# WINDOW MANAGEMENT ALGORITHMS FOR AN ALL POINTS ADDRESSABLE DISPLAY

by
John Will Webster III

Submitted in Partial Fulfillment
of the Requirements for the

Degree of Bachelor of Science
at the
Massachusetts Institute of Technology

May, 1983

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science, 1983

Certified by [illegible]
Thesis Supervisor

Accepted by
Chairman, Departmental Committee on Theses

# WINDOW MANAGEMENT ALGORITHMS FOR AN ALL POINTS ADDRESSABLE DISPLAY

by
John Will Webster III

supervised by
Carl E. Hewitt,
Associate Professor of
Electrical Engineering and Computer Science

## ABSTRACT

This thesis discusses issues associated with the implementation of window management systems. The results it presents include: an algorithm for updating overlapping windows without consuming large amounts of memory, a design for multiple-font menus, and a design for a facility to support a general multiple-font text formatter on an all points addressable (APA) display.

TABLE OF CONTENTS

FIGURES

TABLES

# INTRODUCTION

## A WORD ABOUT WINDOWS

In the realm of display devices, we define a "window" to be a rectangular region on a display device through which data is viewed and manipulated. The window abstraction is particularly useful in dividing one physical display device into several virtual display devices to be shared by one or more application programs and their components (Fig. I.1). Window management systems are software packages which implement the window abstraction. Examples of existing window systems include the LISP machine window system (Weinreb and Moon), Smalltalk (Tesler), and, the primary object of our discussion in this thesis, JAWS (for "Jaws A Window System").

JAWS is an application-independent, general-purpose window management system which is currently implemented on the PERQ personal computer, a PASCAL machine with 1MB of primary storage (PERQ), and supports an all points addressable (APA) display. One of the essential features of JAWS is its general nature; lack of dependence on application programs allows JAWS to be used easily in several different applications. To date, several application programs, including a 3270 terminal emulator (O'Hara), the graphic editor which was used to draw the illustrations for this thesis, and an integrated real-time editor/formatter known as POLITE (for Personal On-Line Integrated Text Editor)(Borkin and Prager), have been written using JAWS for screen support. This thesis is concerned with the

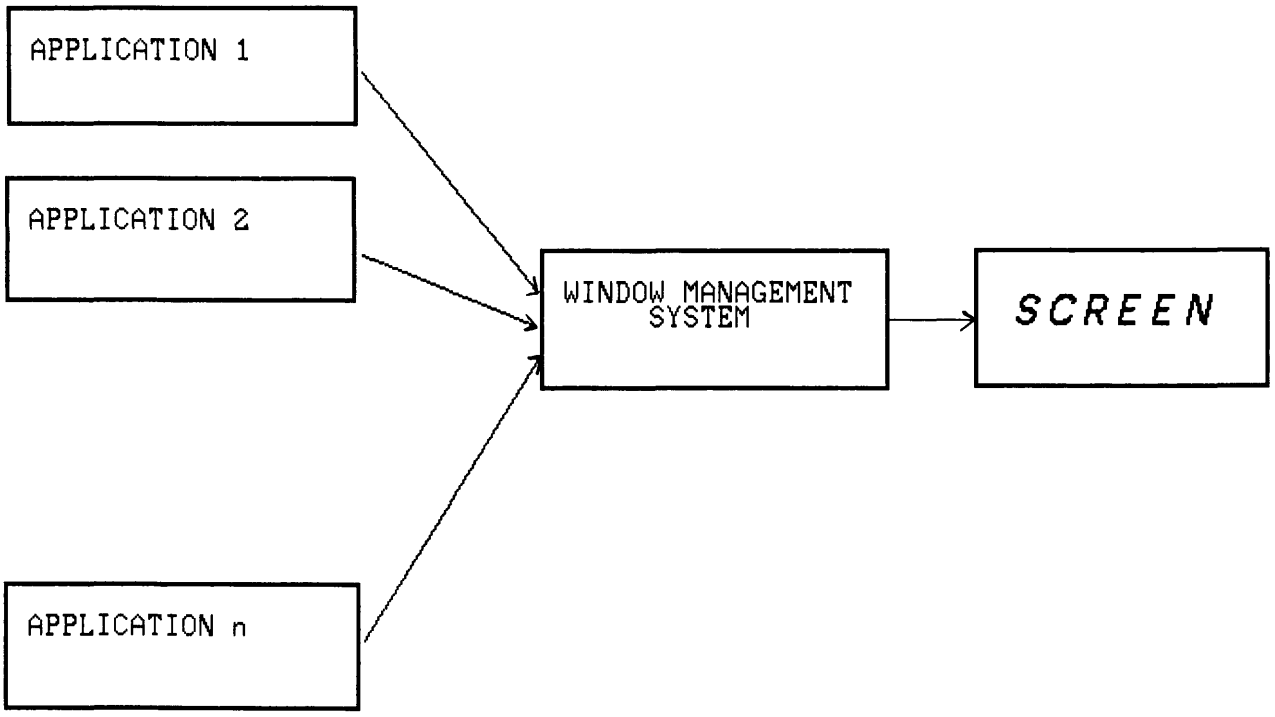


Fig. I.1 -- The window management system enables several applications to share the same physical screen.

development of some of the window management algorithms and techniques used in JAWS and uses POLITE to illustrate how JAWS might be used to support a useful application.

## ABOUT JAWS

To begin, a summarization of the state of JAWS at the initiation of the work described in this thesis is in order. The original design and basic implementation of JAWS were done by John C. Gonzalez (as his bachelor's thesis at MIT) and Robert P. O'Hara of IBM (while on sabbatical at IBM's Cambridge Scientific Center). A complete documentation of this effort is provided in Gonzalez' thesis, <u>Implementing a Window System on an APA Display</u> (Gonz), and should be consulted for details beyond those which I provide here.

There are two basic image-containing structures in JAWS: "canvasses" and "windows" (Fig. I.2). Canvasses correspond roughly to the "world" of the Core Graphics Standard (Siggraph) in that they contain the data which is to be manipulated by the application program; likewise, windows correspond to the Standard's "viewports" in that they "view" (i.e. contain the translated image of canvas data, particular canvasses. Note that although a window may view only one canvas at a time, there is nothing to prevent several windows from viewing the same canvas.

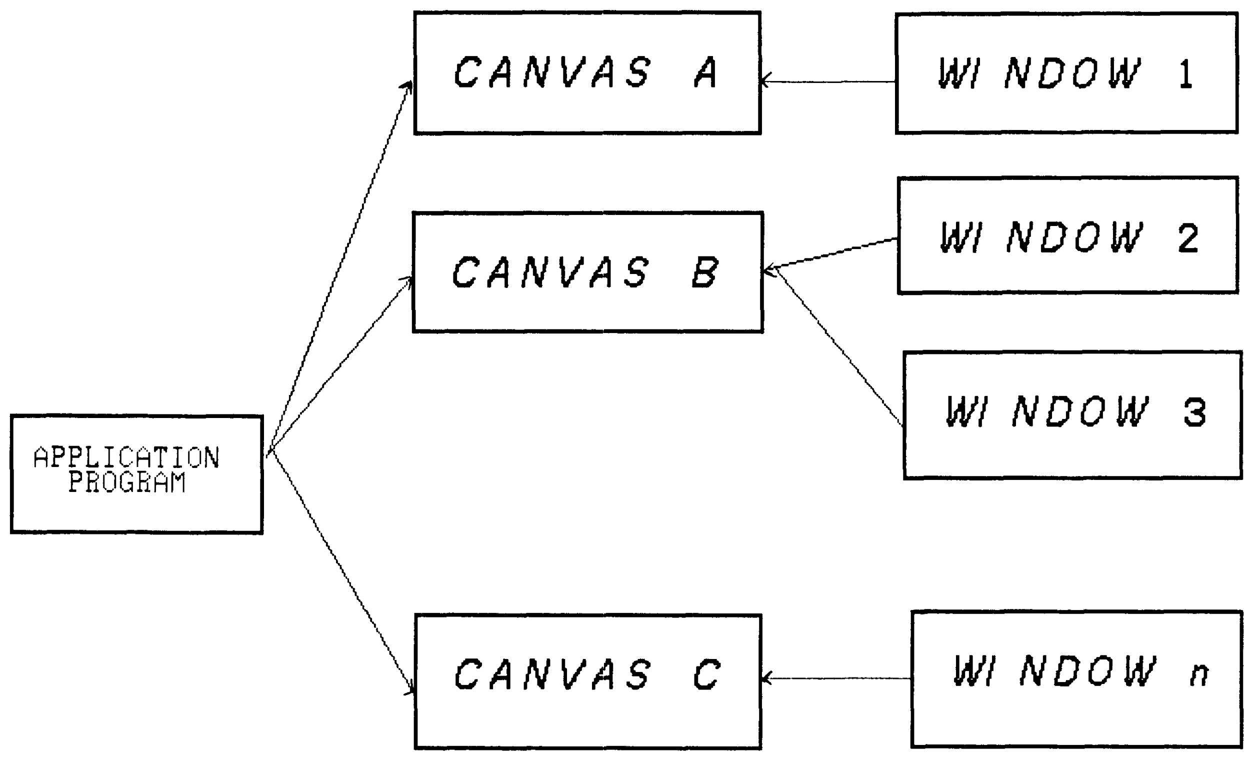


Fig. I.2 -- The application program writes to canvasses which are viewed by windows.

## THE CANVAS DATA STRUCTURE

There are several different types of canvasses, differing in their representations and in the routines used to manipulate them. The collections of routines which support these different canvas types are known as "flavors". Thus, to manipulate a particular canvas, the application program would call routines available in the flavor which corresponds to the type of that canvas (Fig. I.3). For example, "CHRFLAVOR" is a canvas flavor which supports text-oriented operations. The underlying representation, which is, of course, hidden from the application program, of a CHRFLAVOR canvas is a vector of characters; the routines used to manipulate it may include such things as MOVE_CURSOR, WRITE_CHARACTER, and so on. Fig. I.3 shows the original four flavors of JAWS: CHRFLAVOR is explained above, BITFLAVOR supports a bit-mapped canvas, QUIXFLAVOR supports a canvas of vector endpoints, and WINFLAVOR supports windows with no canvasses (more about this in Chapter 3). Canvas A in Figure I.3 is of type CHRFLAVOR while canvasses B and Z are both of type QUIXFLAVOR.

## THE WINDOW DATA STRUCTURE

Each window (Fig. I.4) contains pointers to the canvas it views, to its "screen buffer", which contains a representation of the canvas' contents in a form intelligible to the display device, and, possibly, to a list of child windows. A child

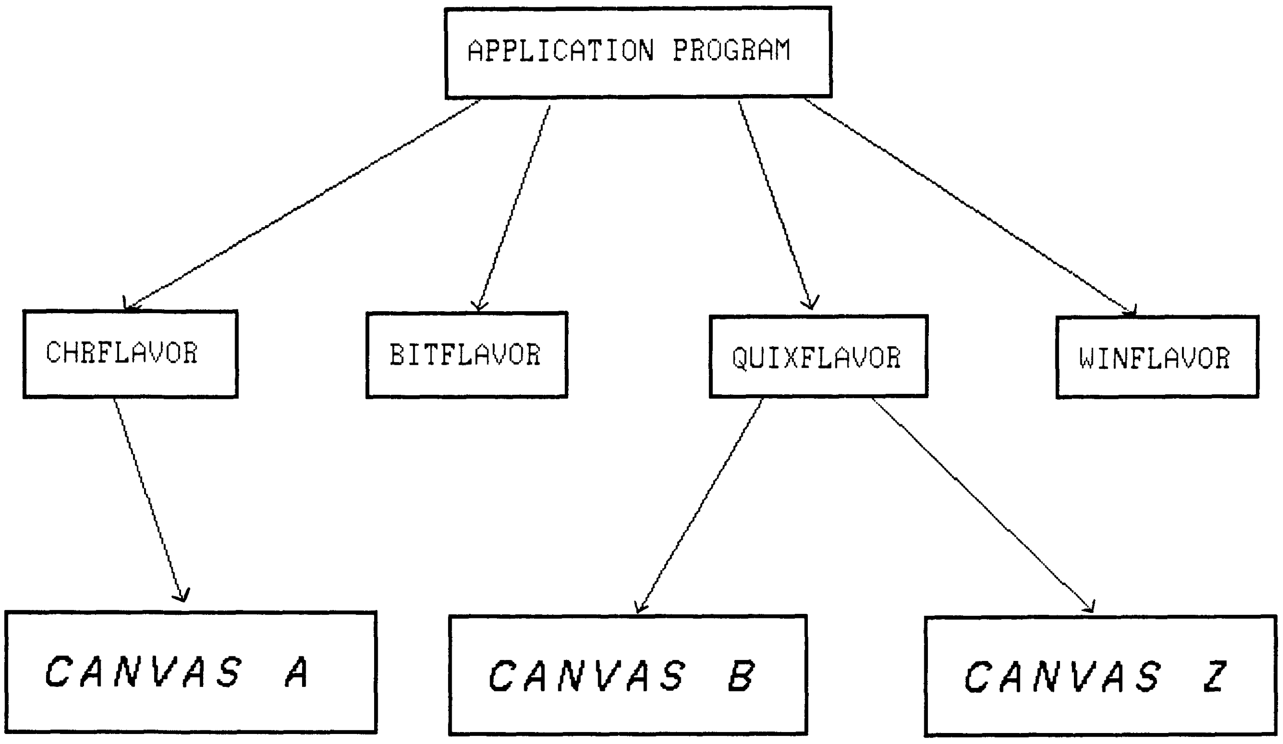


Fig. I.3 -- The application program manipulates the canvasses through the routines provided by the appropriate flavor package.

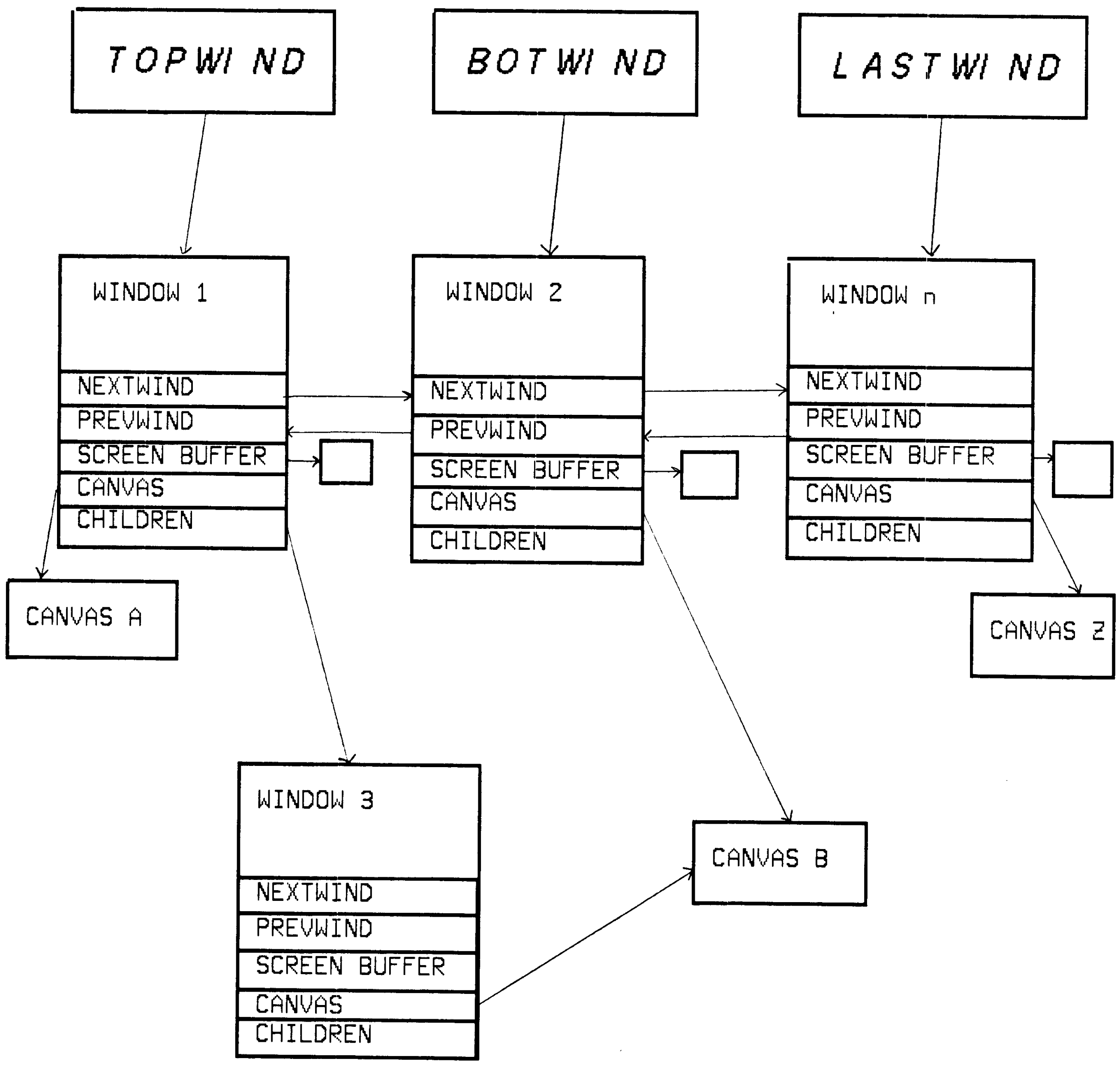


Fig. 1.4 -- The window hierarchy.

window is no different from its parent in structure (i.e. it can have siblings, children, and a view of a canvas). It is different, however, in that its existence is limited to the region occupied by its parent and it shares its eldest parent's screen buffer (i.e. if a window is a child of some other window, it has no screen buffer of its own).

