BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems

Norman Meyrowitz
Margaret Moser

Department of Computer Science
Brown University
Box 1910
Providence, Rhode Island 02912

ABSTRACT

With only one process viewable and operational at any moment, the standard terminal forces the user to continually switch between contexts. Yet this is unnatural and counter-intuitive to the normal working environment of a desk where the worker is able to view and base subsequent actions on multiple pieces of information.

The window manager is an emerging computing paradigm which allows the user to create multiple terminals on the same viewing surface and to display and act upon these simultaneous processes without loss of context. Though several research efforts in the past decade have introduced window managers, they have been based on the design or major overhaul of a language or operating system; the window manager becomes a focus of -- rather than a tool of -- the system. While many of the existing implementations provide wide functionality, most implementations and their associated designs are not readily available for common use; extensibility is minimal.

This paper describes the design and implementation of BRUWIN, the BRown University WINdow manager, stressing how such a *design* can be adapted to a variety of computer systems and output devices, ranging from alphanumeric terminals to high-resolution raster graphics displays. The paper first gives a brief overview of the general window manager paradigm and existing examples. Next we present an explanation of the user-level functions we have chosen to include in our general design. We then describe the structure and design of a window manager, outlining the five important parts in detail. Finally, we describe our current implementation and provide a sample session to highlight important features.

1. INTRODUCTION

Normal computer terminals provide a two-dimensional window into the computing environment. Typically, the computer system, whether it be a personal computer or a mainframe, offers some **command interpreter** as part of its operating system. From this command interpreter, the user is able to "converse" with the system using functions supplied by the command interpreter: querying the number of users, sending messages, checking the time, compiling programs, etc. Some functions of the command interpreter give entrance into sub-environments like the editor, the symbolic debugger, and the mail system. In addition, the command interpreter usually provides both a special command parsing to pick out specially defined characters (e.g character delete) and keywords, and an interface to the file system.

Authors' present address: Bolt Beranek and Newman, 10 Moulton Street, Cambridge, MA 02238

This work was sponsored in part by a grant from the Digital Equipment Corporation and in part by the Office of Naval Research, under contract No. N00014-78-C-0396.

The single terminal model requires that the user only view *one* context at any moment; either the editor *or* the compiler's error messages *or* the mailbox, etc. Yet, like an executive at a desk, the computer user should be able to view multiple sources of information; the clock *and* the editor *and* the error messages *and* the mailbox should be both viewable *and* accessible without any change in context. In essence, the system should provide an arbitrary number of full-fledged virtual (non-hardware) terminals with which the user can switch back and forth at will without any loss of information or context. The user should be able to bury such windows at the bottom of the "desk" and bring them on top without a loss of state. Additionally, processes should be able to operate and update the display in parallel. Thus, the new paradigm requires that the following conditions be met:

(1) The user should have complete control of the size and location of windows.

(2) The user should be capable of creating and running parallel processes.

(3) The user should be guaranteed that the state of a window remains alive regardless as to whether that window is being used at the present time.

(4) The user should have to make *no* changes to existing software when a window manager/virtual terminal system is introduced to the computer system.

1.1. Terminology

Many of the terms used throughout this article are common to several areas of computer science, though unfortunately, the definitions of these terms often differ. A **window** indicates a rectangular area of an actual display surface which holds the contents of a simulated terminal. A **virtual terminal** is a software emulation of a *hardware terminal*. Typically this virtual terminal is designed to alleviate device dependencies. A standard communication method or protocol is used by programs that need to write to a display device; the virtual terminal emulator maintains a virtual image of the display screen and performs the mapping of this virtual screen to the variety of physical display devices available in a given system.

1.2. Existing systems

Several highly successful attempts at window manager/virtual terminal systems have been made. In XEROX *Palo Alto Research Center's Smalltalk*, [LRG76, GOLD79, TESL81], the window is a primitive of the environment. The XEROX Interlisp Programmer's Assistant [TEIT77, TEIT81] allows the user of the ALTO minicomputer to create and manipulate windows into the MAXC computer system at PARC. The RIG network [LANT79, LANT80] at the University of Rochester is an important work in the generalization of

windows and virtual terminals. RIG's Virtual Terminal Management System (VTMS) is a major and highly-integrated component of the RIG distributed operating system. [MCCR78], [APPO81], and [SYMB81] provide several other examples of existing systems.

