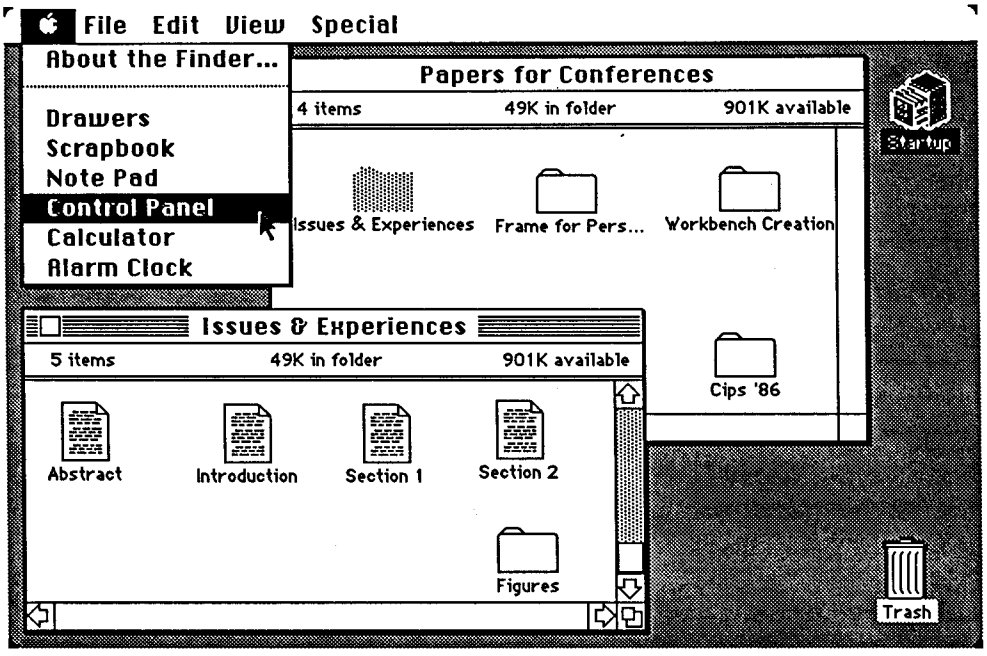


Figure 1: Overlapped Windows and a Pull-Down Menu — The Apple Macintosh

```
* VTERM - vaxb /dev/ttyp4
.ce
.ps+4
.ce
Issues and Experiences in the Design
.sp
.ce
N P B E T X W Y * 1title=1title 12:56pm 3.69[6] (Normal)  Top
.sh "\fBAbstract\fR"
\(em Window systems underly many successful human-computer interfaces.
But constructing them involves several fundamental design issues which are
resolved in various ways by different implementations.
These affect both user's and programmer's interface, and include
tiling \fIvs\fR overlapping windows;
program \fIvs\fR user control of windows;
.ce
N P B E T X W Y * 2abstract=2abstract 12:56pm 3.69[6] (Normal)  24%

                    ISSUES AND EXPERIENCES IN THE DESIGN
                       OF A WINDOW MANAGEMENT SYSTEM

                                Saul Greenberg
                              Murray Peterson
                                 Ian Witten


Abstract -- Window systems underly many successful human-computer interfaces.
But constructing them involves several fundamental design issues which are
resolved in various ways by different implementations.  These affect both
user's and programmer's interface, and include tiling vs overlapping windows;
program vs user control of windows; techniques for coping with changeable
window sizes; design of transient windows; low-level vs high-level tools for
N P B E T X W Y * roffed-doc=[None] 12:55pm 3.01[7] (Normal)  13%

Yes Sweetie: ls
0header          3introduction    5.2issues        9bib             cover.letter
1title           4motivation      5issues          IansDraft        roffed-doc.CKP
2abstract        5.1issues        6jade            Makefile         working.notes
Yes Sweetie:


N P B E T X W Y * shell=[None] 12:55pm 3.01[7] (Normal)  Bottom
Search for: Abstract
```

Figure 2: Tiled Windows — The Emacs Editor

Many window managers allow the user to pan or "scroll" across and down all text and graphics windows. (In others, this kind of facility must be implemented at application level, creating considerable headaches for the programmer.) But viewing your text through a peephole makes it difficult to scan easily. An alternative is to wrap lines that exceed the window boundary; but breaking at arbitrary points within words compromises readability. A better, although computation-intensive, plan is to reformat all text dynamically into the current window size. This implies that programs should output structured document descriptions instead of plain text (see, for example, Witten, 1985).

Other techniques are more radical. Real-time zooming, via continuously-variable scaling, is attractive but somewhat impractical for text with current display technology. The "bifocal display" of Spence and Apperley (1982) is an interesting alternative. Here, a few items of a data base are shown in a central region of the screen in detail, while two outer "demagniified" regions contain symbols representing the contents of the data base. Moving a symbol into the center expands upon it. More general are "fisheye" views, which present a balance of local detail and global context in a single window (Furnas, 1986). However, these techniques depend on what is being displayed and cannot be used with any kind of data. For example, fisheye views require each item of information to be annotated with a "degree of interest".