Windows are assembled in an hierarchy which determines the order in which they are displayed on the screen. There are three special pointers into that hierarchy: TOPWIND, BOTWIND, and LASTWIND. The windows between TOPWIND and BOTWIND are visible on the display with TOPWIND appearing to be above all other visible windows and BOTWIND appearing to be beneath all other visible windows. This gives a "2-1/2"-D quality to the display and allows the user to think of overlapping windows just as he would overlapping papers on his desk. The windows between BOTWIND and LASTWIND are said to be "hidden", meaning they are not visible on the display at all.

In Fig. I.4, then, Window 3 would be "displayed" (i.e. have the contents of its screen buffer transferred to the display device) first, then Window 1. Window 2 is never explicitly displayed since it is a child of Window 1 and shares its screen buffer (i.e. its screen image was displayed automatically when Window 1 was displayed). Window n is never displayed since it is hidden. Thus, when portions of more than one window occupy the same space on the physical display, which window is visible to the user is determined by its position in the window hierarchy.

The order of the hierarchy is originally determined by the order in which the application program defines windows. JAWS does, however, provide operations in the "window manager" which alter that order. "Surfacing" a window means that it will always be displayed last (i.e. it is TOPWIND); "burying" a window causes it to be displayed first (it is BOTWIND); "hiding" a window causes it to become invisible to the display (however, its screen buffer is still updated in the event of a change in its canvas). In Fig. I.5, we have surfaced the window containing the clock display, in Fig. I.6 we have buried it, and in Fig. I.7 we have hidden it.

## THE WINDOW UPDATE CYCLE

The changes made to a canvas are not immediately made visible on the display device; instead, the application program must explicitly indicate that it desires to have the changes to the canvas be made visible before a screen update occurs. This allows long and complex changes to the canvas seem instantaneous to the user.

The application program notifies JAWS through the "canvas manager" that it wishes to have the changes made to a certain canvas transferred to the screen. This causes a series of events

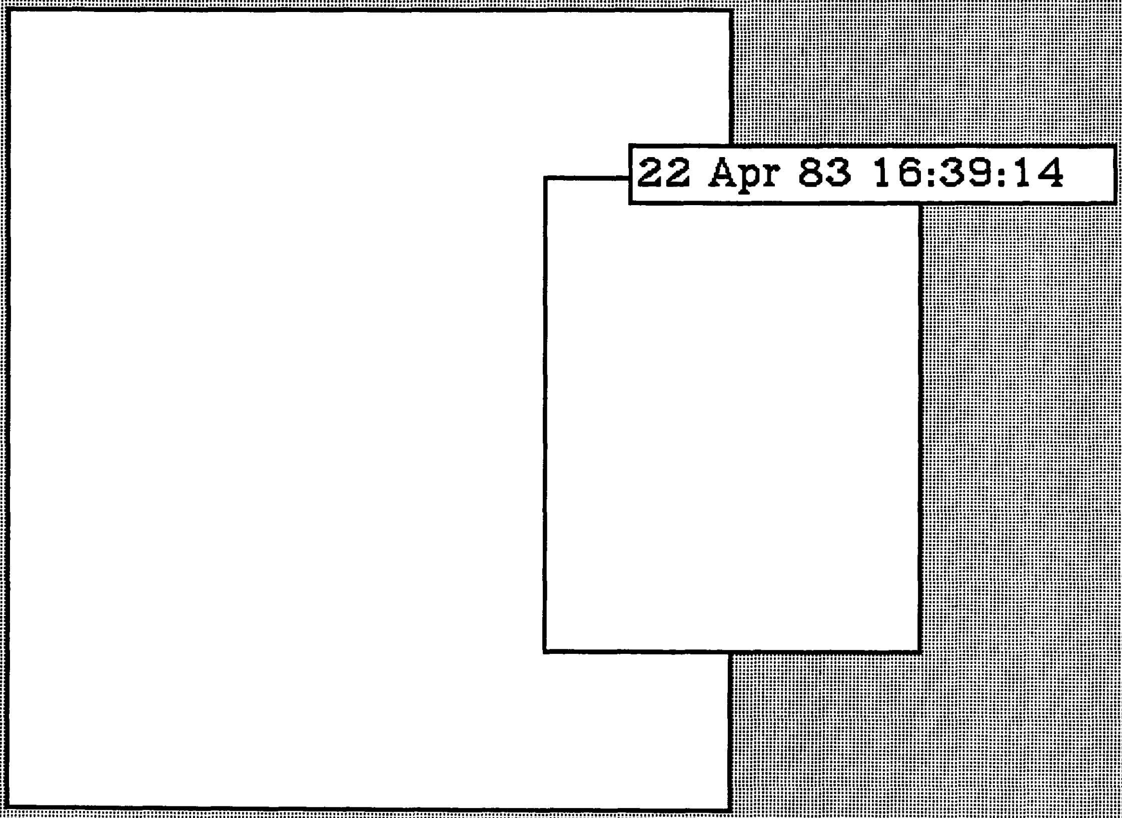


Fig. I.5 -- The clock display is surfaced.

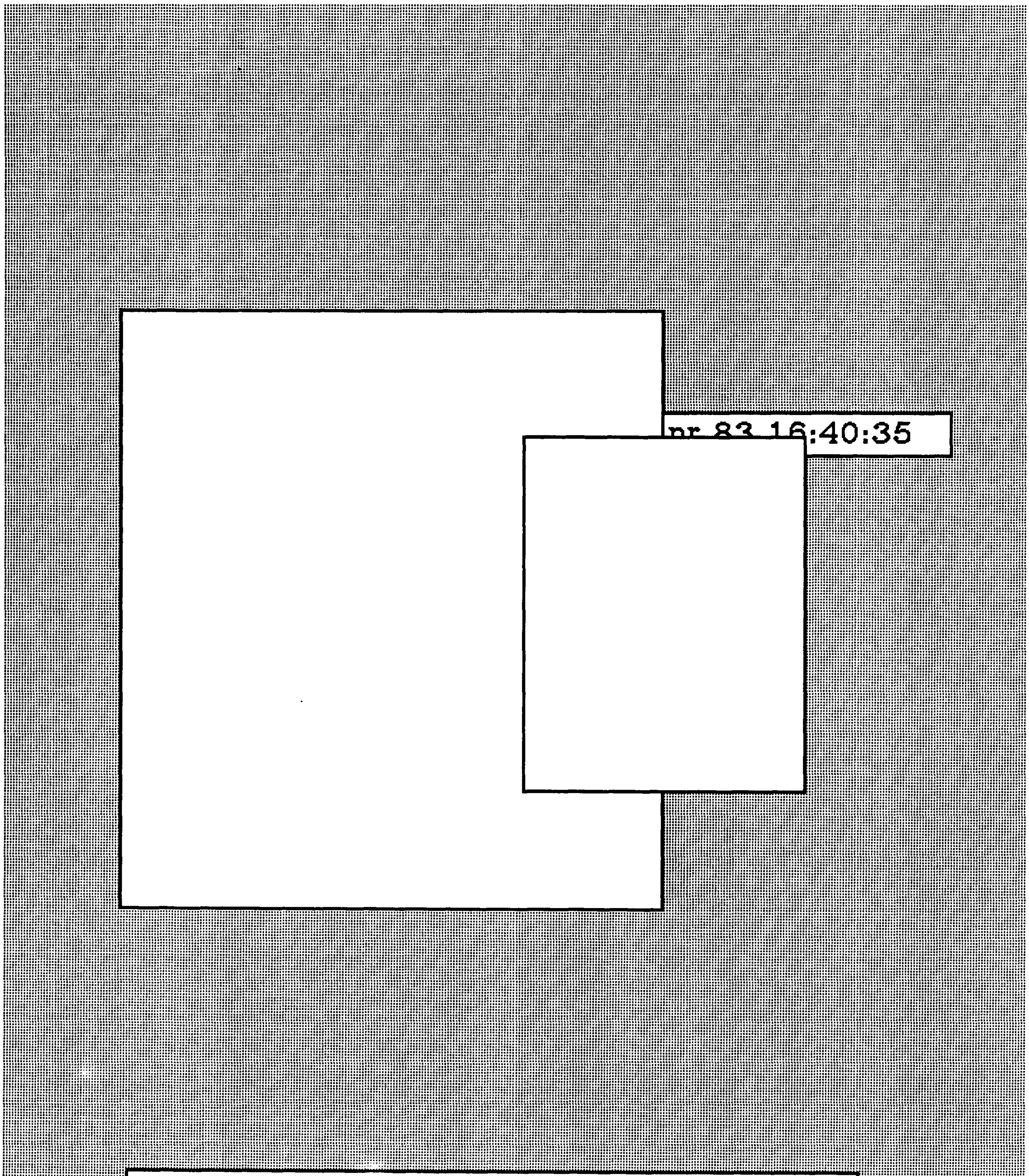


Fig. I.6 -- The clock display is buried.

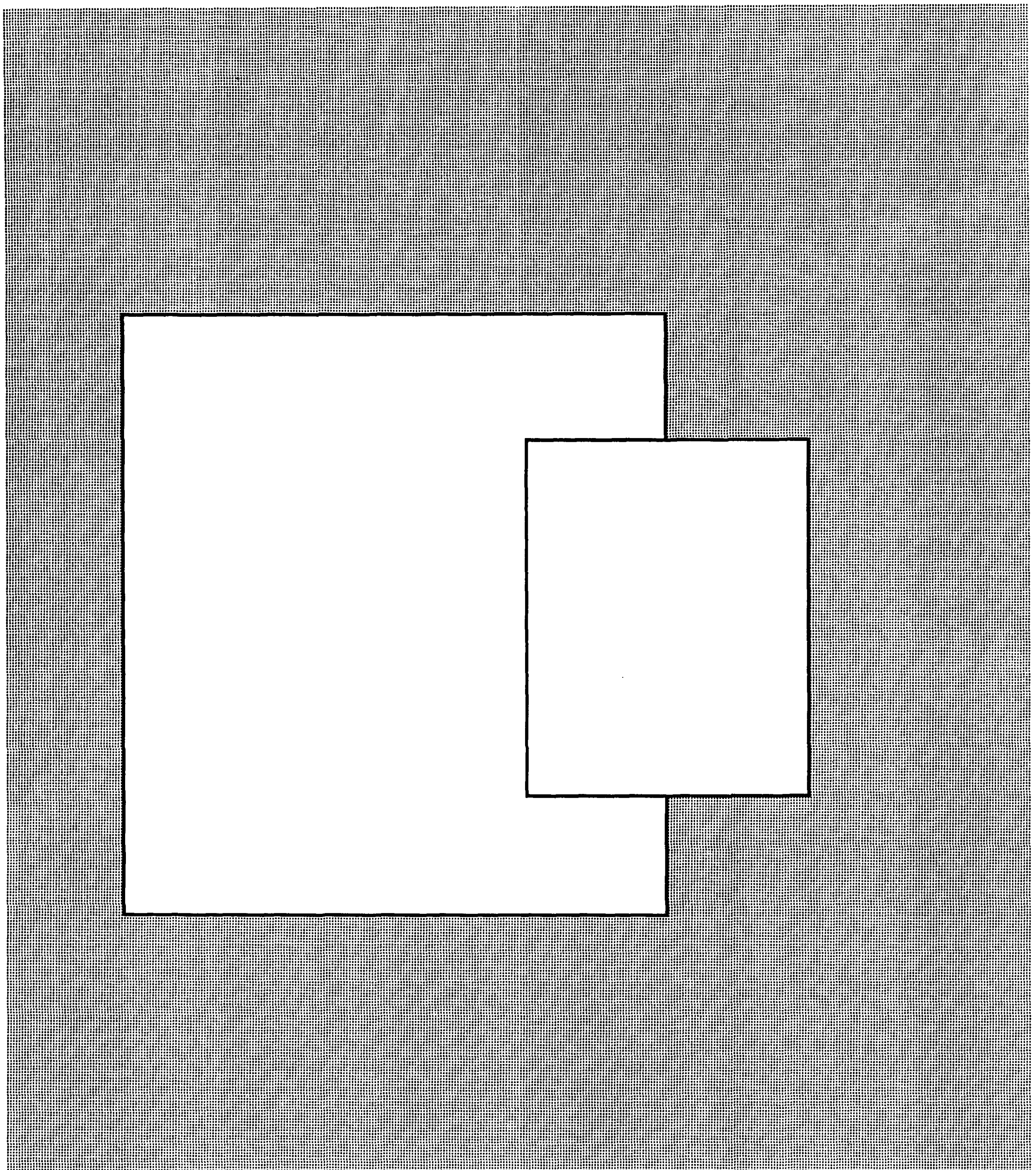


Fig. I.7 -- The clock display is hidden.

to occur (Fig. I.8). First, the canvas manager places the changed canvas on a queue (the "CANVAS QUEUE") for processing by the window manager and invokes that manager. In Figure I.8, canvas B has changed and is therefore on the CANVAS QUEUE. For each canvas on the CANVAS QUEUE, the window manager traverses the entire window hierarchy updating the screen buffers of those windows which view the changed canvas as is necessary by flavor dependent translation routines (these will be discussed in more detail in Chapter 2) and placing them on the "REDISPLAY QUEUE". In Figure I.8, windows 2 and 3 are placed on the REDISPLAY QUEUE since they both view canvas B (see Fig. I.4). When this screen buffer update process is completed, the window manager invokes the "screen manager". For each window that is on the REDISPLAY QUEUE, the screen manager traverses the entire window hierarchy and displays each window therein on the display so that all changed windows are updated and the depth relationships between the windows remain the same. Were the screen manager to redisplay only the updated windows, it may occur that some obscured window would become the obscurer of another window which logically (i.e. according to the window hierarchy) lies on top of it (a clear mistake). The process of redisplaying each window in the window hierarchy is known as "reburying" the changed window. This screen refresh completes the update cycle.

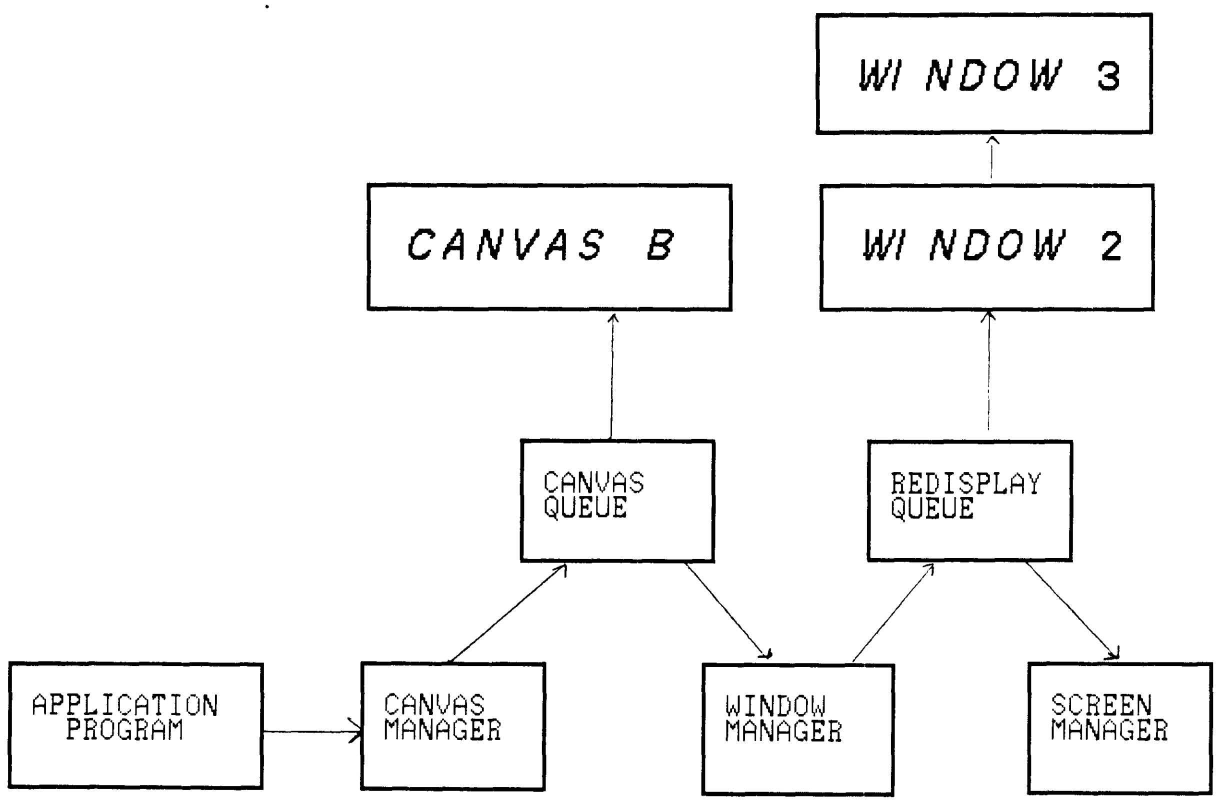


Fig. I.8 -- The window update process.

## THE UTILITY ROUTINES

Just as the application program must perform flavor dependent manipulations on the canvasses, so must the various managers sometimes perform operations which vary with the type of canvas a window is viewing. An example of this occurs when the window manager tries to correlate a point in a window with a point in the canvas the window views. Obviously the way in which this is done varies with the type of canvas the window views. Because of this, there is a need for generic procedures in JAWS (e.g. a generic correlation routine). Since PASCAL does not support such procedures, we emulate them by using a dispatch module as shown in Fig. I.9 where the flavor utility routines are analogous to the flavor routines seen in Fig. I.3. These utility routines are hidden from the application programmer.