1.3. Our System

Though many of the above systems provide wide functionality, most are specific to particular computers, machines, or operating systems, and require the applications programs to have some *a priori* knowledge of the existence of the window manager/virtual terminal system. Computer users often have huge amounts of effort and time invested in their software; an aim of the BRUWIN project has been to create a useful tool within the structure and constraints of the existing computing facilities while requiring **no** modification of existing applications programs. The product is a general system- and device-independent window manager/virtual terminal system design supplemented by modular system- and device-dependent routines only at the lowest level. We describe the functionality, structure, implementation and use in the following sections.

2. FUNCTIONALITY

BRUWIN allows the user to create on the viewing surface (desk) any number of arbitrarily-sized, arbitrarily-positioned rectangular windows (pieces of paper), each of which behaves exactly like a standard hardware terminal. A window can overlap other windows, completely cover other windows, or be completely disjoint from other windows. Each has its own private command interpreter subsequently called the **shell** (using UNIX* terminology). Each shell operates in parallel with the others; likewise, each window is updated in parallel with the others. When switching from one window to another, the processes running in all the windows retain their respective states. Because of this, processes generating terminal output will continue to generate output; processes waiting for terminal input will continue to wait for terminal input.

BRUWIN gives the user two modes in which to operate, **command mode** and **process mode**. Command mode allows the user to select display manager functions and viewing conditions, while process mode simply allows a user to use a particular window as the **current window**. The concept of a current window is very important; though an arbitrary number of windows may be available and active at the same time, the user (due to human limitations) can only be typing *into* one window, the current window, at any given time.

Associated with a window is a **pen color** and a **paper color**. The window is depicted as a sheet of paper in the paper color with characters printed in a complimentary pen color. On a monochrome display device, the pen color is the foreground color and the paper color is the background color. The window is topped by a ribbon (banner) which contains the user-assigned **window title**. This name aids the user in quickly distinguishing between windows; it has no meaning other than that attached to it by the user.

2.1. Command Mode

The BRUWIN user interface is based on the following minimum set of commands:

Cancel
Create
Change
Move
Destroy
Quit

We presume only that the user has a keyboard and some method of driving a tracking cursor used minimally to pick window coordinates and optionally for menu-picking and light-button handling.

*UNIX is a Trademark of Bell Laboratories

The **cancel** command allows a user to terminate the currently picked BRUWIN command. This only works in non-critical sections of the command, so that important steps in display management (e.g. updating a linked list) are not halted in the middle.

Create allows the user to define a new window. The system picks the next standard paper/pen combination (for monochrome screens this is black and white) and prompts the user to pick two points designating the diagonal of the window. The window outline is drawn, and a window name is prompted for and typed, the shell is started, the window is made the current window, and process mode is entered.

The user may create windows *anywhere*. Just as papers can overlap one another on a desk, so can windows overlap one another on the display surface. This is essential, as the power of a window manager system lies in the ability to bury and subsequently retrieve windows like pieces of paper with the guarantee that the associated processes are still running.

Change allows the user to change the size of an existing window. After picking the change menu entry, the user is prompted to pick an existing window by either pointing to it in the window space or pointing to its name in the **title menu**. (Containing the names of all the windows, the title menu exists to take care of the possibility of completely covered windows. Any time a window is prompted for, the user may pick either the window itself or its name from the title menu). Next, the user is prompted to pick the two new diagonal points of the window. The window is redrawn, the window is made the current window, and process mode is entered.

Move is a degenerate case of the change command; the window maintains the same proportion but is simply translated to another point on the screen. The user is prompted to pick an existing window as in the change command. Next, the user picks the new left corner of the window. The window is redrawn, the window is made the current window, and process mode is entered.

Destroy effectively "logs out" of a window; it causes the picked window to be erased and the corresponding shell and shell processes terminated.

In addition, if the user picks a point on an existing window, the system performs the **current** command. This takes the picked window (which may be partially covered), brings it on top of the pile of windows, and enters process mode.

2.2. Process Mode

Process mode is simply the mode in which a user can type into the current window. All windows and their associated shells stay active during process mode; after an internally defined time lapse, any pending output for any existing window is written to that window's virtual terminal buffer and (if the window is visible) updated in the appropriate window on the display surface. This continual update allows, for example, the ability to have constantly changing processes like clocks updated without a loss of accuracy.