Transient windows. A primary concern of window systems is to allow convenient, dynamic, management of screen "real estate" — a most valuable resource. Although normal windows remain on the screen until explicitly removed by user or program, temporary popup windows are invaluable for displaying transient information, for they disappear immediately after use. These windows are particularly appropriate for menus, to clarify options through simple prompts, to alert the user about the system status, or to show help information.

Pop-up (or pull-down) menus, for example, are typically invoked by pressing a button on the mouse or by pointing at a menu label, and persist until the button is released. The mouse can be moved around the screen and used to select an item from the menu. In this way, consistent access to normally hidden functions is available at any time. Most systems offer a hierarchical arrangement of transient menus, of which three variants are shown in Figures 3a, 3b and 3c. The first takes the form of an overlapping stack; obscured ones come to the front when the mouse moves to them. The second design illustrates a button menu, in which the first level of the hierarchy is always visible as a menu bar. Pointing to the button raises the menu next to the symbol. An ever-present set of buttons creates a feeling of familiarity and consistency for the user at the cost of screen area. The Macintosh pull-down menu (Figure 1) is a variant of this — buttons are ordered in a bar at the top of the window. Figure 3c shows a walking menu. Moving the mouse to the right of an entry causes a sub-menu to appear. This has the advantage that the path through the hierarchy is always visible. Pull-down and stacked menus are illustrated within a window system in Figures 1 and 4.

Along with instantly-available actions invoked by pointing at objects on the screen, users need corresponding instantly-available help messages to explain the effect of actions or the meaning of objects. Transient windows, invoked perhaps by a dedicated help button, are ideal for this. Since the display of the message will not affect the appearance of the application, a generic help facility can be installed at all levels of the interface. When linked to menus, for example, they can identify an operation better than the short, sometimes cryptic, name of an item. Figure 4 shows an example.

Transient windows may also result from a program action, and remain in view until the user acknowledges the program's request. For example, status messages may be relegated to a transient alert box rather than dedicating an area of the screen to it. Simple queries are also migrating towards transient windows in the form of dialogue boxes which display a message and (perhaps) a set of possible responses (Figure 3d). Transient forms allow rapid selection and clarification of system options (Figure 3e).

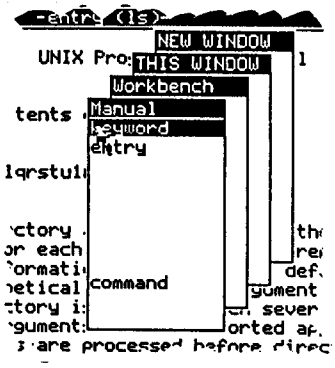
However, there is no consensus as to how transient windows should appear. There are almost as many variants of popup menu formats as there are window managers. Hierarchical menu arrangements generate even more variety, such as walking menus, menu bars and stacks of menus. Property sheets, which use both visual controls and a form-based dialogue, are gaining popularity for managing attributes of windows and objects within applications. Yet in spite of the plethora of techniques available, there is almost no research which evaluates and contrasts their effectiveness within an interface.

Design issues from the programmer's viewpoint

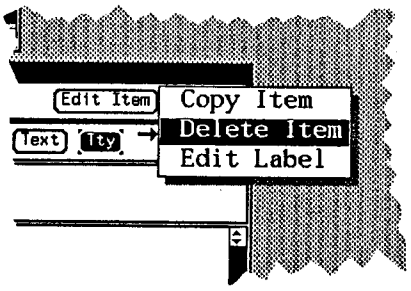
Some window systems are embedded in turnkey applications, and the issue of writing programs within them simply does not arise (eg Xerox Star). In the others, programming applications within the window system is a completely different level of activity from using windows, and is usually quite difficult. Some systems provide windows as a more-or-less integral part of the programming environment (eg Lisp or Smalltalk machines); in others, facilities are grafted on to a foreign (eg UNIX-based) host; still others provide programmers with access to the routines used to create a turnkey end-user interface (eg Macintosh Toolbox; Espinosa and Hoffman, 1983).

Multitasking. The fundamental motive behind windows is the desire to support concurrent, parallel contexts on a single physical screen. Additionally, the system must somehow permit all the generic window manipulation actions — creation, destruction, movement and sizing — as well as menu selection. All these occur as asynchronous events. Consequently a fundamental issue in window system design is whether multitasking is the responsibility of the programmer or the system.