## ARCHITECTURAL BENEFITS

An important design feature of JAWS is that the managers mentioned above (i.e. the canvas, window, and screen managers) communicate through queues. This arrangement allows JAWS to run in a multi-processing environment. Not only can several application programs run concurrently and use the same window management system, but also the various managers of JAWS can run concurrently so that events not related to one another can occur independently. For instance, in a single process environment, all canvasses have to be translated into the appropriate screen

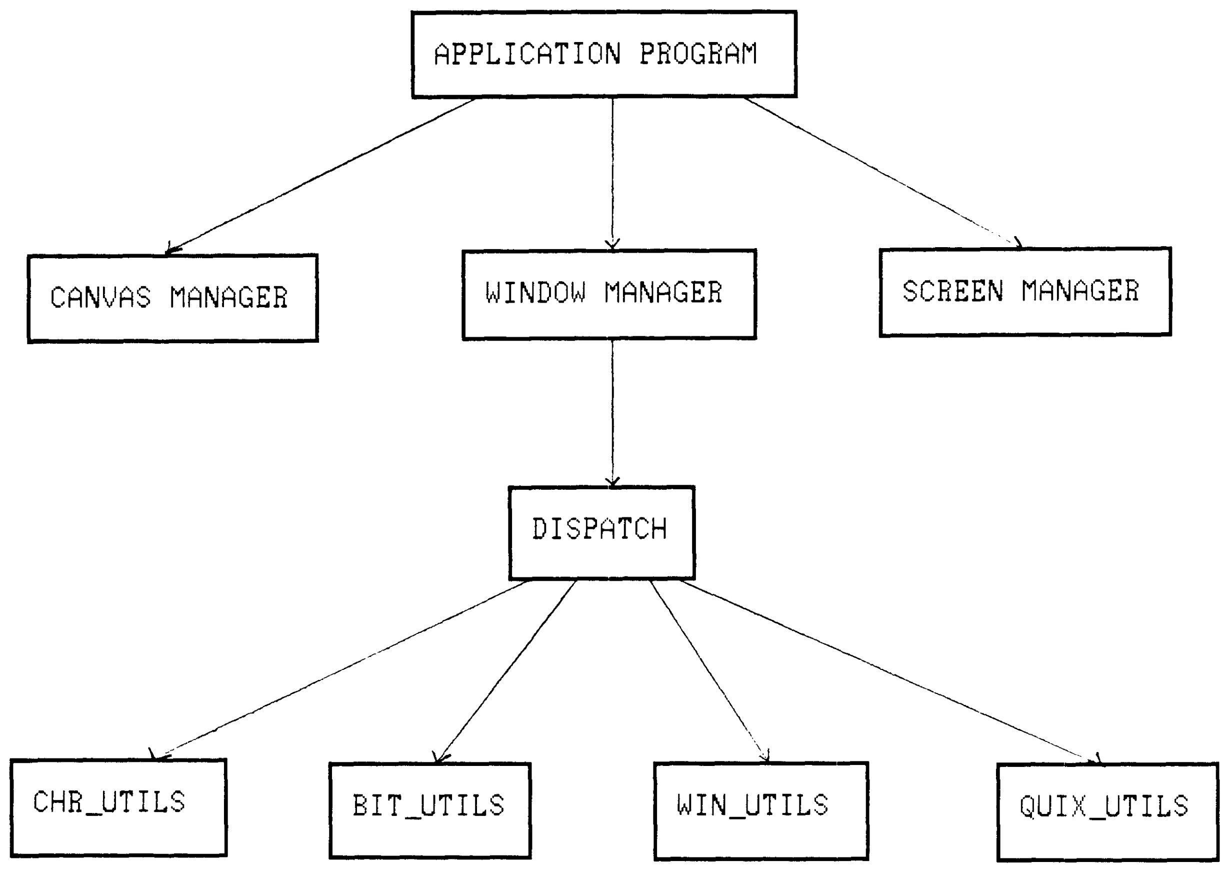


Fig. I.9 -- The DISPATCH module emulates generic procedures for JAWS managers.

buffers before the screen buffers can be redisplayed. This is logically unnecessary since the display of a given screen buffer depends in no way upon the translation of a canvas into another screen buffer. Also the queue architecture makes it extremely easy to add new managers to JAWS.

A second important benefit of JAWS' design is that application programs written using JAWS are somewhat device-independent. By this I mean that if JAWS were to be moved to another machine or made to support a different device, application programs written using JAWS would still run (provided the language in which they are written is supported on the new machine) since the interface to which they were written (i.e. that provided by the three managers and the flavor routines) would still be the same.

Another benefit is that JAWS can be readily expanded to support different canvas data representations. To do so, one simply writes a flavor package to support the new representation. There is no theoretical limit to the number of flavors which can be supported by JAWS; there may, however, be a practical limit imposed by the system supporting the implementation (for instance, some linkers may have a limit to the number of separate modules a program can contain). Two examples of flavor additions are detailed in subsequent chapters.

## SOME INADEQUACIES

The design just described has several problems whose solutions are the major focus of this thesis. The most visibly apparent problem (to one who is running a program using JAWS) concerns the reburying of changed windows. Each reburial causes the screen to flicker slightly as the various windows are displayed. The source of this problem is that to accomplish reburial, JAWS' screen manager redisplays every visible window in the window hierarchy in its entirety each time a screen buffer is changed (i.e. each time something appears in the redisplay queue). The resultant flickering is intolerable in interactive applications which require several screen updates per second. As is detailed below, this is neither necessary nor efficient and has been greatly improved upon.

A second major problem with the above design is its handling of child windows. The original design of JAWS makes the following distinction between child windows and windows with no parents ("top-level" windows): only top-level windows have screen buffers (see Windows 1 and 2 in Fig. I.4). This has several implications. For one, child windows are restricted to not overlapping, for if they were to overlap, one child's screen image would be destroyed by the other and would have to be regenerated from the canvas. Another implication is that if a top-level window has children and is subsequently updated, the screen image of the children would be overwritten and would have to be regenerated from the canvas. A third drawback is that child windows have to be treated differently from top-level windows by the window manager and by the screen manager. Consequently, these two managers are filled with special case code which could be eliminated were child windows to be treated in a more general fashion.

Another small, but significant, design oversight is the lack of a facility for defining windows which have no screen buffers. There are four possible combinations of possessing screen buffers and canvasses; JAWS exploits two of these (buffer/canvas and buffer/no canvas), but ignores the other two (no buffer/canvas and no buffer/no canvas). The no buffer/canvas option is similar to the default configuration of windows on the Lisp Machine (Weinreb and Moon). Its main advantage is that it saves storage (this is especially true on a raster display). Of course, such a concept is applicable to only a small set of applications. Since the screen manager does need to refresh the screen, the screen image of the canvas must be easily generated. Also, in order to make the storage savings worth the loss in update time, the canvas representation must be fairly compact. The details of the implementation of this feature and of the development of a canvas flavor which utilizes it are given in Chapter 3.

Another deficiency affected JAWS' usefulness for interactive programs. This is not the result of a design oversight; it is simply a lack of function. Often when using interactive programs, we select from a small set of commands and data. In such a case, it is both feasible and more desirable to select the command or datum from a menu instead of typing it in from the keyboard. When a pointing device (such as a puck or a mouse) is available, this feature becomes even more desirable since the user need only know how to move the pointer and indicate a selection. In some cases, to save screen space, we may wish for the menu to display only some of the available options and to have the ability to scroll to the other ones. Also, we may wish to have a facility for indicating some modal difference between various options on the menu (such as between commands and arguments). To avoid having each application program implement its own menu facility, a pop-up menu flavor, which supports scrolling and multiple-fonts, has been added to JAWS. The details of its implementation (given in Chapter 2) provide insight into writing flavors for JAWS.

## THESIS ORGANIZATION

The remainder of this thesis is devoted to providing more detail on the issues mentioned above. Chapter 1 details the various screen management algorithms used to solve the reburial problem; Chapter 2 describes the implementation of pop-up menus from the original design considerations to the final data structures and algorithms used; Chapter 3 describes how JAWS might be modified to support canvasses whose screen images are generated dynamically instead of being buffered; Chapter 4 discusses the advantages and disadvantages we have experienced with the two implementations of child windows (i.e. with and without screen buffers); Chapter 5 discusses how JAWS might be used to support POLITE.

# CHAPTER 1

# SCREEN MANAGEMENT TECHNIQUES

There are two major steps involved in achieving efficient screen management. The first step involves limiting the area of the screen management to only that area which has been changed. This may seem to be an obvious procedure, but it is one which the original implementation of JAWS ignored (recall that the reburial process was accomplished by redisplaying EACH window in the hierarchy in its ENTIRETY). A related but ancillary step is to determine the bottom (i.e. closest to BOTWIND) window to be redisplayed so that redisplay of the entire hierarchy is not necessary. For instance, if a window is being surfaced, then it is the bottom window to be redisplayed and the area to be redisplayed is the area filled by the window.

## COMPUTATION OF THE CHANGED AREA

The actual calculation of the extent of the changed area is done by the redisplay routine. The routine (Fig. 1.1), expressed for this thesis in terms of Pascal plus set operations, first computes the window's absolute offset on the screen by summing the offsets of its parents from their parents (ultimately, the offset of a top-level window from the origin of the screen is included in this sum). Next, the redisplay routine computes the absolute offset of the changed area by adding the window's absolute offset to the offset of the changed area within the window. The width and height of the changed area were previously determined by the caller of the

```
PROCEDURE redisplay(window, startx, starty, width, height)
  [xoffset <- window->.offsetx; yoffset <- window->.offsety;
   parent <- window->.parent

   s <- { all ancestor windows of window }
   $\forall$ w $\in$ s DO
     [xoffset <- xoffset + w->.offsetx
      yoffset <- yoffset + w->.offsety]

   xoffset <- xoffset + startx; yoffset <- yoffset + starty;
   enqueue(redisplay_queue, window, xoffset, yoffset, width, height)
   screen_manager]
```

Fig. 1.1 -- The redisplay algorithm.

redisplay routine; therefore, if the entire window were to be redisplayed, the redisplay routine would be called with offsets of zero (indicating to redisplay from the upper-left-hand corner of the window) and width and height equal to the width and height of the window.

After it computes the absolute changed coordinates, the redisplay routine creates a queue element containing the above information (the window to be displayed and the changed area) and places it on the redisplay queue. It then invokes the screen manager, whose job it is to place the window in question on the screen in its proper position with respect to both the screen's coordinate system and the window's position in the hierarchy. The screen manager faces the task of computing exactly what is visible to the user. There are several ways of doing this computation; all have different strengths and weaknesses.

## SCREEN MANAGEMENT ALGORITHMS

JAWS' screen manager was implemented using three different algorithms for the overlap computation. The following sections detail and compare those algorithms.

## THE DO NOTHING ALGORITHM

The original enhancement to JAWS employed what I term here the "Do Nothing" algorithm (DNA). Starting with the bottom window (BOTWIND), DNA (Fig. 1.2) examines each window to see if it is "influenced" by the changed coordinates. If it is, only that portion of the window which was influenced is displayed on the display device. DNA continues until it reaches TOPWIND, at which point it is done.

The best feature of DNA is its simplicity; the code is extremely simple and executes quickly. Unfortunately, this algorithm is extremely inefficient. To begin, by starting with BOTWIND, it ignores the fact that only those windows which lie above the window on the redisplay queue need to be placed on the screen to insure its proper reburial. Secondly, by ignoring the relationships the windows in the hierarchy have to one another (i.e. that some windows overlap), DNA often displays pixels which are only going to be overwritten by some subsequent writing in the same reburial. The first inefficiency compounds the second.

DNA's simplicity would be worth the incurred inefficiencies if those inefficiencies did not noticeably affect the screen manager's performance. Unfortunately, they do; the reburial of any large area of the screen (large means approximately one-thirtieth of the screen) results in a very noticeable flicker of the image. If any sort of interaction is occurring in that area (for instance, scrolling), the resultant flickering is very

```
PROCEDURE screen_manager()
  [s <- { windows between BOTWIND and TOPWIND }
   window <- dequeue(redisplay_queue)

   ∀ w ∈ s DO
     [IF (window influences w) THEN
        [(display influenced portion of w)]]]
```

Fig. 1.2 -- The "Do Nothing" algorithm (DNA).

distracting and unappealing. The reason for this is that in an intensely interactive application, the obscured window is displayed almost as frequently as the windows which are supposed to be obscuring it.

## DOING NOTHING TO A SHADOW BUFFER

The second algorithm tried is like DNA, only instead of writing the images directly onto the screen, it writes them into a "shadow buffer" which is subsequently copied onto the screen, thereby eliminating the flickering we experienced with DNA. We call this algorithm DNASB (for DNA in a Shadow Buffer). A second advantage DNASB has over DNA is that it displays only those windows from the changed window forward (DNA displayed each window in the hierarchy).

Since DNASB does not cause any screen flicker, it would seem that it is much better than DNA. This may be true as far as the aesthetics of JAWS is concerned, but we must also consider what price we are paying to achieve that performance. First, since we now have an additional buffer to manage, we need more code (compare to DNA's simplicity) and more time to perform the management. Second, we must consider that the shadow buffer must be capable of containing an entire screen's worth of data (since that is the largest part of a window can be displayed at once). The implication of this is that we must have an additional amount of storage equal to the storage consumed by the screen buffer (in the PERQ, that comes to 96K bytes) set aside for this buffer. If the underlying computer system does not have this much memory, cases may arise in which the algorithm simply does not have room enough to run; if the underlying system has a virtual memory system, this storage consumption may lead to excessive swapping for highly interactive programs. Finally, we note that DNASB is inherently less efficient than DNA since it has to do almost twice as much work just to display the data on the screen. This inefficiency occurs since DNASB has to do work equivalent to that of DNA to generate the image in the shadow buffer and almost as much work as DNA does to move the shadow buffer onto the display. Of course, DNASB's tactic of displaying only those windows lying above the changed window in the hierarchy does give it some redeeming value, but DNA could easily be modified to do the same without taking on any of DNASB's less desirable features.

## THE FRAGMENTATION ALGORITHM

The third algorithm we tried is the best of the three we considered. For reasons which should become clear shortly, we term it the "multiple pass fragmentation" algorithm (MPF). Its basic form is given in Figure 1.3. The idea behind the algorithm is to exploit the notion of "subtracting" an obscured window from an obscuring window and being left with the visible

```
PROCEDURE mpf(window)
  [IF (chged_window influences window) THEN
     [s <- { window }
      ∀ w ∈ s + { window's children } + { window's higher siblings } DO
        [IF (chged_window influences w) THEN
           [∀ v ∈ s DO
              [f <- { fragments from v subtracted from w }
               s <- s - { v } + f]]]

      redisp_lst <- redisp_lst + s]

   IF burying OR surfacing THEN
     [(call mpf on window's children)
      IF burying THEN
        [(call mpf on window's higher siblings)]]]

PROCEDURE screen_manager()
  [chged_window <- dequeue(redisplay_queue)
   redisp_lst <- {}
   mpf(chged_window)

   ∀ w ∈ redisp_lst DO
     [(display w on the screen)]]
```

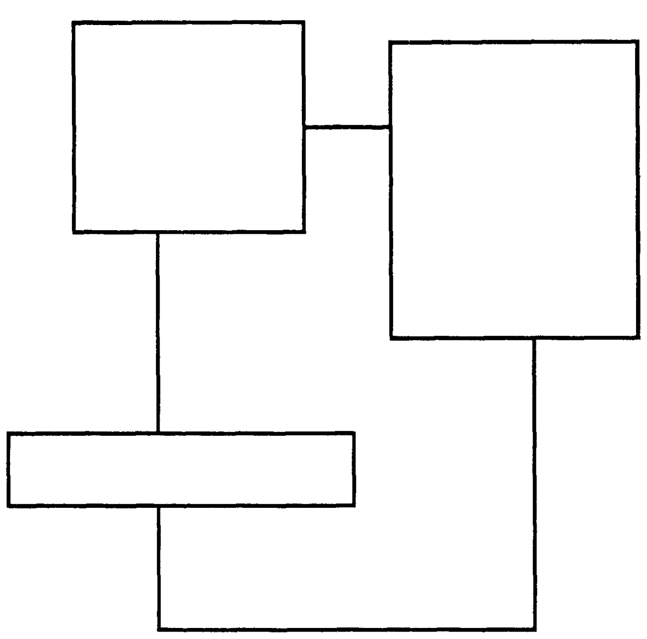
Fig. 1.3 -- The Multiple Pass Fragmentation algorithm (MPF).

fragments of the obscured window. This notion is key to the rest of the algorithm. Now, to determine what is to be displayed when a window changes, we simply subtract it from the window above it, subtract each of the resultant fragments from the next window up, and so on. Unfortunately, this is not always adequate. For instance, if a window is being buried, it is not enough to figure what part of it is showing, but we must also figure what parts of the windows above it are showing to properly bury it. This simply means that in the case of a window being buried, we must invoke the algorithm recursively on the window's children and on the windows lying above it in the hierarchy. A similar situation arises in the case of a window being surfaced; in this case, however, the algorithm need be recursively invoked for the changed window's children only.

How does MPF compare to DNA and DNASB? First, it does not allow the screen to flicker, so it is immediately a step ahead of DNA. Second, its storage consumption is limited to descriptors for window fragments instead of large display buffers as we saw in DNASB. But the major distinction between MPF and the other two algorithms is MPF's interesting performance characteristic: its performance degrades in proportion to the difficulty of the problem to be solved. The implication of this is that the user who desires a simple, non-overlapping display does not incur the same overhead as the user who wishes to have a complex display with lots of overlapping windows. By contrasting this property with DNA (where overlapping windows are not even supported very well) and DNASB (where the user must always have the storage and time overhead whether he uses overlapping windows or not), we see that MPF affords the user a great deal of flexibility. At its best, MPF performs as well as DNA does on non-overlapping windows. On average, it outperforms DNASB and sometimes even DNA on overlapping windows (see Appendix A for details).