2.3. Mode Switching

In general, the user will want to move from the current window into command mode to quickly choose a new current window. Thus, BRUWIN must provide a quick method with which to switch from **process** mode to **command** mode. For example, to execute, a quick change from the "editor" window to the "compiler" window while the "editor" window is the **current window** in **process** mode, the user simply returns to **command** mode, moves the cursor to the compiler window and performs a **current** command (perhaps by simply picking a point in that window with the cursor device). The user is now in **process** mode in the "compiler" window.

A typical BRUWIN session is found in Appendix A.

3. STRUCTURE

The BRUWIN design is broken into five major parts: three main modules (display manager, virtual terminal emulator, tasking controller), and two interface modules (display/vte interface, vte/task interface), as shown in Fig. 1. The three main modules have no knowledge of each other's existence; interaction between the three are completely governed by the two interface modules. This structure enables a major module to be replaced with a different (improved) version with no change to the other modules. This is especially important for the sake of adaptability.

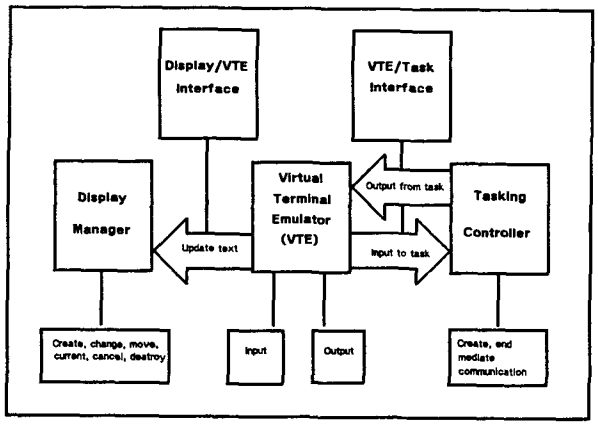


Fig. 1 -- **Structure of BRUWIN**

3.1. Display Manager

The **display manager** is a module which controls the mapping of windows to the physical device screen, performs the actual display update, and allows the user to manipulate the screen image. From a systems point of view, this can be broken into three parts: the command interface, the manipulation of the information associated with each window, and the graphical operations.

The command interface encompasses both a mechanism for receiving and interpreting user commands and the capability of displaying messages to the user.

User messages include both prompts for user input, instructions, and error messages. One way to provide the user messages is to reserve static screen space for a **system window**. Alternatively, user messages could be written into a dynamic system window, which is only drawn (typically at or near the cursor) when there is a message to be displayed and erased as soon as that message is no longer needed.

The display manager can receive commands in several lexical forms: as function key interrupts, as coordinates picked from a graphical menu, or as tokens of a command language; the method is a function of the peripherals available. With a data tablet and puck, user commands and their parameters (except textual input) can be specified using a cursor tracking device. Alternately, the display manager could read user commands from function keys, a command language, or from a program interface.

Maintenance of the display necessitates the ability to manipulate information to decide when and where to draw the windows. Decisions on the appearance of the windows will determine the amount of information to necessary to maintain the display. A simplistic display manager with disjoint windows might have a small number of predefined viewing areas. In this scheme, a window is described by the viewing area assigned to it. At the other extreme, a window description might contain the window boundaries (indicating the size and position), a title, pen and ink colors, and an array of flags indicating which windows overlap this one. This requires more fields in the description data structure and a method for resolving space conflicts which will be discussed below. Since a window is updated only when it is visible, the display manager provides a means for determining if a window is covered by another.

Finally, the range of graphic operations necessary is minimal. Primitives are needed to draw lines, boxes, and characters. Assuming these, the graphic routines needed by the display manager include ones to draw the initial screen configuration, draw a window, display a user message, and enter a window title. When it is decided that a window is to be updated, the task of actually redrawing that window belongs to the display manager. The capability to draw *portions* of windows is useful. However, this requires the computation of the portions of windows which should be visible in a screen area, the determination of which part of the virtual terminal screen is in a window portion, and the calculation of where characters should be clipped, a resource-intensive set of calculations which may be slower than redrawing the entire window with no computation for intersections.