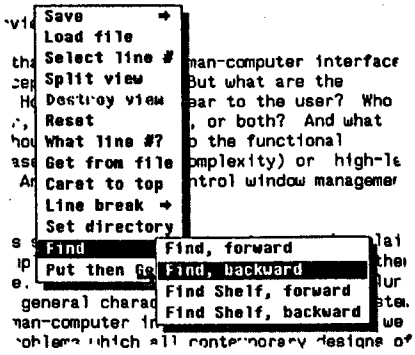
If the base operating system does not support multitasking, each individual application program must handle asynchronous events itself. This introduces many problems. First, the programming environment becomes extremely complex, for applications must cater for all possible sequences



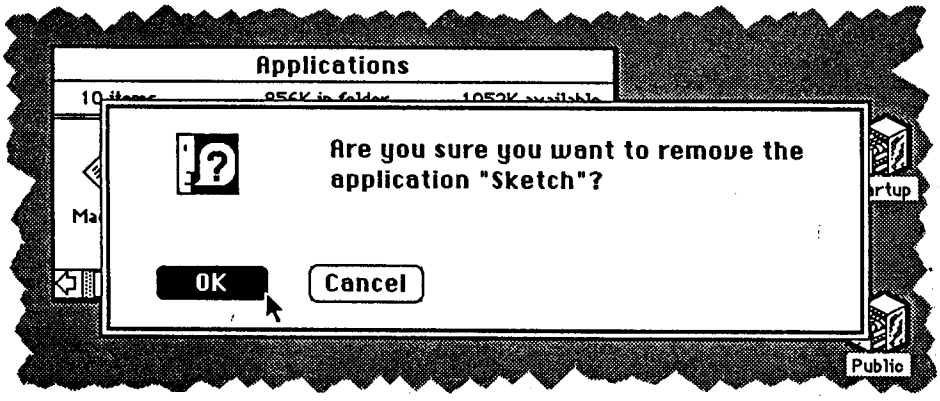
3a: Stacked Menu



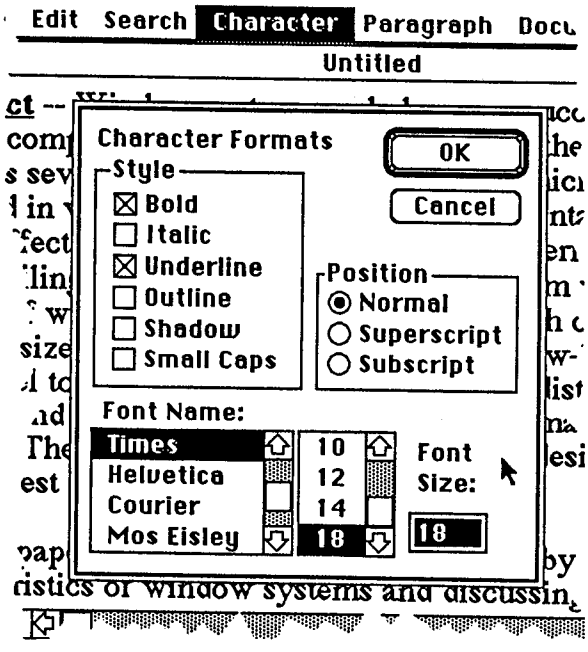
3b: Button Menu



3c: Walking Menu



3d: Dialogue Box



3e: Property Sheet

3a: JADE's stacked menu is raised by pressing a dedicated menu button. Moving the mouse to a hidden menu brings it immediately to the surface.

3b: The SUN 3's button menu is raised by pointing to the always visible button (in this case "Edit Item") and pressing a mouse button.

3c: The SUN 3's walking menu is raised by pressing a dedicated mouse button. Pointing the arrow right of the entry raises the auxiliary menu.

3d: The Macintosh dialogue box asks the user to select one of a set of choices

3e: This Macintosh property sheet, raised from a menu selection, describes the properties of a character font. Starting at the upper right and moving clockwise, it includes: checkboxes ("Character Formats"), normal buttons ("Ok" and "Cancel"); mutually exclusive buttons in a panel ("Position"); a text field ("Font Size"); and two scrollable menus ("Font Name").

Figure 3: Transient Windows.

of events, and do so in real time. Obviously a compute-bound program cannot respond satisfactorily to a popup menu request, unless an explicit check for menu activity is made periodically in the program. Second, as multitasking is limited to only those tasks expressly catered to by the application, the user will not enjoy system-wide concurrency.

The Apple Macintosh is a good example of a window-based system lacking a full multi-tasking kernel. A user may not have two windows on the screen running different applications. For a single application to make use of multiple windows, it must simulate multi-tasking within itself. The program must be aware of all possible states and must handle explicitly any occurrence in any window. In effect, each programmer must write his own (albeit restricted) multi-tasking kernel.

Desk-accessories add another level of complexity to the Macintosh. Within any application, the user can invoke his favourite accessories, such as calendars, notepads, clocks, calculators, and so on. When finished, they disappear and the user resumes his primary task. Yet it is the application program (and thus the programmer at considerable bother to himself) who is responsible for making the accessory available, and for activating it when selected by the user. As Webber (1986) states, "every program on the Mac becomes a frantic conspirator in the multitasking cover-up".