Of course, there are two sides to every story, and MPF's story is no exception. To begin, there is a lot more code for MPF than there is for either of the others. The difference, however, amounts to only about ten per cent of the entire window system, so if one can afford JAWS without MPF then surely he can afford JAWs with MPF. The second problem is more serious and concerns the execution time of the MPF algorithm. Simply put, MPF is slow on large problems (compare with DNASB's constant performance). As Appendix A indicates, MPF begins to become slower than DNASB when the user tries to bury a window beneath about six others (this really depends on how the six are arranged on the screen; complex patterns tend to take longer than less complex ones (Fig. 1.4)).

If we plot the performances of MPF and DNASB versus problem complexity, we will see a point at which it is better to run DNASB than it is to run MPF. The optimal algorithm, then, would be able to analyze a window configuration according to some


(a)

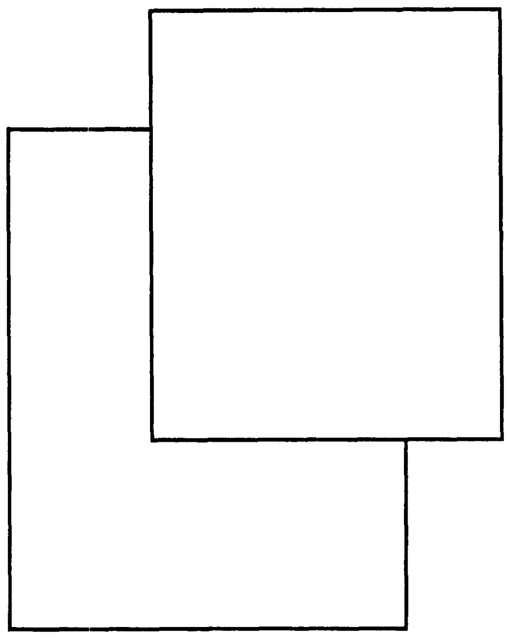

(b)

Fig. 1.4 -- Configuration (a) is relatively complex while (b) is relatively simple.

complexity criteria and then run either MPF or DNASB depending on where the configuration's complexity fell on the graph. Of course, then we would regain the problem of scrounging up enough memory to run DNASB in its worst case unless we took that factor into account while deciding which algorithm to run.

The perspicacious reader will no doubt have noticed that of the three screen management algorithms we implemented, none was the technique known as "double buffering" (i.e. having two complete bit-maps for the display and drawing on one while viewing the other). The reason we did not implement this solution is that one of our principal objectives was to reduce storage consumption; double buffering guarantees that we will use twice the storage normally consumed by the display.

# CHAPTER 2

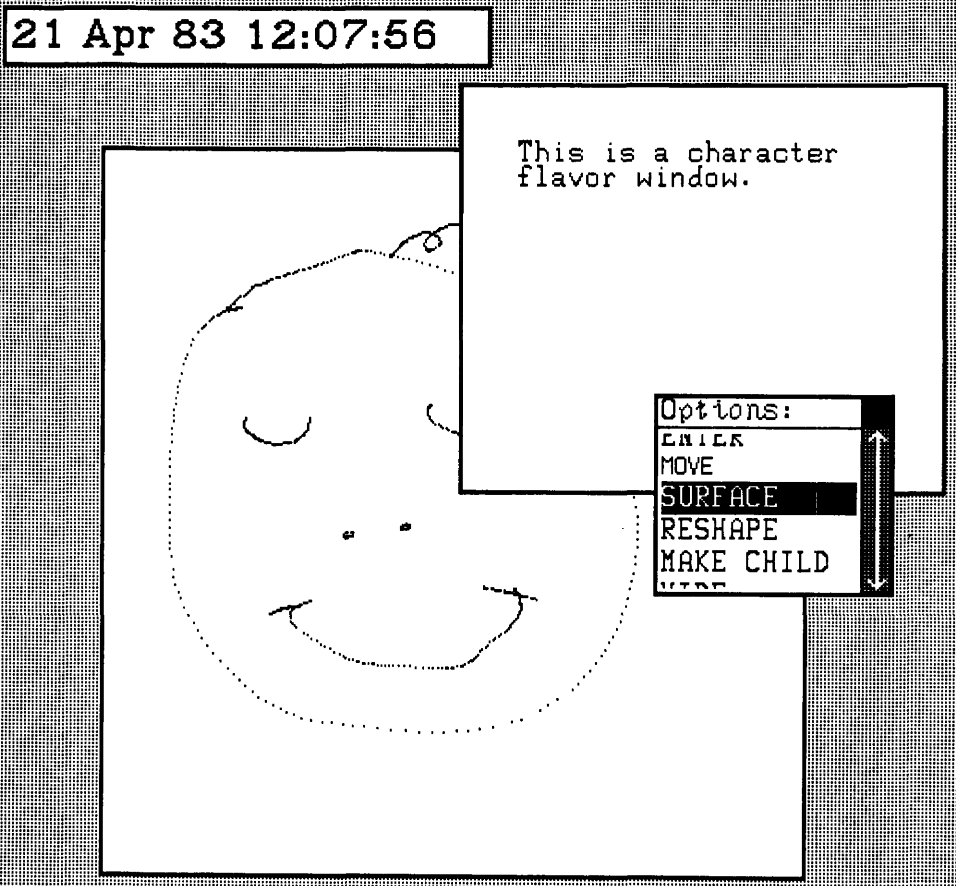
# POP-UP MENUS

This chapter is concerned with the implementation of a canvas flavor which supports the pop-up menu abstraction described below. Because of its function, we have chosen to name the flavor POPFLAVOR. This chapter is divided into two major parts: the first part refines the pop-up menu abstraction and the second part details the implementation of POPFLAVOR.

## WHAT SHOULD A POP-UP MENU DO?

A typical pop-up menu allows the user to select an entry by moving a pointing device (such as a mouse or a puck) into a position which is enclosed by the area of the desired entry. The menu indicates to the user that it recognizes a selection by highlighting the selected entry in some way (Fig. 2.1). To aid the user in browsing, the pop-up menu also provides a scrolling feature; the pop-up menu in Figure 2.1 has been partially scrolled by using the scroll bar on its right margin. The user then makes some finalizing action (such as pressing a button) to let the menu know that the selection is final. After the selection is finalized, the menu is hidden from the user and the entry selected is acted upon by the application program.

When we attempt to implement such an interaction as described above, we find ourselves faced with some pressing issues. How does the menu get its contents? Who is in control of the selection action: the application or the window system? If the window system is in control, how much information does it need to



This is a character flavor window.

Fig. 2.1 -- The pop-up menu indicates a selection by highlighting the selected entry.

know about the contents of the menu in order to support the selection activities? How is the selected entry related to the application (if the window system is in control of the selection)? What happens if there are more entries than will fit into a reasonably large screen area? What happens to the screen area that is temporarily obscured by the pop-up menu? How does the user indicate that he wishes to make no selection, but just wants the menu to go away? The following description of the implementation of pop-up menus addresses these and several other issues.

## IMPLEMENTATION DESIGN ISSUES

The first issue we must resolve is who is to be in control of the pop-up menu interaction: the application or the window system? Clearly, there are advantages and disadvantages to each. One may wish to have the application implement the pop-up menu if one desires a considerable degree of control over the features provided by the menu; if the window system provided a pop-up menu, there is a danger that the application programmer would have no choice about the fine details of its behavior. The disadvantage to having no window system provided facility is that each application would then have to create its own. This would be inefficient since functionally equivalent code would be have to be written and debugged for each application (unless there were some compatibility among applications that allowed modules to be shared, but then we open up a can of file system management worms). It seems, then, that a reasonable approach to the problem would be to have the window system provide a pop-up menu facility with application-defined options. Although this does not provide the flexibility of an application specific pop-up menu facility, it is more flexible than having one standard pop-up menu provided by the window system.

Once we have chosen to have the window system provide the pop-up menu facility, we must establish a method of communication between the application and the window system. First, how should the application specify the contents of a menu to the window system? Since the basic components of menus are entries, it seems logical that the application should specify a menu on a per entry basis. So, for each entry the application wants in the menu, it must specify the contents of that entry and a list of options (e.g. what font to use when writing that entry into the window). The window system would use this information to build an internal representation of the application's menu and give the application a name it can use to refer to that representation. To use the menu to get input from the user, then, the application would simply invoke the window system's routine for handling input from pop-up menus with the desired menu's name as argument.

The invocation of the pop-up menu input routine brings us to the next major issue: how is the input received by the window system specified to the application? There are two obvious ways of communicating the information. One is to return the number of the entry selected to the application. The disadvantage of this is it requires the application to maintain his own data structure of the contents of the menu and to keep that data structure consistent with the data maintained by the window system (although this is easy to do when pop-up menu is first defined, it may be troublesome if we allow the user to change the menu interactively). This method's advantage is that it does not require the application to parse the input. Parsing may not be much of a problem when the menu's entries are restricted to those defined by the application, but what happens when we place no restrictions on the user's ability to change the menu to include commands mixed with data instead of using several levels of menus to specify data? How can the application understand what the user is trying to do without performing some parsing?

The other way to communicate the selected entry is to return the string contents of the selected entry. This method has advantages and disadvantages which are the opposite of the entry number method's. Since either method is better under different circumstances, the best solution is to allow the application to specify in what form he wants the input to arrive.

## SPECIFYING THE IMPLEMENTATION

The following discussion on the implementation of POPFLAVOR is a general one whose ideas apply equally well to all JAWS flavor packages. POPFLAVOR is used here only as a medium for expressing those ideas.

Now that we know the inputs and outputs of the pop-up menu facility, we must decide how to implement something which adheres to that specification. Since we are implementing this as part of some window system, we should first decide how this feature fits into the rest of that system. In our case, the window system is JAWS; since, as was indicated in the introduction, abstract data types in JAWS are typically implemented by flavor packages, we should implement pop-up menus as a flavor (hence its name, POPFLAVOR).

As a flavor package, POPFLAVOR must do two things: 1) it must provide an interface to the application to allow it to manipulate POPFLAVOR canvasses and 2) it must provide routines to support applicable generic routines for the window system. Because the functions provided by a pop-up menu are so narrow, the application interface can be rather small. The major functions required by the application are the ability to insert an entry, the ability to delete an entry, and the ability to get input from the user through the pop-up menu (the application programmer may also desire functions which allow him to modify existing entries, but for the purposes of our discussion, these are ancillary to those functions listed above).

Supporting the functions required by the rest of JAWS (through the DISPATCH module as mentioned in the introduction) is a slightly more demanding matter. In addition to the somewhat trivial tasks of creating and destroying POPFLAVOR canvasses, the POP_UTILS module must also support the correlation of points on the APA screen with entries in the canvas and the translation of the canvas representation into a bit-map.

## THE IMPLEMENTATION OF A POP-UP MENU CANVAS

The implementation detailed below was developed with the above considerations in mind. The description given first specifies the data representation used and then proceeds to specify the algorithms used to perform the correlation and translation (these are the two most interesting problems associated with the pop-up menu canvas).

Since the application may make and delete entries at arbitrary points in the menu and the menu may grow to an arbitrary size, the data structure we choose must support these concepts easily. For this reason, the top-level structure of the canvas is a linked-list of entries (Fig. 2.2). The only disadvantage to this representation is the storage consumed by the pointers in the linked-list (on the PERQ, pointers consume two sixteen-bit words

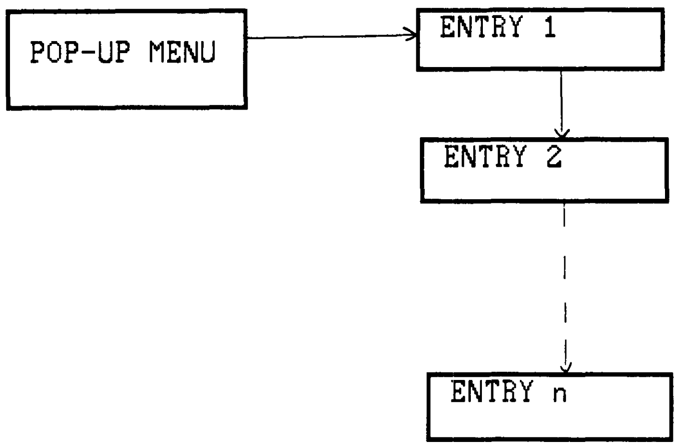


Fig. 2.2 -- The top-level structure of the pop-up menu canvas is a linked-list of entries.

each). There are two reasons why this is not a major issue in our case: 1) typical pop-up menus are on the order of ten to twenty entries (remember that they are supposed to make a more friendly interface, so they will probably be kept brief to avoid clumsiness) and 2) the PERQ has one million bytes of primary memory, so a few words wasted are not that important to us when balanced against the increase in efficiency we achieve. We are, however, forced to assume a more conservative attitude when presented with either a more demanding task (such as the multiple-font canvas presented in Chapter 5) or a smaller machine.

Next, we must specify what constitutes an entry. An entry (Fig. 2.3) consists of three major parts: the string form of the entry (TITLE) and the two attributes it can assume (FONT and SPACING). The FONT field determines what font the TITLE string is to be written in (note that although there can be several fonts per menu, there can be only one font per entry); the SPACING field determines how many bits of white (or background) space is to be left between consecutive entries or an entry and the window border.

Entries must also contain information to assist the implementing code. An example of such information is the forward pointer for the linked-list. There are several other such fields in the entry structure which we present along with the algorithms that utilize them.

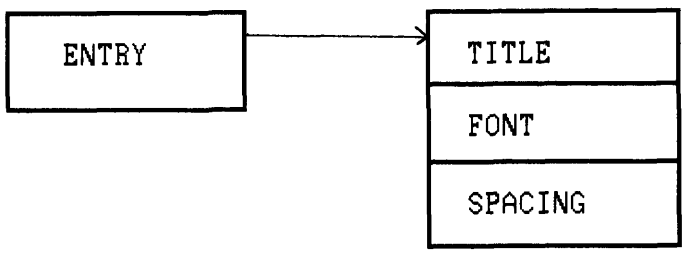


Fig. 2.3 -- An ENTRY is composed of three fields: the TITLE field, the FONT field, and the SPACING field.

We now consider the correlation and translation algorithms. Since the translation algorithm depends on the correlation algorithm, we present the latter first. The primitive functions needed to perform correlation are conversions from entry coordinates to bit coordinates (ENTRY2BIT) and vice versa (BIT2ENTRY). Once these primitives are established, it becomes easy to write the correlation routine (Fig. 2.4).

The real question, then, becomes how to write these two primitives. The simplistic answer is to start at the top of the canvas and count the bits consumed between the first entry and the entry of interest. This basic technique is the underlying principle for both ENTRY2BIT and BIT2ENTRY (Fig. 2.5). However, life isn't quite so simple. Were we to actually implement the primitives this way, we would find that the canvas selection process would be too slow to keep up with the user and that smooth scrolling would be out of the question. What can we do to improve the performance of this basic algorithm? The answer lies in remembering the answers to our previous queries (somewhat like dynamic programming). To implement this solution, we need to add a field (the BITS_CONSUMED field) to our entry structure which tells us how many bits from the top of the canvas each entry is. Now, ENTRY2BIT becomes trivial; we simply look at the BITS_CONSUMED field (Fig. 2.6). BIT2ENTRY still requires us to start at the top of the canvas and search down, but now we are merely examining the BITS_CONSUMED field and totaling our

```
function popcorrelate(window, winx, winy) : integer
  [cany <- bit2entry(window^.canvas,
                     window^.viewposy + winy - window^.topmarg)

   RETURN(cany)]
```

Fig. 2.4 -- The correlation routine written assuming the existence of BIT2ENTRY.

```
function BIT2ENTRY(canvas, bitcoor)
  [E <- { all entries in canvas }; tot_dist <- 0
   ∀ e ∈ E DO
     [tot_dist <- tot_dist + (the height of e's font) +
                   (the total spacing for e)
      IF tot_dist > bitcoor THEN (leave loop)]

   RETURN(e)]

function ENTRY2BIT(canvas, entrycoor)
  [E <- { all entries up to entry labelled entrycoor }
   tot_dist <- 0
   ∀ e ∈ E DO
     [tot_dist <- tot_dist + (the height of e's font) +
                   (the total spacing for e)]

   RETURN(tot_dist)]
```

Fig. 2.5 -- The primitive correlation routines: BIT2ENTRY and ENTRY2BIT.

```
function BITZENTRY(canvas, bitcoor)
  [E <- { all entries in canvas }; tot_dist <- 0
   ∀ e ∈ E DO
     [tot_dist <- tot_dist +  e.bits_consumed
      IF tot_dist > bitcoor THEN (leave loop)]

   RETURN(e)]


function ENTRY2BIT(canvas, entrycoor)
  [e <- (the entry corresponding to entrycoor)
   RETURN(e.bits_consumed)]
```

Fig. 2.6 -- BITZENTRY and ENTRY2BIT modified to use the BITS_CONSUMED field of each entry.

findings (Fig. 2.6) instead of calculating the bits used by an entry through finding and totalling its font height and total spacing.