3.2. The Virtual Terminal Emulator

A process's "interactive" input (output) in a standard operating system is normally directed from (to) a terminal. The process normally should have no knowledge about the terminal from (to) which it is reading (writing), though unfortunately, this is often not the case. Terminals provide an array of hardware functions which are invoked by writing special terminal-dependent control sequences. One higher-level technique is the use of a terminal database [JOY81], which allows programs to be written with device independence. Common terminal functions such as scroll, insert char, delete line, are looked up in the database with a generic name; programs need only know these generic names and not the specific terminal codes. Most of the screen manipulation and update (scrolling, deletion, etc.), are done in terminal hardware. The viewing surface is only updated by writes to that terminal or typing at the keyboard; the only storage of the displayed text is in the terminal's hardware character buffer. The viewer simply sees a one to one mapping of this buffer onto the display surface.

Currently, much work is being done in the area of **virtual terminals** and **network virtual terminal protocols** [LANT79, BAUW78, DAY80, SCHI78]. Here, perhaps using some defined protocol, an operating system's low level operating system **write (read)** primitive does not send (receive) characters directly to (from) the terminal hardware, but rather sends to (receives from) a **virtual terminal emulator** (vte). The virtual terminal emulator is a universal terminal -- all writes (reads) occur to (from) the same "brand" of virtual terminal. At the process level, then, the complexity of terminal i/o is greatly reduced, as input (output) need not be translated modulo specific devices -- every process does its reads (writes) from (to) the same universal terminal.

Of course, the device-independent to device-dependent translation must be done at some level; this is the job of the **virtual terminal mapper**. This software must be capable of accepting arbitrary input (output) and formatting it on actual device screens. Since the input (output) is not coming from (going to) a device which has the hardware to perform screen manipulation operations, the virtual terminal emulator performs these screen manipulation operations in software, operating on the virtual terminal data structure.

In Rochester's instantiation of the virtual terminal [LANT79], the virtual terminal data structures are the **line**, the **pad**, and **the window**. The line is a queue of characters

(or theoretically other events) that are stored when input by the user. The pad is a stable storage data structure which holds the entire contents of a virtual terminal session. The pad is a two-dimensional, ragged right array of lines, accessible by line number and character position within that line. Only a particular portion of the pad is displayed on the actual hardware screen at any time. The user has the ability of traveling through all previous output in any pad at will. Additionally, the pad provides editing capabilities: cursor movement; character; word, line, and page deletion; character insertion; string location and substitution; text selection; etc. One pad may be viewed in several different windows on one or more terminals. Thus the pad offers far more than the simple virtual terminal -- it offers editing facilities normally reserved for application-level editors.

The more conservative BRUWIN strategy is to build upon existing facilities rather than molding a particular operating system extensively for the virtual terminal capability. Thus, the BRUWIN design uses a **map**, a two-dimensional array of characters with a private cursor, similar to the pad. Unlike the pad, the map has no extraneous capabilities such as editing; it simply is a replacement for the hardware terminal's character buffer. That this is less powerful than the pad concept is unquestionable; but with the existence of numerous application-level text editing programs, the decision was made to use the simpler, more conventional, and easier to implement map. Input is typed directly into the current window's map at the current cursor position; line-correction facilities normally provided by the hardware terminal device (such as backspace, character delete and line delete) are emulated by the vte.

3.3. The Tasking Mediator

Multiprocessing is rightfully a support task of the operating system. Often, however, this support has severe restrictions. Some operating systems prohibit more than one process from receiving input from the same terminal or from sending output to more than one terminal. Others "toss a coin" in accepting input, leaving the user confused as to what process is actually going to receive the typed characters. Similarly, for output, these operating systems allow all processes to write to the terminal whenever necessary, creating a screen of haphazard text.

A prime function of a window manager is to manage the input/output contention of multiple processes in a transparent and logical way. To support a window manager, an operating system itself must either have built-in ability to distinguish between the input and output of multiple windows mapped to the same screen, or alternatively, must have built on top of it a tasking manager to do the same. It is this second alternative, the less operating system intensive of the two, which we address below.

In the general BRUWIN design, one process acts as the **mediator**, accepting and routing all input(output) from(to) a screen's windows. Each window's system shell is instructed to receive input from the shared mediator and send output to the mediator as opposed to normal terminal input and output. (See Fig. 2). We will call these input and output routes **paths**.