If multitasking is supported, the programming chore is greatly reduced. Concurrent actions are handled by creating a corresponding set of processes. Processes responsible for real-time interactions, such as displaying popup menus, are given higher priority. Although fully supported multitasking makes application programs easier to write, it introduces the problem of physical screen management and synchronization between many processes which (in theory) have little or no knowledge of each other. For example, some mechanism must prevent programs from writing on areas of the screen which are "owned" (even temporarily) by another process. If a popup menu is currently obscuring part of a window, all output to that window should be either stopped completely, or at least prevented from spoiling the menu. Fortunately, this conflict between multitasking and screen integrity can be solved in several ways.

Screen arbitration. One method of screen arbitration is to create a single window manager process which owns the entire physical bit map. All window operations, including redirection of mouse and keyboard input, take place through requests to this process. A message-passing protocol is suitable for transferring requests between the application and window manager processes, and has the further advantage of allowing application processes to be distributed across multiple computers (Neal *et al*, 1984). But the design of such a manager is complex — diverse, asynchronous requests are not handled comfortably by a single process. The monolithic window manager cannot take advantage of the multitasking facilities offered by the base operating system. Instead, it is now responsible for simulating a multitasking kernel inside itself.

Alternatively, process synchronizing primitives such as semaphores or monitors can be used for screen arbitration. Then the window manager becomes a set of multiple processes, each handling some small task (eg control of the pointing device). All requests for screen access, by either the window processes or the application, are sequentialized through a semaphore or monitor. However, a serious problem occurs when a process which has gained ownership of screen resources dies unexpectedly. The operating system must exercise considerable ingenuity in identifying these resources and returning them to the global pool. Also, application processes cannot reside on a different computer unless they make requests through some server process on the local machine.

Low- versus high-level functionality. Programmers of window systems are difficult to please, for they usually seek high-level window management facilities as well as low-level flexibility. At one extreme, if the window manager provides very basic primitives like *set-pixel*, programmers have the opportunity to do exactly as they wish — subject to the costs of programming time and execution speed. At the other extreme, high-level libraries, although minimizing programming complexity, impose a commitment to a particular user interface style.

The major advantage of libraries is that most of the difficult work has already been done before-hand, work which may be beyond the skills and budget of the average developer. These libraries incorporate a philosophy of what a user interface should look like and how it should be implemented. This is especially important for integrated environments, where different applications have similar interface styles. From a limited set of pre-defined interface fragments written by experts in human-machine interfacing, the programmer chooses the kind of interaction he wishes.

One good example of such a library is the Apple Macintosh toolbox. The programmer's manual begins by describing the philosophy of the user interface, and expects the developer to follow that philosophy (Espinosa and Hoffman, 1983). The toolbox provides the means for implementing the Macintosh view of the interface. This library includes scrollbars, limited text structures and editing packages, dialogue and alert boxes, different window configurations, and pull-down menus. Using these tools is relatively straightforward. With a minimal amount of work, one can construct (supposedly) a clean menu-based interface to an application†. Of course, you have to be happy with the Macintosh view of the world. If, for example, you would rather use a popup menu, you are out of luck.

Although this is widely accepted as good practice, one could argue that these design decisions should be the responsibility of the application programmer rather than the window system implementor. An obvious solution is to have the window manager offer very low-level primitives and build increasingly higher-level software on top. Then the programmer can select the layer appropriate for the application in mind. Unfortunately, this scheme does not

†Unfortunately, programming the Macintosh is not as easy as one would expect. Although individual toolbox routines are reasonable, programming is complex because: a) the toolbox provides little assistance for event-handling; and b) the machine does not have a multitasking kernel. The system simply does not supply good support for programming within the interface philosophy it insists on! See Webber (1986) for a good discussion of these deficiencies and how they are overcome by the AMIGA user-interface software.

work in practice. The lowest level primitives (such as *set-pixel*) are much too slow, for each request involves (at minimum) a subroutine call and a semaphore or message-passing operation. True layering is impossible, as higher level routines must be built into the window manager if it is to have acceptable performance. Consider, for example, a line-drawing function. A high-speed Bresenham line algorithm should manipulate the window bitmap directly, rather than use the slow *set-pixel* primitive. Layering and flexibility are sacrificed for the sake of efficiency.

The choice of levels depends on the desired flexibility, execution speed, programming ease, and enforcement of consistency by the interface. Inevitably (for speed) the window manager will incorporate some high-level primitives that cannot be easily changed. There are no correct decisions in this area; one can only strive to achieve an acceptable trade-off for any particular system.

Lessons from the JADE window manager

The goal of the JADE project at the University of Calgary was to build a programming environment for prototyping and implementing distributed computer systems. Early in 1983 it was resolved to design and implement a small multitasking operating system kernel and window manager on a bare 68000 workstation. A simple message-passing protocol forms the core of all JADE software, and provides a natural means of communicating with the workstations from a central cluster of large UNIX systems. The JADE Window Manager (JWM) is the result.