We must remember to take care to be sure that the BITS_CONSUMED field is always accurate. There are two alternative methods for achieving this goal. One method would be to calculate the BITS_CONSUMED field for each entry in the menu upon menu creation and whenever a new entry is inserted in the middle of the menu or deleted from the menu (these operations affect the bit coordinates of subsequent entries because those bit coordinates are absolute, not relative, values). This method has the disadvantage of making insertion and deletion of menu entries more expensive; we recognize, though, that these operations will probably be done rather infrequently. The alternative is to maintain another field (called COMPUTED) which indicates whether or not the BITS_CONSUMED field is accurate. The ENTRY2BIT and BIT2ENTRY algorithms would be modified, then, to first see if the BITS_CONSUMED field for the concerned entry is accurate. If it is, then the algorithms would proceed as described above; if it is not, then the algorithms would calculate the correct value, insert it in the BITS_CONSUMED field, and use the COMPUTED field to mark it as being accurate. This method has the disadvantage of making operations unpredictably slow at times. Which method is preferable depends on where the application programmer prefers to incur the overhead associated with this technique.

Once we have the correlation primitives, we are ready to write the other major routine: the translator (Fig. 2.7). Recall that the mission of the translation routine is to take the canvas representation of a pop-up menu and convert it into a bit-map. The first step we must take toward performing the translation is to decide what portion of the canvas to translate. We get this information from one of two sources: Either the caller of the routine tells us what portion of the canvas to translate or we can determine what to translate by looking at the changed coordinates written by the mutating routines (i.e. the routines which change entries). Next, we must determine what portion of the translation's target buffer (this is typically a screen buffer) is affected by the change in the canvas. The call to INFLUENCED in Figure 2.7 performs this task. Once we know the bounds of the translation (in bits), we simply write the TITLE field of each entry within those bounds into the target buffer using the font indicated by the entry's FONT field and leaving as many bits of space as are requested by the SPACING field.

## HANDLING INPUT

The actual handling of user input is made trivial by the power of the primitives we have defined. The input routine (called

```
procedure POPTRANS(window, canvas, chg_coords)
  [IF chged_coords = {} THEN
     [chged_coords <- { changed coords on canvas }]
   chged_coords <- influenced(window, chged_coords)
   IF chged_coords ≠ {} THEN
     [∀ c ∈ chged_coords DO
        [( write the canvas entry corresponding to c in window at
           ENTRY2BIT(canvas, c) with proper font and spacing )]]]
```

Fig. 2.7 -- The translation routine is easy write using the ENTRY2BIT primitive.

POP_PICK_N_CHOOSE) is simply a loop (Fig. 2.8) which fetches the coordinates of the user's pointing device, correlates that point with a point in the canvas, checks for any button presses, and does the appropriate thing (highlight an entry, scroll, terminate, etc.). This routine is also given responsibility for placing the pop-up window upon entry and hiding it upon exit. These functions are easily implemented using the window system primitives. Highlighting is achieved by changing the highlight attribute of the selected entry and invoking the translator to transfer the change onto the screen; scrolling is achieved by changing the window's view on the pop-up menu and invoking the translator; popping the pop-up window onto the top of all other windows on the screen is simply a SURFACE operation; making the window go away when the user is done is simply a HIDE operation (recall that JAWS supports overlapping windows, so refreshing the area which is temporarily obscured by the pop-up window is done automatically by the window system).

One remaining issue is how to terminate the loop in POP_PICK_N_CHOOSE. We chose to terminate when the user presses a button while pointing outside the pop-up menu's window (Fig. 2.8). There are, however, alternatives to this strategy. If there are several buttons available to the user, the application programmer could designate one of them as the terminating button. We could extend this notion to include all keys on the keyboard (we should be careful here to remember that we use pointing

```
function POP_PICK_N_CHOOSE(window)
  [surface(window)
   REPEAT
     [I <- ( pointer position in bits )
      C <- BIT2ENTRY(window^.canvas, I)
      IF C ∈ { all entries in window^.canvas } THEN
        [highlight(C)]
      ELSE
        [IF ( I is in scroll region of window ) THEN
           [(change view of window on canvas and call POPTRANS )]]]
   UNTIL ( button is pressed )
   hide(window)
   RETURN(c)]
```

Fig. 2.8 -- The input handling routine for pop-up menus.

devices to get away from using the keyboard so much). Finally, we could designate an area within the pop-up menu's window to be the termination area. All of these alternatives have one thing in common: they free one expressive resource (e.g. a window area or a button) at the expense of tying up another. Although there is no one paradigm that is correct for all situations, it is certain that the window system should allow the application programmer as much freedom as possible in specifying such features. Such a perspective is consistent with our desire to provide the application programmer with the convenience of a pop-up menu facility that is built into the window system while not constricting his ability to specify what features that pop-up menu facility possesses. Once again, these principles are applicable not just to POPFLAVOR, but to all flavors we design.

# CHAPTER 3

# DYNAMIC REDISPLAY

The original implementation of JAWS was flexible enough to support windows whose canvasses were their own screen buffers (Fig. 3.1). This arrangement had the advantage of saving the extra storage that would have been taken up by a bit-mapped canvas of the same size as the window's screen buffer (note that this notion is applicable only to windows which view bit-mapped canvasses since screen buffers are bit-mapped). The disadvantages, of course, are that any changes to the canvas become immediately apparent in the screen buffer and such canvasses must be the same size as their viewing window's screen buffer. This concept has a limited scope of application, but we did find it to be useful for windows which contained static images, such as a window containing a background pattern for the screen (Fig. 3.2).

An analogous concept which was not exploited in the original implementation of JAWS is that of windows which view canvasses, but have no screen buffer. Such an arrangement is similar to the default windows on the LISP machine (Weinreb and Moon). Its principal advantage is the saving of the space that would have been used for the screen buffer's bit-map; its principal disadvantage is that sometimes it may be difficult to regenerate a window's screen image from its canvas quickly enough to support the application. This concept, too, has a limited range of application and is useful mainly for windows which view patterned canvasses since the regularity of patterns makes it easy to

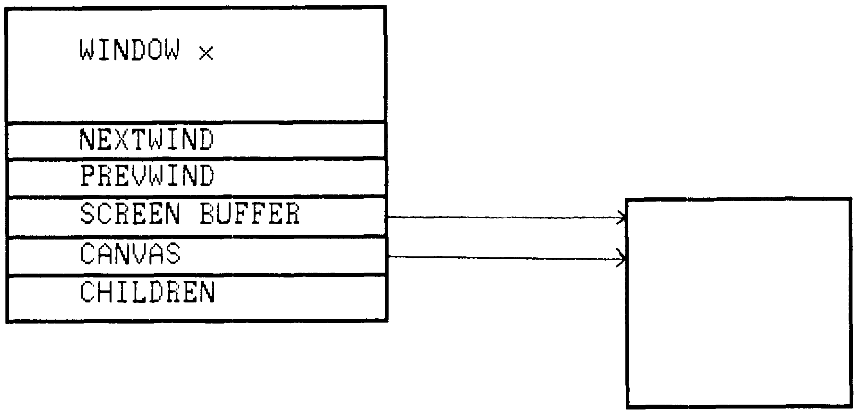


Fig. 3.1 -- A window's screen buffer may also serve as its canvas.

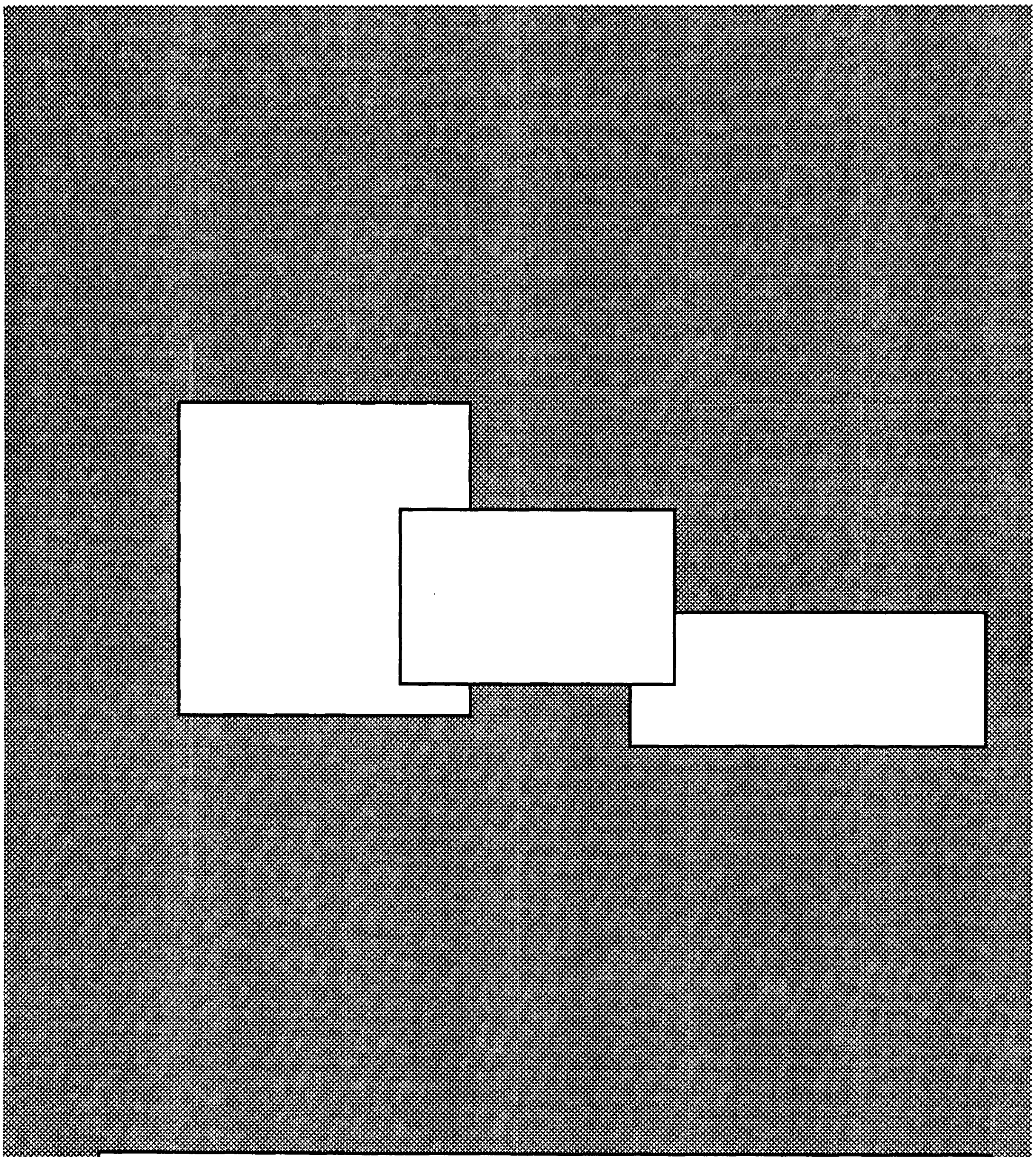


Fig. 3.2 -- The background pattern is implemented using a window whose canvas is its screen buffer. This implementation uses half the storage that the same window would use were it implemented normally (i.e. with separate canvas and screen buffer).

generate them dynamically. The following section describes the use of this concept for a more space-efficient implementation of the background pattern of the screen mentioned above.

Since flavor construction is discussed in Chapter 2, I assume here that the reader is familiar with the basic concepts involved in flavor construction and, therefore, concentrate here on those features of this flavor (termed BACKFLAVOR since it is used primarily for the generation of background patterns) which are different from the analogous features of "normal" flavors (such as CHRFLAVOR).

BACKFLAVOR's major different feature is that it performs the translation of data from the canvas representation into the bit-mapped representation directly into the display's buffer. This differs from the normal flavors which perform the translation into the window's screen buffer. Because this feature makes the translation visibly apparent to the user, the translator must perform both quickly and gracefully (i.e. it should not leave a mess on the screen en route to completing the translation).

As an aside, we note that it may be desirable to have all of the flavors provide translator routines in which the target buffer is parametrized. This would enable us to have bufferless windows which can view any canvas instead of limiting us to using buffered windows with all normal flavors and bufferless windows with BACKFLAVOR. The routine which displays windows on the screen, then, would have to be modified to first check the window to see if it is buffered. If it is, the routine would display it on the screen just as it does today; if it is not, the routine would call the generic translator routine with the display's buffer as its target parameter. This arrangement is clearly superior to what presently exists in JAWS. The only reason it was not implemented is that the present structure of having flavors determine what the target of their translation will be is so thoroughly built into JAWS that performing the modification would have consumed an unacceptable amount of the author's time.

As mentioned above, the translation routine must have the property of being both fast and graceful. These goals are particularly easy to achieve in our case since we are trying to produce a constant pattern. Since the pattern is repeated every four pixels in each direction, we can represent the entire pattern as a four pixel by four pixel seed. The translation routine, then, would consist of replicating this seed throughout the target area. To do this, we employ an algorithm developed by J. Gonzalez which expands the pattern geometrically throughout the target area first horizontally and then vertically (Fig. 3.3).

Because our translator must be compatible with algorithms that seek to display only part of a window, the algorithm we use in

```
PROCEDURE make_pattern(pattern, width, height, area)
  [offset <- (width of pattern)

    WHILE offset < width DO
      [if 2 * offset < width then
          rem <- offset
       else
          rem <- width - offset

       (draw pattern on area at offset,0 in a rectangle of
        height equal to pattern height and width equal to rem)
       offset <- 2 * offset]

    offset <- (height of pattern)

    WHILE offset < height DO
      [if 2 * offset < height then
          rem <- offset
       else
          rem <- height - offset

       (draw pattern on area at 0,offset in a rectangle of
        height equal to rem and width equal to width)
       offset <- 2 * offset]]
```

Fig. 3.3 -- The algorithm for spreading a pattern across a rectangular area.

BACKFLAVOR's translator must be able to match the existing pattern at any given point in the window without causing a visible seam to appear. To do this, we must enhance Gonzalez' algorithm as shown in Figure 3.4. The idea behind the enhancement is to calculate where we are in the window and use whatever component of the seed is appropriate for matching the pattern at that point in the window.

Since BACKFLAVOR has been successfully integrated into JAWS, we have been able to compare its performance to that of the more traditional implementation of the background pattern. We have found that for the most part it does not adversely affect the performance of JAWS programs. The only time it makes a noticeable difference to the user is when he is dragging windows across the screen. Because the background is being filled dynamically behind the dragged window, the dragging process is noticeably, but not unacceptably, slower. In most cases, the space we gain is worth this small loss in time; in the case of the background pattern, our previous storage consumption of 96K bytes has been reduced to one word.

# CHAPTER 4

# CHILD WINDOWS

This chapter compares the relative advantages and disadvantages of having child windows share the screen buffers of their parents and having child windows maintain their own screen buffers. Before we proceed to compare the two implementations, we must define what a child window is.

A child window (Fig. 4.1) is a window defined within another window. The window which contains a child window is referred to as being the child's "parent". The implications of this are twofold: (1) The child window cannot be larger than or move outside of its parent and (2) whenever the parent moves, the child must both move with it and maintain a constant offset within it (Fig. 4.2). Recall from the introduction that children can view canvasses separate from their parents' and can have children of their own (so several levels of children are possible). We termed a window with no parent to be a "top-level" window.

The original implementation of child windows did not allow children to have their own screen buffers; instead, they shared their top-level ancestor's screen buffer (Fig. 4.3). This meant that when the canvas of a child window was being translated, the target buffer of the translation was the child's top-level ancestor's screen buffer. We lose in several ways here. First, all of the window management routines which have to handle screen buffers must become more complex to properly handle child windows in addition to top-level windows. Second, we lose the ability to

```
PROCEDURE backtrans(window, qrec)
  [(qrec carries information about the area to be translated:
       modposx = starting x coordinate
       modposy = starting y coordinate
       modwid  = width of area
       modhgt  = height of area)

   IF (modposx mod 4 = 0) AND (modposy mod 4 = 0) THEN
      make_pattern(window^.canvas, modwid, modhgt, window^.buffer
   ELSE
     [x_off <- modposx mod 4; y_off <- modposy mod 4;
      rem_wid <- (window's canvas' width - x_off)
      rem_hgt <- (window's canvas' height - y_off)

      (use x_off, y_off, rem_wid, and rem_hgt to draw the
       odd part of the area to be translated; what's left is
       something that make_pattern can handle)

      make_pattern(window^.canvas, modwid - rem_wid,
                   modhgt - rem_hgt, window^.buffer)]]
```

Fig. 3.4 -- The translation routine for BACKFLAVOR must be able to match the pattern stored in a window's canvas to any location in the window's buffer without leaving a seam.

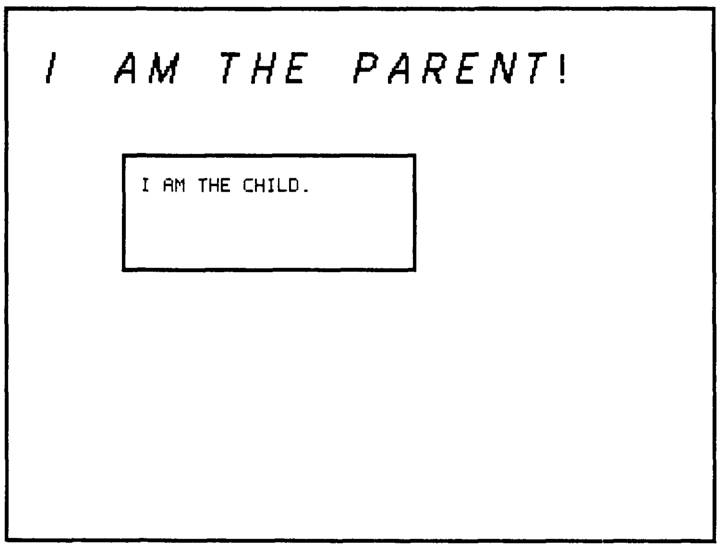


Fig. 4.1 -- An example of a parent-child relationship. Note that the child is viewing a canvas which is different from that viewed by the parent.