The mediator may work either **actively** or **passively**. An active mediator constantly **polls** the input paths of existing windows, reading any pending output and sending it to the **vte/task interface** to update the virtual terminal. The pseudocode for such a mediator would resemble:

```
loop forever
  loop for each window
    if pending output in out_path(window) then
      read (out_buffer) from out_path(window)
      write (out_buffer) to vte_task_interface(window)
    endif
  endloop
endloop
```

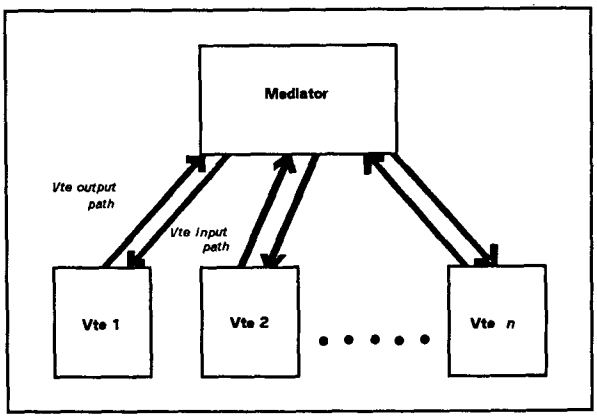


Fig. 2 -- **The Tasking Mediator**

Note that this active polling requires the existence of a **non-blocking read** primitive or a method of determining whether a path has data to read; since a shell may never send output until given some input, using a **blocking read** to read that shell's output path invites the possibility of deadlock.

Because of the wasted time looping to check for output on a path, the above method is less desired than the passive method. Rather than performing a busy wait, a passive mediator would simply sleep, waiting for a shell to send a wakeup signal or wakeup message containing the shell output. Such a mediator in Ada, an experimental language with support for such passive waiting, would resemble the following:

```
if (mode = PROCESS) then

  select
     accept WINDOW1 (in out_buffer)
  or
     accept WINDOW2 (in out_buffer)
  or


  or
     accept WINDOWn (in out_buffer)
  or
     accept OUT_PATH(CUR_WIN) (in in_buffer)
  or
     delay TIME
  end select
endif
```

Here the mediator sleeps for TIME seconds before timing out, waiting for a **rendezvous** with an output buffer from any open entryname (port) or the input buffer connected with the current window's input path. Note that the Ada tasking facilities provide total synchronization and contention support for the window manager. Note also that the SELECT statement is powerful enough to handle both the input and output paths of the shells at the same time; Ada runtime routines "randomly" choose which of the open SELECT alternatives should be accepted. This randomness assures that one window's output will not glut the system.

3.4. Vte/Task Interface

Given a tasking facility to mediate between the input and output of multiple (shell) processes and a vte facility to emulate the universal terminal, the BRUWIN design strategy needs a method by which to implement the aforementioned **paths**, which link the tasking mediator with the vte.

Many different strategies may be used to implement

these paths. In the UNIX operating system, processes, rather than writing to (reading from) standard output (input) files (usually the terminal), the shell would substitute **pipes**, special two-ended, synchronized files read by one process and written by another. DEC's VMS operating system, has a similar convention, the **mailbox**, to which many processes may write messages and from which one process may read these messages.

Another approach, most appropriate for a network network window manager (in which different virtual terminals may be maintained on completely different processors but may be displayed on the same physical screen) is the message-based transfer of input and output. Instead of reading from (writing to) the physical terminal, a process would send a message to (read a message from) a **port**. This message-based transaction methodology requires a handshaking-type synchronization or a more elaborate message protocol between the mediator and the shells.

A simpler variation is the use of a higher-level language with built-in, message-based, transparently synchronized input and output. The Ada example shown above describes such a system. Other languages [see HANS78] offer similar high-level support, while most languages will allow for such primitives to be constructed [TREN81, RASH80, JOY81, FELD79].

3.5. *Display Manager/Vte Interface*

With a display manager to control the placement and drawing of windows, and a virtual terminal emulator to "format" and hold the input and output of each window's shell, the BRUWIN design strategy calls for a display manager/vte interface which correlates the map of each vte to the proper window on the screen. This interface is broken into several parts. First, the vte needs to find out the location of the window. In a network, this might mean the location of both the hardware device and the location of the window on the device's screen. On other systems, the output device may be predefined; only the location of the window on the screen need be found. Secondly, the interface needs to correlate a line in the window data structure a location on the physical device screen. Finally, the interface offers a routine to actually transfer the text from the vte to the hardware screen. This module is the virtual terminal mapper described earlier.