Figure 4 shows a picture of a typical JADE screen, with a number of windows (some overlapping), and a stack of popup menus and help windows. There are five labeled windows, not counting the transient ones. The first, second and third are respectively: a virtual terminal into the UNIX system, a scrollable file viewer; and a small analog clock. Windows four and five are "workbenches", a locally developed application that provides a popup menu-based interface to UNIX (Greenberg and Witten, 1985). The fifth window, which is a workbench to the UNIX on-line manual, overlaps two others. By pressing the "menu" button on the mouse in this window, the user has popped up a stack of menus, of which one item ("keyword") is now pointed to. Pressing a "help" button has caused a stack of help windows to appear, where the top one contains a brief description of the "keyword" menu entry.

Let us review the major design decisions, why we made them at the time, and how they feel now after three year's daily use. We have been able to accommodate a sizeable user community (one of the advantages of adopting bare-bones microcomputers was that many could be purchased within a tight budget), comprising both end users of a window-based interface to UNIX and software developed for JADE, and programmers creating such software. It will be convenient to follow closely the structure of the last two sections in considering the design choices that were made.

1. Windows are overlapped, with buried ones inactivated. As Figure 4 illustrates, we opted for an overlapping window system; a decision which was in fact taken quite lightly. Much more controversial was the fact that output to windows which are covered by others (even partially) is blocked and the relevant processes suspended. For example, windows one and two in the Figure cannot accept output because they are partially obscured by another window†. This decision involved considerable — and, in retrospect, unnecessary — agonizing.

Deactivation of buried windows was a reasonable choice. It simplifies implementation very considerably, and users never saw it as a problem. The overlap issue was not as clear cut. In practice, most users rarely have more than three or four windows on their screen at any time, often arranged as tiles when feasible. In retrospect, tiling should have been supported in addition to overlaps either at the screen or window level.

One scheme that uses both tiles and overlaps allows a hierarchy of windows. The top-level windows may be overlapped. Each of those windows may be further subdivided into tiled sub-windows supporting specific capabilities, such as: virtual terminals; scrollable areas; panels of buttons and switches; forms; and so on. (A good example of this type is the Suntools package on the Sun workstation.) Another reasonable scheme tiles windows into a single frame. Although it is possible to overlap and to change the size of windows, the default layout of a set of related windows is two-dimensional tiling with an effective balance of sizes. (See, for example, the *document examiner* described in Symbolics, 1985).

2. Users have total control over windows, programs virtually none. It was decided early that users would control all aspects of window manipulation, including creation, destruction, movement and shaping. Programs would be denied this kind of control. This was because it was deemed essential that users feel in control at all times. Some in our group who had experienced excessive program control of windows in other environments (such as INTERLISP on the Xerox Lisp machines) are content with this scheme. Most others think that this degree of user control of window manipulation is excessive — it can make the user feel a slave to the system! However, all agree that lack of program control produces a very consistent user interface.

Most application programmers despise their programs' impotence to control windows. Particularly annoying is the inability to fix a window's size, and to destroy windows — the user must be relied on for all cleanup. Eventually, the JWM architect bowed to pressure and included a facility for program control of windows.

In retrospect, users seem to pay an excessive penalty for the privilege of being the only one who is allowed to manipulate windows. We would now recommend building extensive program control of windows into the system's primitives. However, we would also supplement this with a standard set of system defaults and guidelines to encourage sensitive use and consistency across the interface.

3. The window manager supports neither panning nor auto-formatting. With the exception of a few low-level primitives such as raster scrolling and simple

† If the clock were covered and therefore stopped, one gets an amusing effect on uncovering it. It runs fast to catch up, because the update events have been queued meanwhile.

clipping, each application has to perform output control and resizing unaided. Although the window system is graphics oriented, in fact textually-oriented applications dominate. But JWM's conception of text is no more sophisticated than that of a standard VDU. It has no text formatting capabilities and does not accommodate multiple fonts.

This lack of help with what application programmers considered vital functions is probably one of the greatest weaknesses of JWM. Many users implemented their own text panning and scrolling packages; some created font handlers. The wheel was reinvented many times and, as each version was different, interface consistency suffered. More emphasis should have been placed in the original specification on text editing and scrolling primitives. Panning within graphics and text is highly desirable for most applications, whereas autoformatting is useful only within specialized textual contexts. A few users would have liked zooming capabilities, even for text. In retrospect, we spent too much of our limited resources on graphics.

4. Stacked popup menus with associated help windows are fully supported. A popup menu capability with realtime response was considered essential for a consistent user interface. Popup menus take the form of an overlapping stack (Figure 4). A small help window is associated with each popup menu, and a stack of these corresponding to the popup menu stack is always instantly available. The stack also includes help specific to the window itself. Two of the three mouse buttons are dedicated to popup menus and help windows respectively, leaving only one for application programs.