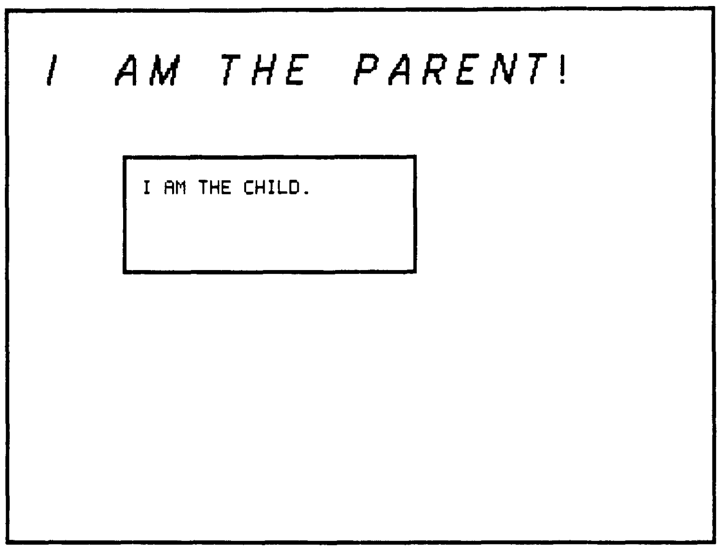


Fig. 4.2 -- The child window both moves with and maintains its offset within the parent window when the parent is moved.

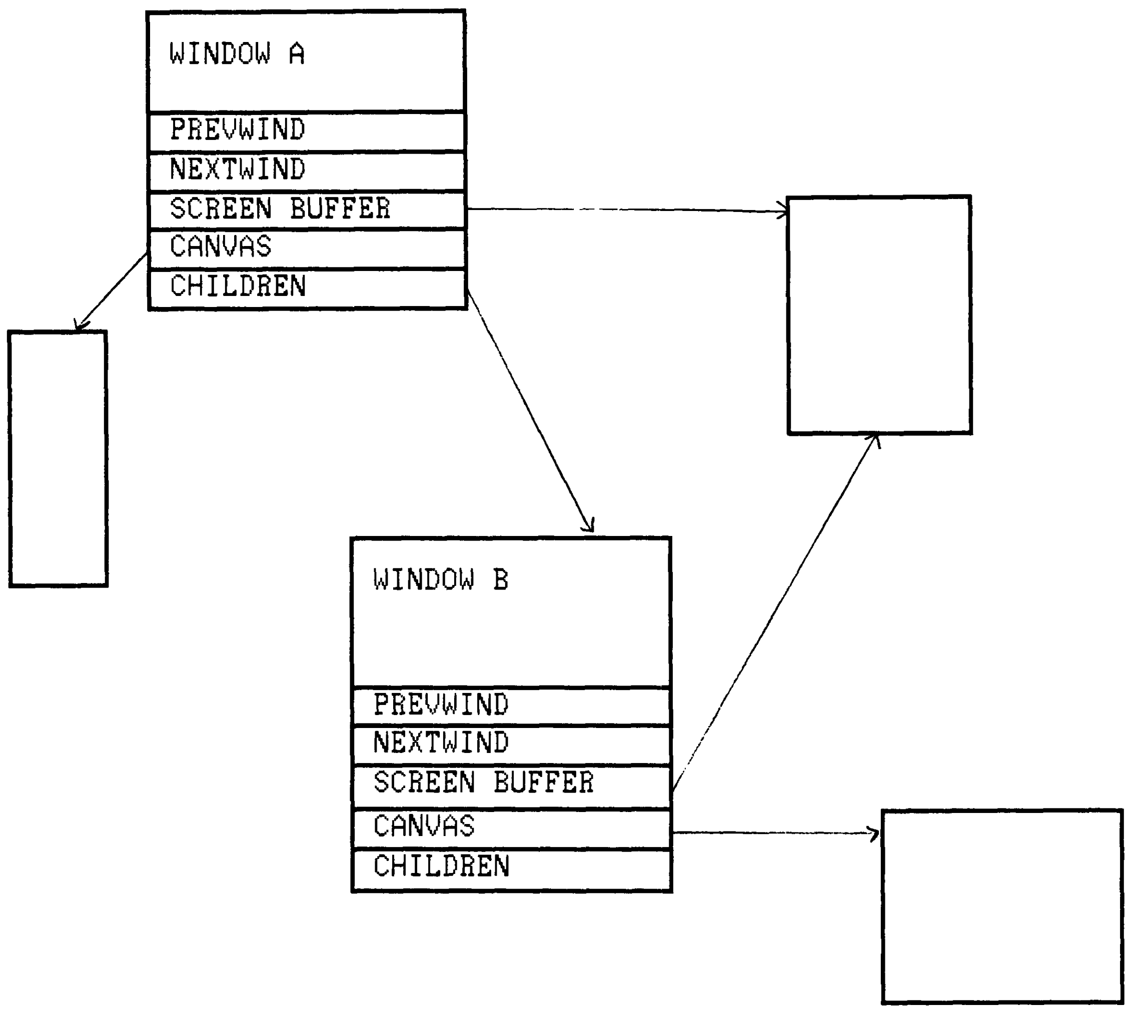


Fig. 4.3 -- The original implementation of child windows forced children to share screen buffers with their parents. Here, WINDOW B is the child of WINDOW A and as such must use its screen buffer Note that WINDOW B does, however, view a canvas which is different from that viewed by WINDOW A.

have overlapping child windows since the overlapping feature inherently requires some buffering. Finally, unless we are willing to add yet more special-case code, we lack the ability to update a top-level window with children without destroying the children's screen images. Another important disadvantage of this implementation is that it forces the perversion of the meanings of some of the window operations. For example, the bury operation cannot be performed on a child window since a child has no screen buffer. Therefore, if the bury operation is invoked on a child window, it will result in the burial of that child's top-level ancestor.

The alternative to the above implementation is to give each child window its own screen buffer. This allows the window hierarchy to become fully recursive, eliminates special-case code, and keeps the window management operations true to their definitions. The only disadvantage to this implementation is its potential inefficiency. Depending on the architecture of the underlying machine, allocating a screen buffer may consume valuable resources. Since child windows are often used to divide a larger window into several small regions, allocating a screen buffer for each child may be unacceptably inefficient in some cases.

From a systems design perspective, then, the second implementation is much more appealing than the first. The first implementation, however, has its characteristics for some very good reasons (as we have seen). How do we resolve this conflict? Some obvious solutions are: use either the first or the second implementation and accept their shortcomings; give the application programmer a choice between the two implementations for each child he makes; employ some intelligent method of determining whether or not a screen buffer is necessary on a child by child basis (this is essentially equivalent to what the programmer would do in the previous proposal only now it is done automatically and without the programmer's knowledge). We can immediately eliminate the first proposal since we don't want shortcomings in our system. The second may do as an easy fix, but it is not really desirable since we don't want the application programmer to have to know details about the window system's implementation. The last proposal is interesting as an artificial intelligence problem, but still leaves us with the special-case code problem and, potentially, the segment use problem. It does, however, preserve the meanings of the window management operations and allow the implementation details of the window system to remain hidden.

Our work in this area went only as far as implementing child windows with their own screen buffers. Perhaps future researchers will develop techniques for implementing the more intelligent solution outlined above.

# CHAPTER 5

# BUILDING A BRIDGE BETWEEN POLITE AND JAWS

This chapter is concerned with the development of an interface between POLITE (a real-time editor/formatter) and JAWS. POLITE was originally implemented on the IBM 370 under CMS (Borkin and Prager). We are now in the process of transporting it from the 370 to the PERQ minicomputer where it will require some window management system to allow it to make use of the APA capabilities of the PERQ without being substantially redesigned. A good way to model the POLITE system is as a powerful text manipulating device which has no way to control an APA screen. For that task, it needs a window management system such as JAWS (Fig. 5.1).

Since there are two major parts to POLITE, the user interface manager (referred to in ref. as the screen manager) and the document manager, we first consider how features of the POLITE user interface are supported by JAWS and then discuss the design of the interface between the manager of POLITE's internal document representation and JAWS' display mechanisms. The material presented here is not intended to be a recipe for making an editor/formatter. Details about POLITE are mentioned only when they affect its interface to JAWS.

## THE POLITE USER INTERFACE

The POLITE user interface manager is responsible for monitoring user input and performing the correct actions (actions include both screen manipulations and document manipulations) to execute the user's commands. The screen

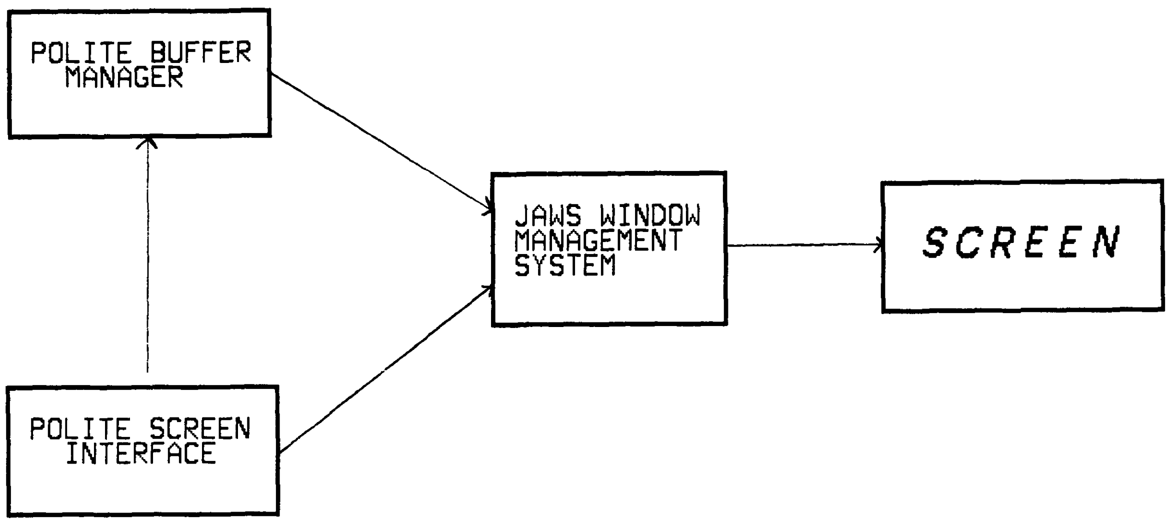


Fig 5.1 -- POLITE needs to use JAWS to interact with the APA screen. The screen interface and the buffer manager each use JAWS for different purposes.

interface relies on JAWS for both input support and window management functions as discussed below.

## ENTERING USER REQUESTS INTO POLITE

User requests are related to POLITE in one of two ways. The user either enters the command via the keyboard in some command area or he selects some command from a menu. In both cases, the result of the user action is that a command string is presented to the POLITE parser, parsed, and finally executed by the POLITE command interpreter. The syntax for the commands on the menu is exactly the same as that for those entered into the command area. This leads naturally to allowing the user to modify and create menus containing the commands (complete with arguments) he desires. If a command is entered but requires more arguments than those which were given, the POLITE parser will wait to allow the user to enter more arguments before it terminates the parse. This prevents the user from being forced to retype an entire command just because he forgot an argument or entered an invalid argument. This also allows the user to put partially complete command strings in the menus he creates, thereby increasing that feature's flexibility.

## HOW POPFLAVOR SUPPORTS THE POLITE MENU SYSTEM

The POPFLAVOR canvas (Chapter 2) in JAWS supports the POLITE menu system in several ways. First and foremost, it supports the dynamic creation and modification of pop-up menus. This facility allows POLITE to support the menu change feature with very little code and little data structure complexity (since JAWS handles all of the messy details of managing the menu data structure). A second way in which POPFLAVOR supports the POLITE menu facility is by giving the application a name for each menu it creates. This simplifies the implementation of a POLITE facility for allowing the user to name menus and subsequently request them by name. Finally, by allowing menu entries to have multiple fonts, POPFLAVOR gives the POLITE menu system increased flexibility. For example, POLITE can use multiple fonts to indicate graphically the difference between the command part of an entry and the argument part.

## POINTING AT POLITE OBJECTS

Some commands are of the class referred to as the "pointing commands". This class includes such commands as "move" and "delete". The reason they are called pointing commands is that they require the user to indicate (or "point at") the object, or unit of text, he wants to move or delete. Objects can be anything from the entire document to a single character. The CMS implementation of POLITE is compatible with the 3270 terminal family and uses cursor movement keys to allow the user to point at objects. The PERQ implementation should be able to support pointing via an APA device such as a tablet or mouse.

## MAPPING SCREEN COORDINATES TO CANVAS COORDINATES

Correlation of points on the screen (i.e. bit coordinates) with pieces of text (i.e. character coordinates) is handled by the JAWS window manager. This allows POLITE to quickly and easily find out what window the user is pointing at and what character within that window he has selected. A technique to correlate this coordinate with a point in the internal document representation is discussed below.

The canvas we propose to use to support POLITE text (called TXTFLAVOR) also has a facility for marking blocks of text as being selected. This facility is powerful enough to allow the extent of a selection to be easily indicated, yet is still low-level enough to allow the rules governing the determination of the extent of the selection to be left up to the application (in this case, POLITE). POLITE uses this selection capability to determine the arguments for the pointing commands.

## HANDLING KEYBOARD INPUT

The question of how to determine the extent of a selection can be solved in a straightforward manner. A related but more difficult question is how to handle keyboard input.

We normally modify the contents of an editor buffer in two different modes: "insert" mode and "replace" mode. Insert mode allows new text to be entered without erasing but relocating existing text; replace mode allows existing text to be modified without being relocated.

The actual placement of the entered text onto the screen is no problem: we simply determine the cursor location, place the character into the canvas at that point, and let JAWS update the screen as it normally does after a canvas changes. The major problem we face is in determining where in the buffer the entered text belongs. For this purpose, there must be some manager within the POLITE system which maps canvas coordinates into buffer coordinates (Fig. 5.2). More detail about this manager is given below. Given that we have such a device, we can easily create temporary buffers to hold the user's keystrokes along with their ultimate destinations until we decide to enter the text into the buffer.

A related issue is determining how the screen appears to the user as he types. In replace mode, the user should see the character he enters appear in place of and in the same attribute as the character beneath his cursor (assuming that all fonts used

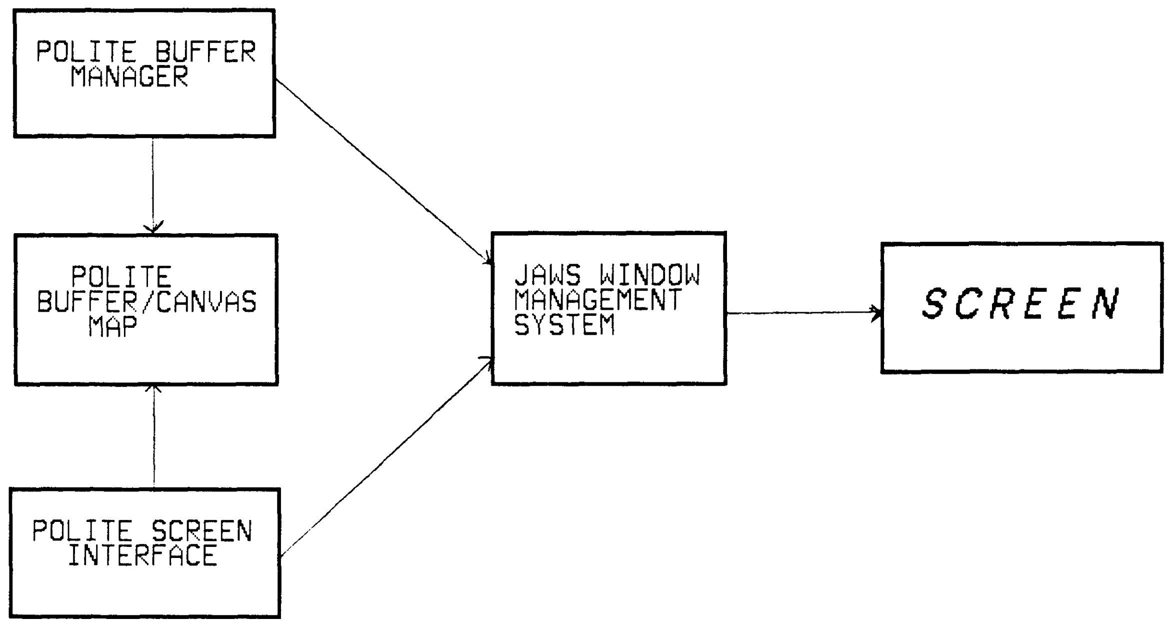


Fig 5.2 -- The buffer/canvas map is used by the POLITE screen interface to correlate canvas coordinates with buffer locations. The POLITE buffer manager fills and updates the map as is necessary when it updates the JAWS canvasses.

are fixed-width); TXTFLAVOR's TXTCHAR command supports this. In insert mode, the user should see the character he enters appear beneath the cursor and the character that was previously beneath the cursor move to the right. But what happens to the character at the end of the line? One solution is to wrap that character around to the next line. Since this may cause that line to spill also, this process must be repeated for each line in the canvas.

This approach works well for fixed font canvasses (provided they contain a relatively small amount of text), but is not appealing for a multiple font canvas for the following reason. A multiple font canvas must be laid out bit-by-bit to have all of its lines be justified; spilling a character from one line to another may very well upset this justification. The solution we chose is to have TXTINSERT (the character insertion routine) work only if there is room on the current line for one more character. If there is not, TXTINSERT indicates this to the application. The application then has the option of opening more space in the canvas by using TXTBRKLINES. This routine simply inserts just the right amount of space in the canvas to cause the character beneath the cursor to appear one line lower. This allows new characters to be placed into the canvas without disturbing the previous justification of the text.