Heavy use is made of popup menus, and they are simple to program and use. The menu layout is quite acceptable and compares favourably with others. One cardinal advantage of the menu stack is that application programs only add menus to the stack, so that the basic ones (for window creation etc) are always accessible at the bottom of the stack. This creates a comfortable feeling of familiarity and control. Set against this is the fact that stacks can grow large and become

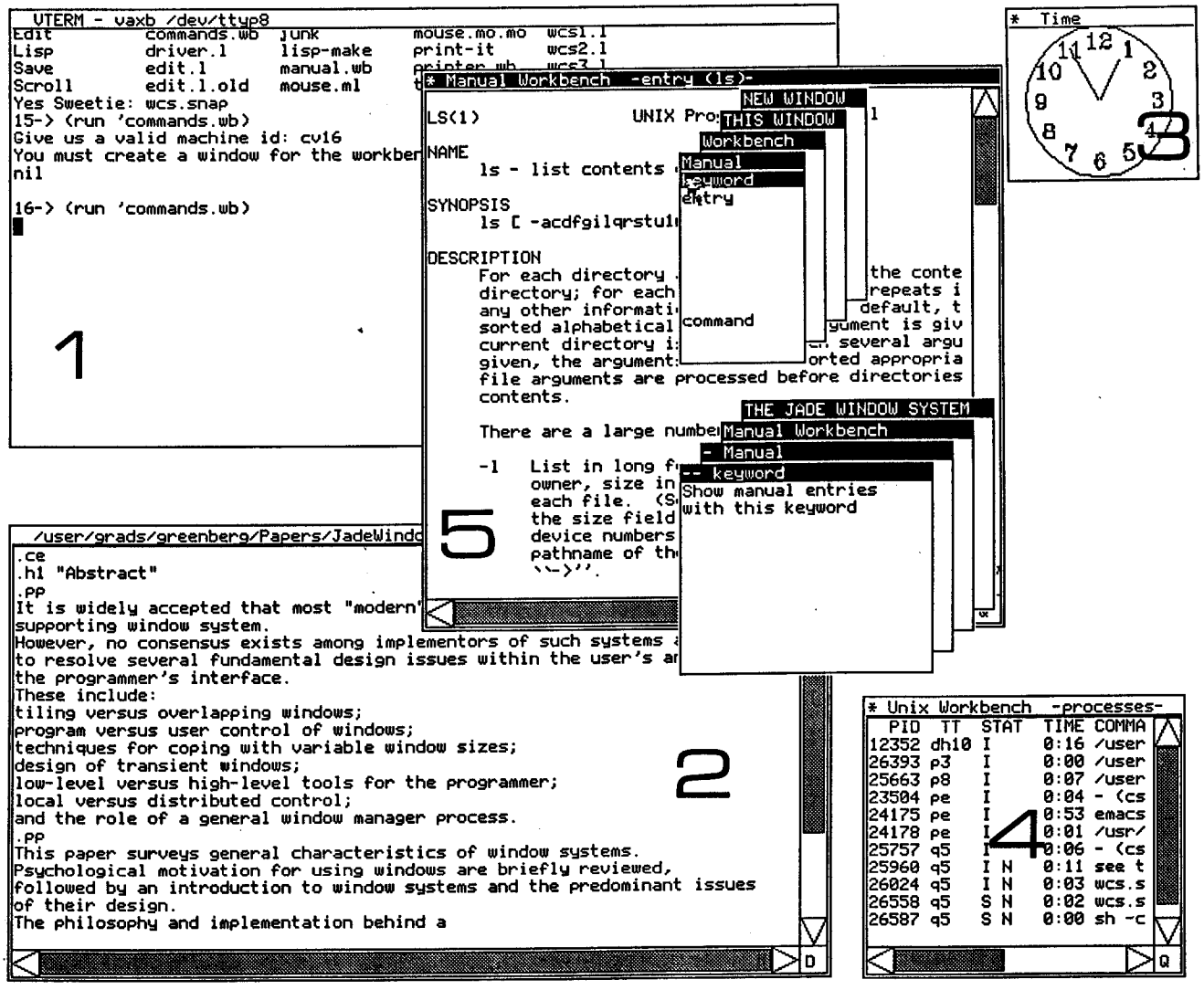


Figure 4: The Jade Screen

cumbersome. Although these menus provided a very high level of functionality, they were inflexible. This does not seem to bother either the programmers or users.

There is disappointingly little use of the help windows. In particular, those associated with the system menus are rarely used. However, some programmers and users think they have merit, especially for first-time or infrequent users of certain complicated application programs. If JADE had a richer set of menu-based application packages, we suspect that the help facility would have been worthwhile.