## SUPPORTING THE DOCUMENT MANAGER

The POLITE document manager's primary need is to have the screen understand the characters it puts out. The JAWS multiple font text canvas (TXTFLAVOR) provides the means for translating character/attribute combinations into bit patterns. To see how, let's look more closely at the document manager's operation.

## THE BUFFER STRUCTURE AND OPERATION

POLITE text is stored as a hierarchical structure to minimize the work done to effect a format change. To produce the proper text layout from this representation, a formatter interprets the representation into what one can model as a stream of character/attribute combinations, which are simply character codes combined with attribute control information (e.g. font, color, underlining, spacing). The receiver of this stream is whatever TXTFLAVOR canvas is viewing the part of the document being formatted (there can be more than one). The idea of canvasses viewing other data structures is somewhat alien to JAWS, but is really no different from that of a window viewing a canvas. In this case, the information relating canvasses to documents is maintained by POLITE; the window system knows only that it is receiving data from its application.

The stream model discussed above is somewhat inefficient in that it requires each character to have a full set of attributes associated with it specifying how it is to be printed. A more efficient model is the "ink" model, where we choose an ink color (i.e. attribute specification), write some characters in that color, change colors, write more characters, and so forth. This takes advantage of the similarity of adjacent characters.

TXTFLAVOR supports the ink model with the TXTCHGATTR routine which is used to change the attributes in effect starting at the present location within the canvas and extending for the indicated number of characters. Characters are written using the TXTCHAR routine, which needs to know only the character to be written and the spacing to be left between it and its neighbors. TXTCHAR determines what attribute to assign the character by looking at the attributes in effect in the rest of the canvas.

## MAPPING A CANVAS COORDINATE INTO A BUFFER COORDINATE

We have solved the output problem quite handily, but we still must handle text input by directing it to the correct spot in the correct document. This problem was touched upon several times in previous sections, but cannot be solved fully without considering its implications.

As was indicated above, it is fairly easy to map a screen coordinate into a canvas and a character offset within that canvas. The remaining task is mapping that canvas/offset pair into a document location. For this we need some table which provides information on what part of what document each canvas views and what document locations selected points in each canvas correspond to. Armed with this information, we can quickly calculate the exact document/document-location pair which corresponds to the canvas/offset pair we received from the JAWS correlation routine. Since this table would have to be updated whenever a canvas is being filled or modified, the responsibility for maintaining the table should lie with the formatter. This should be quite easy for the formatter to do since it has all the information necessary at hand when it is filling a canvas (i.e. it knows where it is in the document and where it is in the canvas being filled).

Using the above technique, we can monitor the user's actions and always know the exact location he is indicating or acting upon. We can then buffer his character input and at the appropriate time place it in the correct document at the correct location. The phrase "appropriate time" is determined by the user interface portion of POLITE and the issues associated with it which were discussed previously.

## THE BENEFITS OF USING JAWS WITH POLITE

The APA capability of the PERQ greatly enhances the user interface by allowing more rapid user pointing, more flexible menus, easier menu selection, etc. But more importantly, the power of the JAWS window management features makes POLITE's code on the PERQ simpler than its code on CMS even though its function is greater. Furthermore, we must realize that JAWS does not make any special concessions to POLITE. The features that are useful to POLITE are useful to other applications as well as is demonstrated by the existence of the 3270 emulator (O'Hara) and the graphic editor mentioned in the introduction.

# CONCLUSION

The techniques and ideas described in this thesis are useful not only for JAWS, but also for other window management systems. The author hopes that the efforts spent on this research will not have to be repeated by future window system designers, but instead will be used as a basis for developing new ideas.

One area for future work is the implementation of the multiple font formatter design outlined in Chapter 5. Such work should provide a fairly stringent test of JAWS usefulness.

Another area for future research is moving JAWS to different machines. This will reveal how machine-independent the JAWS interface really is. We strongly suspect that it will not be very difficult to make a JAWS application run successfully on different machines once they are running JAWS.

Other areas which we might investigate include support for a general-purpose graphic flavor and support for non-character input (e.g. handwriting).

# APPENDIX A

# DISPLAY ALGORITHM RESULTS AND ANALYSIS

## Appendix A (Display Algorithm Results and Analysis)

Table A1.1 contains the running times of four different test application programs each of which were run three separate times using the three different screen management algorithms described in the text (see Chapter 1). The results are adjusted so that MPF always runs in time one while the others take either less than or greater than time one depending on their relationship to the MPF time.

The following is a description of each of the four application programs used in the testing :

1. WINTEST1 surfaces eight windows and then buries them in reverse order. Figure A1.1 shows the display after WINTEST1 has completed its surfaces, Figure A1.2 shows the display midway through the bury operations, and Figure A1.3 show the display at the completion of the bury operations.

2. WINTEST3 surfaces two overlapping windows and moves the bottom window across the screen (Figs. A1.4, A1.5).

3. QUIXTEST surfaces two overlapping windows and draws lines into both windows simultaneously at the rate of several lines per

| PROGRAM | DNA | DNASB | MPF |
|---|---|---|---|
| WINTEST1 | .95 | .96 | 1 |
| WINTEST3 | .82 | 1.23 | 1 |
| QUIXTEST | 1.43 | 1.22 | 1 |
| WINTEST4 | .76 | 1.07 | 1 |

Table A1.1 -- Running times of four test programs under the three different screen management algorithms. Note that the running time under MPF is normalized to one.

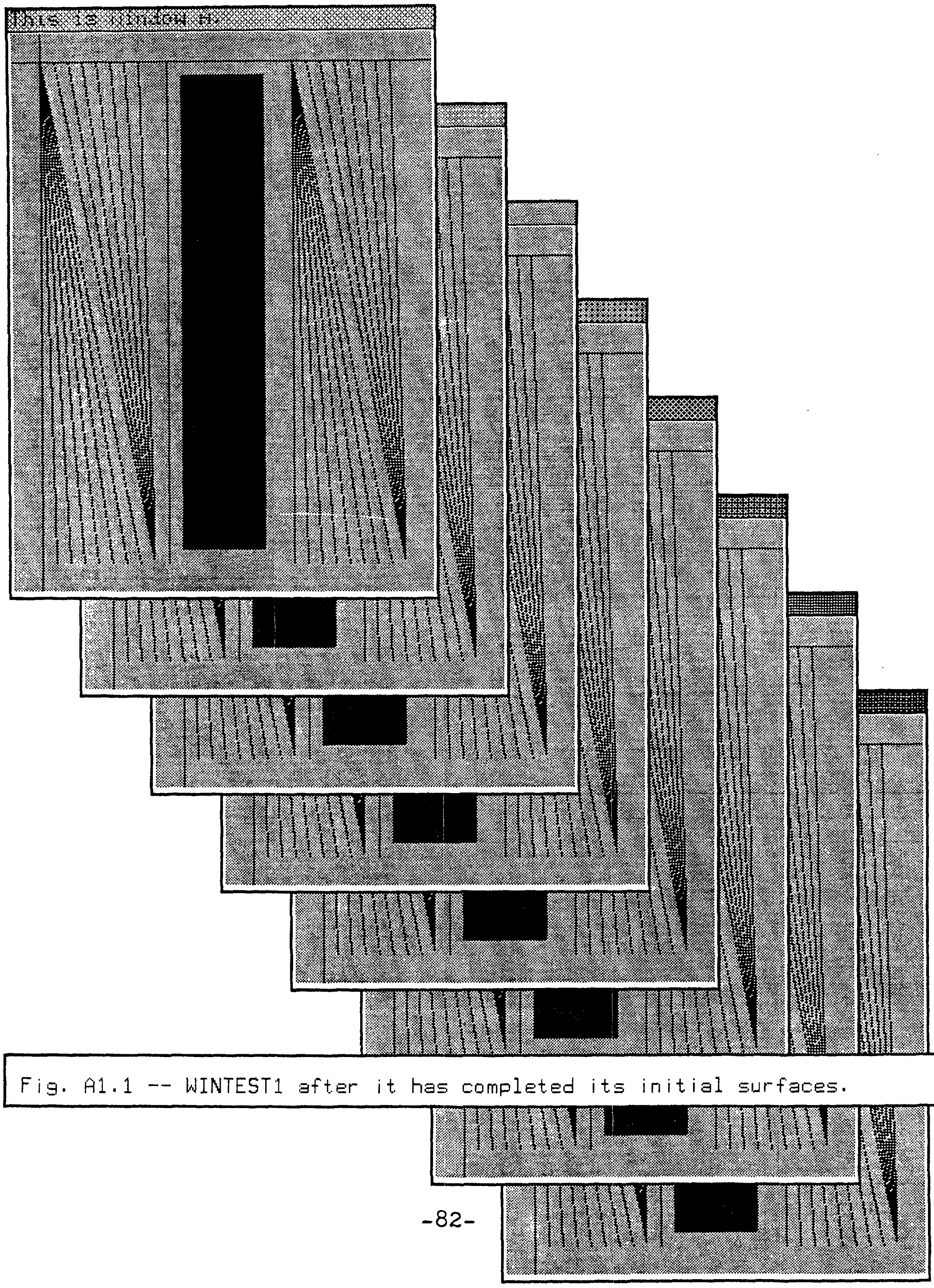


Fig. A1.1 -- WINTEST1 after it has completed its initial surfaces.

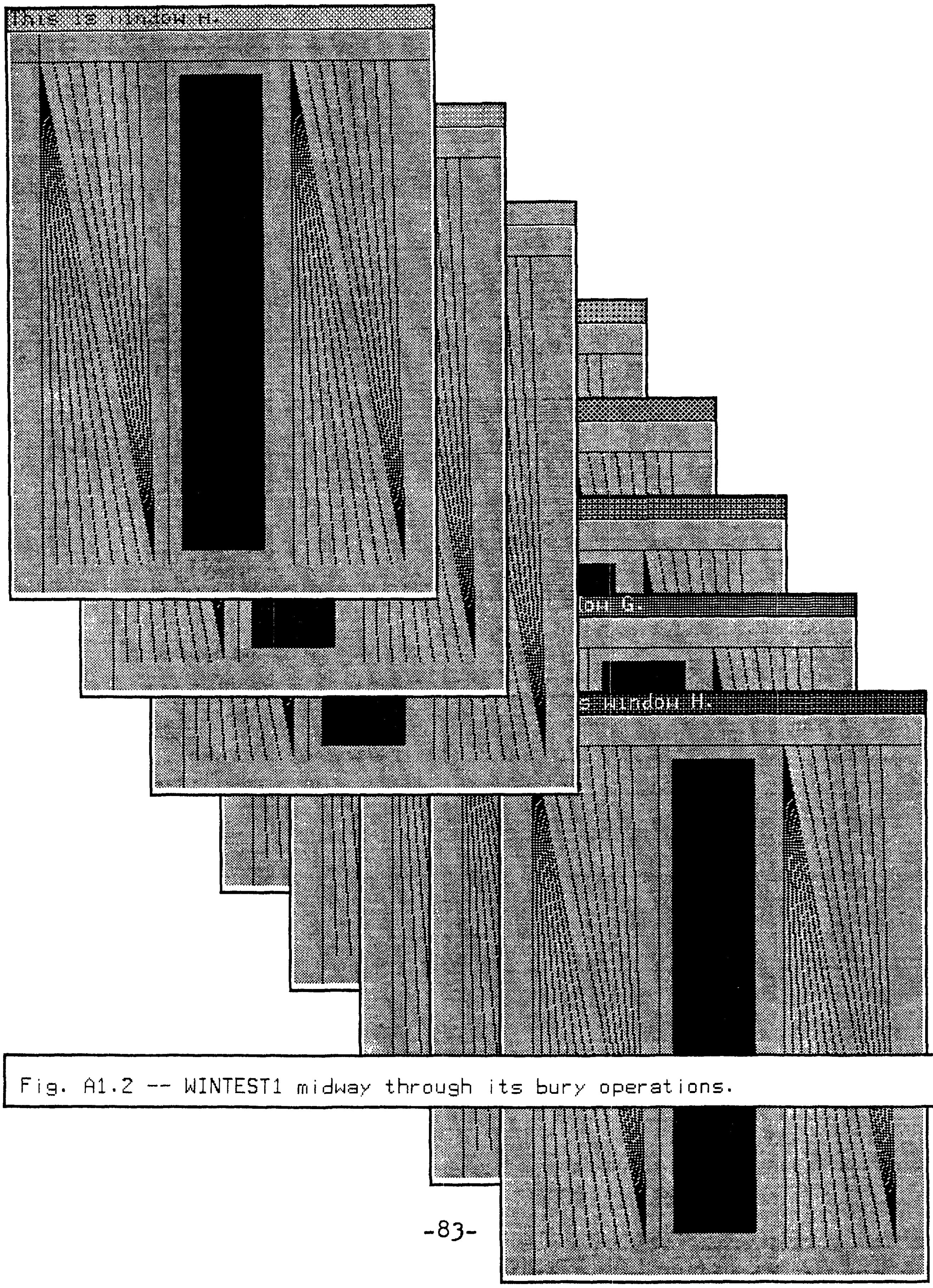


Fig. A1.2 -- WINTEST1 midway through its bury operations.

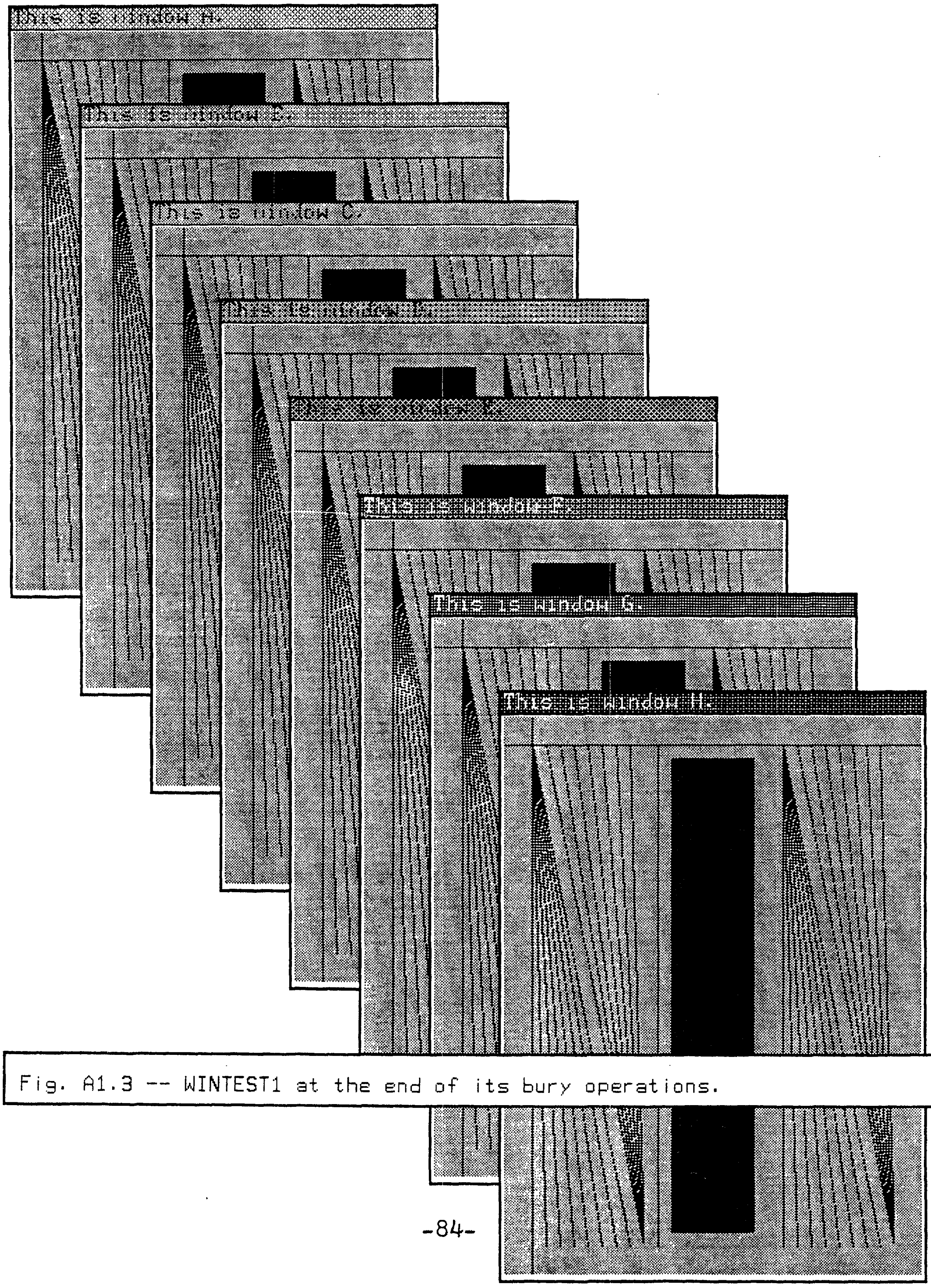


Fig. A1.3 -- WINTEST1 at the end of its bury operations.

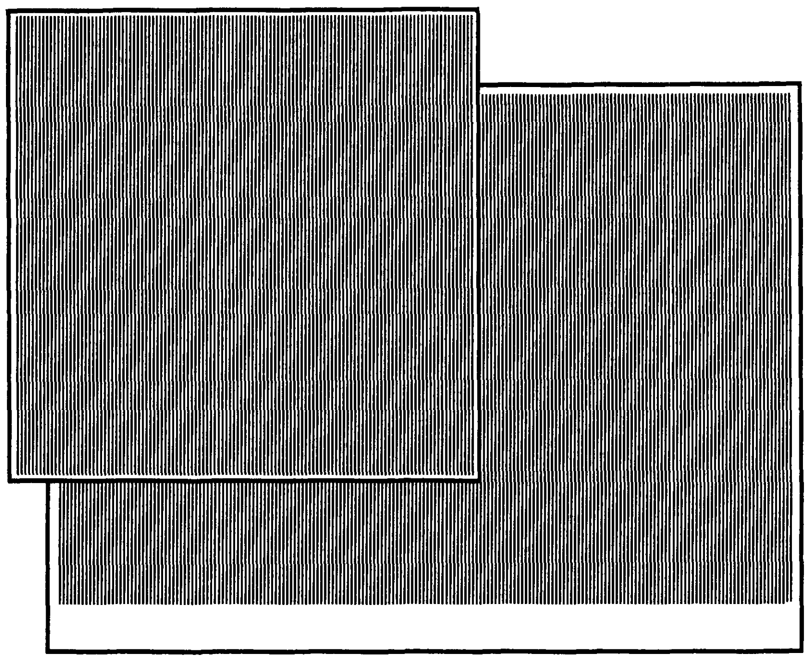


Fig. A1.4 -- The two overlapping windows in WINTEST3.

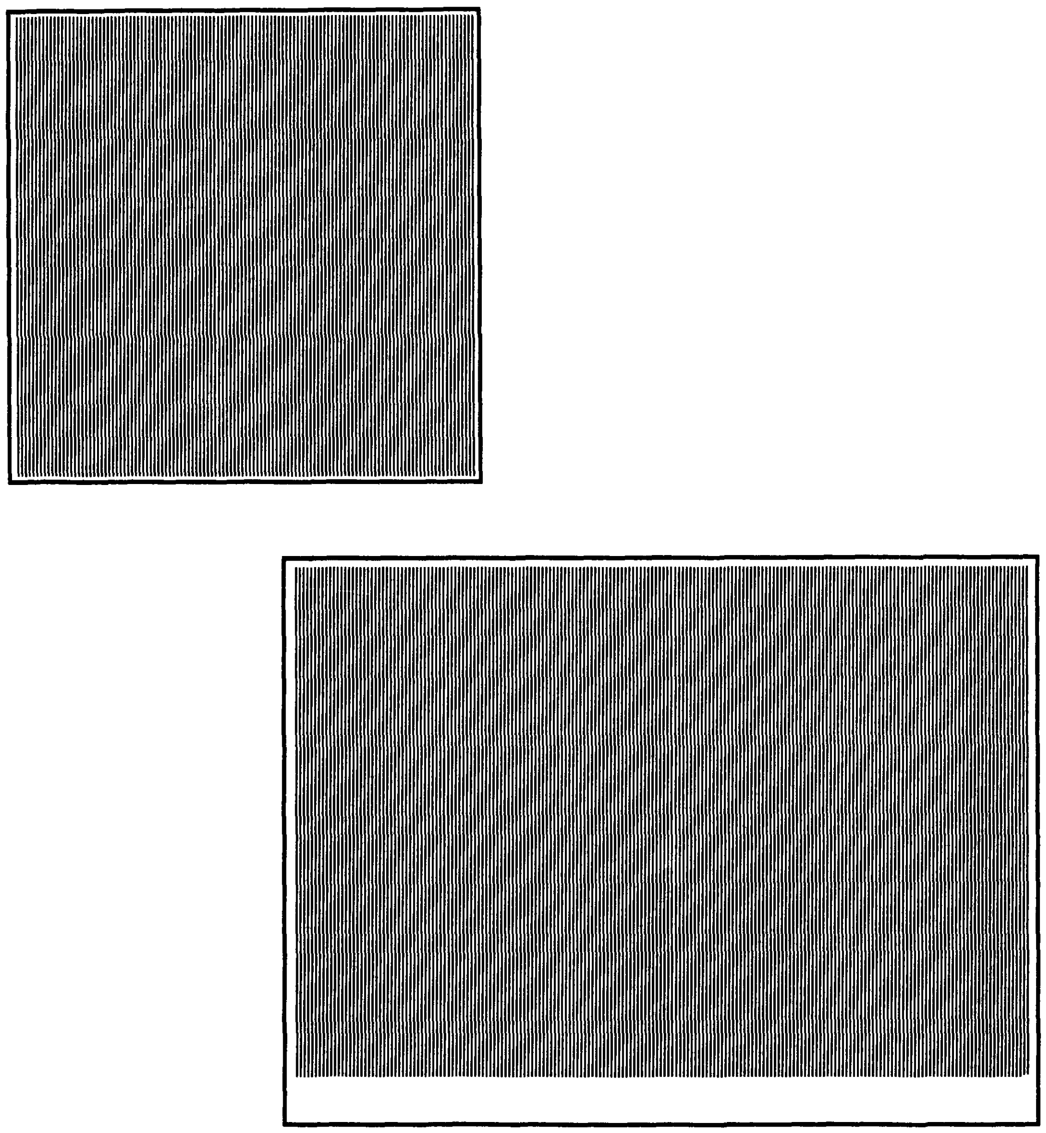


Fig. A1.5 -- WINTEST3 moving the bottom window across the screen.

second (Fig. A1.6). The burden is on the screen manager to update the screen quickly enough to match this rate.

4. WINTEST4 surfaces three non-overlapping windows and writes text into the canvas they all share. Since they all view different portions of the canvas, the writing appears in them at different times (Fig. A1.7).

There are two major criteria by which we can estimate the time performance of each of these screen management algorithms: their adeptness at handling the update of an overlapped window and the amount of overhead they generate when handling simple redisplay cases. An example of such overhead is provided by DNASB since it causes even the simplest cases to take longer by always performing twice as many writes as do the other algorithms (it first writes to the shadow buffer and then to the screen).

Since the focus of our attention is the performance of MPF, our analysis consists of comparing each of the other algorithms to MPF for each of the test programs. This is the reason we have chosen to normalize the time taken by MPF on each test program in Table A1.1.

## WINTEST1

WINTEST1 is the most complex of the four test programs. Accordingly, we expect it to be the most demanding to MPF. Since

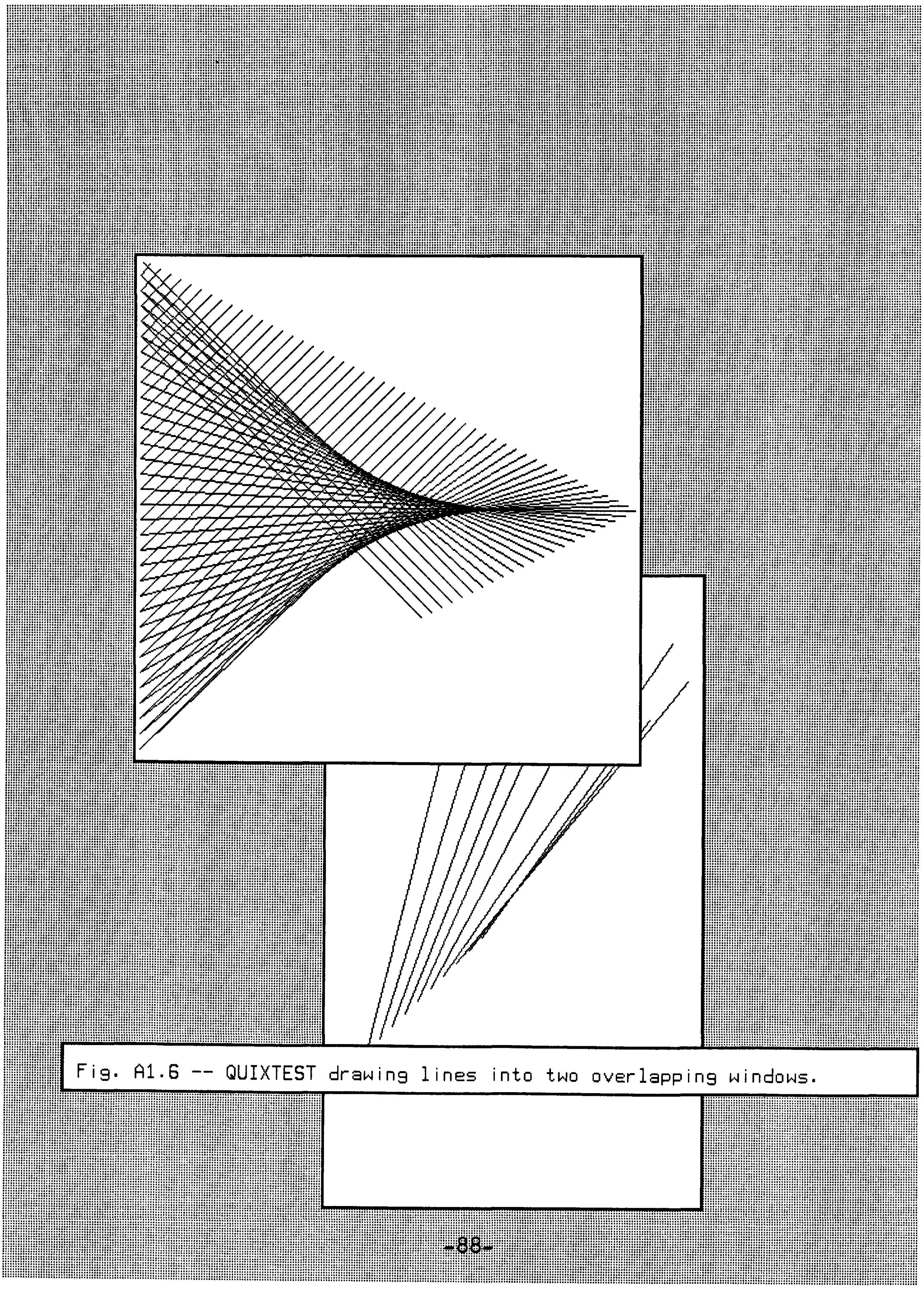
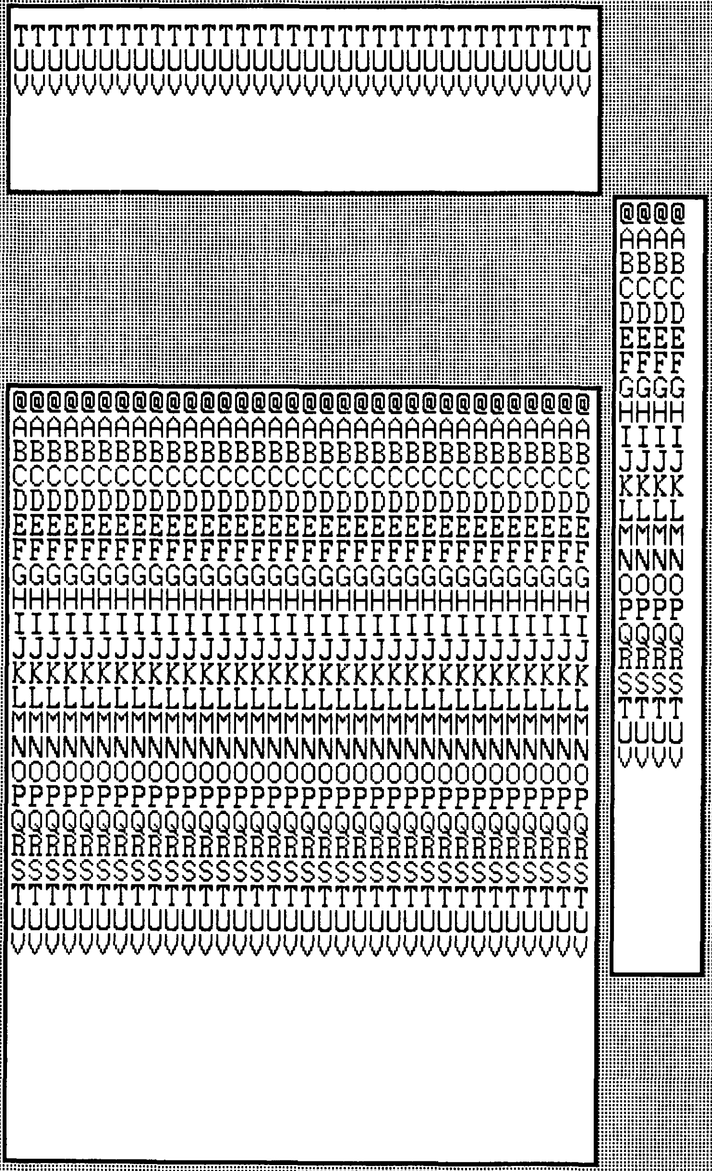


Fig. A1.6 -- QUIXTEST drawing lines into two overlapping windows.



```
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
```

```
@@@@
AAAA
BBBB
CCCC
DDDD
EEEE
FFFF
GGGG
HHHH
IIII
JJJJ
KKKK
LLLL
MMMM
NNNN
OOOO
PPPP
QQQQ
RRRR
SSSS
TTTT
UUUU
VVVV
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
```

Fig. A1.7 -- WINTEST4 allows three windows to share the same canvas.

DNA involves a minimal amount of processing prior to screen writing, we expect it to run faster than MPF, which involves a great deal of preprocessing, does. How much faster DNA is depends on how many bits it writes to the screen unnecessarily (i.e. how many bits are written only to be overwritten in the same update). This figure is in turn determined by the degree of overlapping present in the screen area to be redisplayed; that is the ratio of the area being redisplayed to the sum of the affected areas of each window to be redisplayed. This ratio, the "overlap ratio", is proportional to the difference in performance between a "thinking" algorithm (such as MPF) and a "non-thinking" one (such as DNA). If the ratio is small, then there is a great deal of writing time to be saved by not unnecessarily writing bits; if it is large, then there are not that many bits to be unnecessarily written, so one may as well not bother determining which ones are necessary and which ones are not.

In the case of WINTEST1, we observe a high degree of overlapping (Fig. A1.1). This means that the non-thinking algorithms (DNA and DNASB) should all be bogged down by writing bits unnecessarily. Therefore, although they do not incur the preprocessing overhead of MPF, they lose time because of unnecessary writes and in the end take almost as much time as MPF does. This is verified by Table A1.1.

One may expect that DNASB would take much more time than DNA since it does more bit writing. But once again, the overlap ratio comes into play to tell us that the extra write to the screen (from the shadow buffer in DNASB) is negligible when compared to all of the writes done anyway be both DNA and DNASB. Therefore, very little extra time is taken by DNASB (as is indicated by Table A1.1).

WINTEST3

In WINTEST3, we note that the ratio of overlapping is much higher than in WINTEST1. This, when coupled with the fact that throughout more than half of the test, there is no overlapping, suggests that the algorithms with the lowest overhead should perform significantly better than the algorithms with higher overhead.

The test results do, in fact, show this. The lowest overhead algorithm for non-overlapping configurations and simple configurations (i.e. configurations with a high overlap ratio) is DNA; it performed best on this test. The highest overhead belongs to DNASB (because the high overlap ratio makes its extra writes more significant); accordingly, it is the slowest algorithm on this test. MPF's preprocessing actually costs it time on this test since the time spent trying to save bit writes is greater than the time saved by not writing those bits (once again, because of the high overlap ratio).

QUIXTEST

QUIXTEST is an example of a good configuration for MPF to analyze. The area of overlap is large enough and the configuration simple enough to make the time spent by MPF in preprocessing less than the time it would have taken to write the unnecessary bits. This allows MPF to outperform both DNA and DNASB significantly.

WINTEST4

WINTEST4 demonstrates the disparity in performance by a low-overhead algorithm, a medium-overhead algorithm, and a high-overhead algorithm. In this case, DNA has no overhead whatsoever since it does no preprocessing and there are no wasted bits (compare to WINTEST3 where there were, in fact, some wasted bits to keep DNA's time high). MPF has a small amount of overhead in that it has to determine that there is no need for preprocessing (this determination is, in and of itself, some form of preprocessing). DNASB, of course, has its constant overhead of doubling the writes it must do.

In analyzing the results of such tests, we must keep in mind that there is a vast difference in the quality of the screen management provided by DNA and that provided by MPF and DNASB. We cannot expect to receive such an improvement without paying some price in time. This being so, it is somewhat remarkable that in half the tests, the non-flickering algorithms performed as well as or better than DNA.

Another point brought out by the test results is that although MPF outperforms DNASB in every configuration of simple to medium complexity, DNASB does have the edge in the more complex configurations. This emphasizes the point made in Chapter 1 that a simple complexity test with little overhead could enable us to achieve good performance in all cases by selecting MPF when it is the appropriate algorithm and DNASB when it is not.

## REFERENCES

(Borkin and Prager)
Borkin, S.A., and Prager, J.M., POLITE Project Progress Report, IBM Cambridge Scientific Center Report G320-2140, IBM Corp., April, 1982.

(Gonz)
Gonzalez, J.C., Implementing a Window System for an All Points Addressable Display, IBM Cambridge Scientific Center Report G320-2141, IBM Corp., December, 1982.

(O'Hara)
O'Hara, Robert P., AIDE - An Interactive Display Environment, IBM Corp., September, 1982.

(PERQ)
Three Rivers Computer Corp., PERQ Software Reference Manual, Pittsburgh, Pa., 1982.

(Tesler)
Tesler, Larry, "The Smalltalk Environment", Byte, Byte Publications Inc., August, 1981, vol. 6 num. 8.

(Weinreb and Moon)
Weinreb, Daniel, and Moon, David A., Introduction to Using the Window System, M.I.T. Artificial Intelligence Laboratory Working Paper 210, May, 1981.