Users complain about having to use menus for frequently-invoked actions. In particular, all window manipulation — including creation, placement, resizing, and bringing to the top of the pile — is through menu actions only. We would now choose direct manipulation techniques instead (Shneiderman, 1983), perhaps by reserving small areas of each window as buttons with specific functions. (The Apple Macintosh is particularly effective in this regard.) Other frequent menu operations are for textual cutting and pasting. We recommend allowing keyboard actions to be bound to menu selections, for many users find moving from the keyboard to the mouse and back again annoying — particularly during text editing. As a general solution, panels of soft buttons which are always visible should be available as an alternative representation of a menu.

Application programmers had trouble presenting transient messages to the user in their window-oriented applications. As a result, dialogue boxes were included later to allow attention and error messages and selection requests to be handled uniformly across applications.

5. The window manager is implemented as a monolithic process. JWM is a complex piece of software — a single process which simulates concurrency internally. It owns the entire display, and arbitrates requests from all other processes that wish to manipulate the screen. These processes communicate requests to the JWM by passing it messages. Thus applications which reside on different machines look the same as local ones to the window manager.

An application that needs to do anything not handled specifically by the JWM, must use a special request facility. The programmer writes code encapsulating the desired behaviour, downloads it, and invokes it through the window manager. In effect, the program segment *becomes* the window manager for its lifetime. However, this meant that all normal JWM functions (such as popup menus) disappear for that duration.

In general the monolithic approach, although acceptable to most users, received heavy criticism from the original designers and the few programmers who needed flexibility. First, it was difficult for a single application to change the view of system functionality, since the window manager process controls the larger part of this, not the application. For example, it is impossible to change just the appearance of a popup menu without re-writing and replacing the entire window manager. Second, the extensive interprocess communication through messages was often too slow for adequate realtime response. Third, implementing multi-tasking inside the JWM when a multi-tasking kernel was already available made its original design and later modification difficult and complex.

We recommend building a window manager as a set of multiple processes for the sake of modularity. The coding and modification exercise would be eased considerably. If, for example, a programmer wished to change the appearance of the popup menu, then only the menu process would need replacing. Also, it would be easier to supply and test alternative representations of any particular process. For speed, we recommend that processes residing on the local machine use a procedure-call interface to the window manager. Remote processes could employ a server on the workstation to achieve transparent access.

6. The JWM provides limited functionality. A strict set of moderately high-level primitives is embedded as a library within the JWM kernel, accessible only through requests. Applications may read and write to an identified window, but not to arbitrary screen areas. Programs recognize user actions within a window by receiving one of a pre-defined set of input events. Output is through very simple text and graphics primitives.

The original version of the JWM library was difficult to use by non-JADE experts. First, although embedded within any one of a number of standard host languages, requests were made in the unfamiliar message-passing language provided by JADE. Second, there was no real hierarchy of functionality available. Although one could draw lines and text strings easily, much programming was need for anything more complex than that. Third, the unavailability of some kernel primitives (such as text fonts and *bitblit*) discouraged all but the most ambitious from pursuing certain lines of development.

In respose to pressure from both application designers and users, these deficiencies were eventually overcome. As part of the JADE project, a high-level and extremely powerful 2-D graphical language was constructed on top of JWM primitives. Similarly, a toolbox of subroutine calls, based on the original primitives, added high-level interface constructs such as scroll bars and scrollable text, buttons, dialogue boxes, and so on (Greenberg, 1985). Although far from complete, these constructs eased the programming task enormously and promoted interface consistency.

In retrospect, the array of primitive actions implemented was not rich enough. Font and text-editing primitives, as well as *bitblit* support, should have been installed in the kernel. For those few programmers desiring access below the primitives, a multi-processor window manager described previously would suffice. As the majority of programmers required high-level facilities, a reasonable number of high-level constructs should have been designed and built as part of the original system.

Conclusions

Window systems are having a far-reaching effect on the user interface. Although they can add considerable complexity to the user's perception of the system, this is mitigated by direct manipulation techniques, system-wide interface conventions, and instantly-available help facilities. The added complexity for those who write window-oriented application programs is not so easily disguised, but can be alleviated by a well-designed programmers' interface which allows access to standard facilities offered by the window manager at a variety

of levels. Despite this complexity, window systems promise great benefits in naturalness and ease of use, and mark a new phase in the acceptance of interactive computer systems. They are altering the expectations of users, and are forcing programmers to change how they think about the human interface.

The JWM has had a significant impact on the way we work in the Computer Science Department at Calgary. Many people routinely run several virtual terminal windows on their screen, often connected to different computers. (Users may access any of four VAX-11/780's through popup menu selections.) A return to conventional VDU terminals would be seen as a significant reduction in service. Application programmers are no longer content to develop glass-teletype interfaces, but now use direct manipulation techniques where possible. The teaching program has also been affected. A new course on the technology of office information systems was created to instruct undergraduates in modern techniques of interface design (for example, the use of windows). Another, on distributed systems, uses JADE and the JWM for teaching about concurrency at a far greater depth than was previously possible.

This paper has identified and discussed some of the issues raised by the design of window managers, so that our experience may benefit other implementers. By pointing out the consequences of certain design decisions, and by listing some alternatives and the trade-offs involved, a foundation for the design of better window management systems has been laid. Yet we do not offer a recipe for design. Not only are many issues as yet unresolved, but the ideal window system will depend on the users of the system and the tasks they undertake.

Acknowledgement. We would like to give special thanks to Radford Neal, the chief architect of the JADE project, and all the people that took part in the animated and often heated discussions that preceded the design of the JWM. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

References

Bannon, L., Cypher, A., Greenspan, S., and Monty, M. (1983) "Evaluation and analysis of users' activity organization" *Proc ACM CHI 83 Human Factors in Computing Systems,* 54-57, Boston, December 12-15.

Bly, S.A. and Rosenberg, J.K. (1986) "A comparison of tiled and overlapping windows" *Proceeding of the ACM SIGCHI '86 Human Factors in Computing Systems,* 101-106, Boston, April 13-17.

Card, S.K., Pavel, M., and Farrell, J.E. (1984) "Window-based computer dialogues" *Interact '84 - First IFIP Conference on Human-Computer Interaction, 1,* 355-359, London, UK, Sept 4-7.

Cohen, E.S., Smith, E.T., and Iverson, L.A. (1985) "Constraint-based tiled windows" Research Report, Computer Science Department, Carnegie Mellon University, Pittsburgh.

Espinosa and Hoffman (1983) "Macintosh user interface guidelines (2nd edition)" in *Inside Macintosh.* Apple Computer Inc.

Furnas, G.W. (1986) "Generalized fisheye views" *Proceeding of the ACM SIGCHI '86 Human Factors in Computing Systems,* 16-23, Boston, April 13-17.

Gosling, J.A. and Rosenthal, D.S.H. (1983) "A network window-manager" Report, Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA 15213.

Greenberg, S. and Witten, I.H. (1985) "Interactive end-user creation of workbench hierarchies within a window interface" *Proc Canadian Information Processing Society National Conference,* Montreal, Quebec, June.

Greenberg, S. (1985) "The toolbox manual - a high level approach to the Jade window manager" Internal report, SRDG group, Department of Computer Science, University of Calgary.

Joy, W. (1979) "An introduction to the C shell" Computer Science Division Report, University of California, Berkeley, California.

Malcolm, M. and Dyment, D. (1983) "Experience designing the Waterloo Port user interface" *Proc ACM Conference on personal and small computers,* 168-175, San Diego, California, December.

Neal, R.M., Lomow, G.A., Peterson, M.W., Unger, B.W., and Witten, I.H. (1984) "Inter-process communication in a distributed programming environment" *Proc Canadian Information Processing Society National Conference,* 361-364, Calgary, Alberta, May.

Norman, K.L., Weldon, L.J., and Shneiderman, B. (1985) "Cognitive representations of windows and multiple screen layouts of computer interfaces" Research report CAR-TR-123, CS-TR-1498, Dept of Computer Science, U. of Maryland, MAY.

Pike, R. (1983) "Graphics in overlapping bitmap layers" *ACM Trans on Graphics, 2* (2) 135-160, April.

Rosenthal, D.S.H. (1982) "Managing graphical resources" *Computer Graphics, 16* (4) 38-45, December.

Shneiderman, B. (1983) "Direct manipulation: A step beyond programming languages" *IEEE Computer, 16* (8) 57-69, August.

Shneiderman, B. (1984) "Response time and display rate in human performance with computers" *Computing Surveys, 16* (3), September.

Shneiderman, B., Norman, K., Rogers, J., Arifin, R., and Weldon, L. (1985) "A multi-screen programmer work station based on the IBM PC" Research report, Dept of Computer Science, U. of Maryland, April.

Smith, D.C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. (1982) "Designing the Star user interface" *Byte,* 7 (4) 242-282.

Spence, R. and Apperley, M. (1982) "Data base navigation: an office environment for the professional" *Behaviour and Information Technology, 1* (1) 43-54.

Stallman, R.M. (1981) "EMACS the extensible, customizable self-documenting display editor" *ACM Sigplan Notices — Proceedings of the ACM Sigplan SIGOA symposium on text manipulation, 16* (6) 147-155, Portland, Oregon, June 8-10.

Symbolics (1985) "Using the online documentation system" in *User's Guide to Symbolics Computers.* Symbolics, Inc., March.

Teitelman, W. and Masinter, L. (1981) "The Interlisp programming environment" *IEEE Computer, 14* (4) 25-34, April.

Test, J.A. (1982) "The NUnix window system" Internal report, Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

Webber, A.B. (1986) "Amiga vs. Macintosh" *Byte, 2* (9) 249-256, September.

Williams, G. (1984) "The Apple Macintosh computer" *Byte, 9* (2) 30-54.

Witten, I.H. (1985) "Elements of computer typography" *Int J Man-Machine Studies, 23* (6) 623-687, December.

Wood, R.J. (1982) "A window based display management system" Internal Report, University of Maryland.