3.6. *Summary of Structure*

To summarize, a BRUWIN-based window manager requires the following:

(1) an input device (normally a keyboard)

(2) a cursor tracking device (cursor keys, data tablet and puck, mouse, lightpen)

(3) a command interface (command language, function keys, graphical menu-picking)

(4) a screen manipulation scheme which tracks relative positioning and eliminates unnecessary redrawing during window movement

(5) a path facility to divert normal terminal input and output to the appropriate virtual terminal

(6) a system command interpreter (shell) which is able to read from (write to) the aforementioned paths rather than a standard terminal device and may be invoked multiple times (once for each window).

(7) a virtual terminal emulator which accepts input and output from the appropriate paths and manipulates it per instructions specified for a universal terminal, storing each screen in a map

(8) a facility to map the contents of a virtual terminal to the corresponding window surface.

(9) a facility to mediate multiple parallel processes, reading input from the current virtual terminal "keyboard" and sending output to the appropriate virtual terminal screens. (This implies of course, that any system supporting a BRUWIN-type window manager must allow multiprocessing).

3.7. *Typical operation pipeline*

The functioning of the display manager can best be illustrated by outlining the implementation of a user command. The user enters a command, say **create**, through some **user interface**. The display manager obtains the boundaries of and title of the window, generating prompts requesting these parameters. The returned information is entered into the description data structure for that window. Depending upon the implementation, some more information is computed as part of the data manipulation aspect of the display manager. Graphics routines are next called by the display manager to draw the new window.

As another example, consider the **change** command. To perform this, the user interface and the graphics routines perform the same functions as in **create**: reading user input, displaying user prompts, and drawing the window in its new size and position. In addition, the window in its old position is erased. Assuming the windows may compete for screen space, all windows which overlap the window's old position will now have portions erased and therefore must be redrawn. Furthermore, windows which intersect these newly redrawn windows may need to be redrawn (see Fig. 3). The BRUWIN strategy for minimizing the number of windows redrawn is discussed in a later section of this paper.

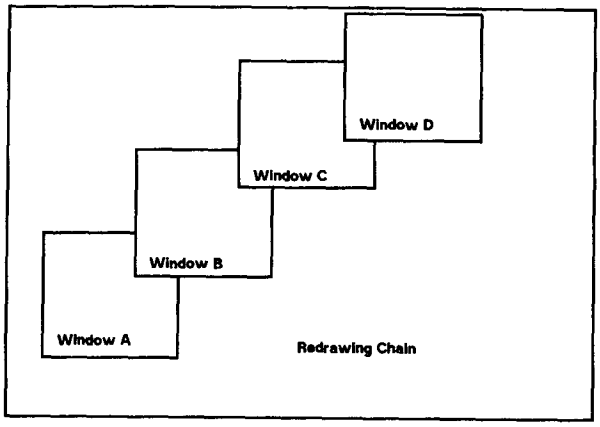


Fig. 3 -- **Redrawing Chain**

Notice that redrawing Window B necessitates redrawing Window C and Window D, even though Window D does not intersect Window B. Window A need not be redrawn because it is underneath Window B.

4. IMPLEMENTATION OVERVIEW

4.1. *Hardware/Software Configuration*

BRUWIN is a C language program running on a Digital Equipment VAX 11/780 under the 4.1 release of the UNIX operating system distributed by University of California at Berkeley and on a Bolt Beranek and Newman C/70 running Version 7 UNIX. Two versions exist: one uses a Ramtek color graphics systems as the display device, the other uses any terminal device with a description available in the UNIX *termcap* database [JOY81], a program accessible database containing descriptions of the features and control codes of particular terminals.

Hooked to the VAX through a UNIBUS connection, the Ramtek 9400 Graphic Display System is a graphics processor coupled with a high resolution 1024 x 1280 pixel color

window_information	
title []	window title (array of characters)
limits []	window boundaries
intersect_flags []	which other windows overlap
vte_information	
window_length	number of text lines in the window
window_width	number of characters in a line
current_line	line number of terminal cursor
current_char	cursor char postion in current line
char_map [][]	character map
task_information	
proc_id	process number of terminal's shell
to_read_path	descr. for path-to-shell read file
to_write_path	descr. for path-to-shell write file
from_read_path	descr. for path-from-shell read file
from_write_path	descr. for path-from-shell write file

Fig. 4 -- **BRUWIN Data Structures**

monitor. Driven by a Z-80 microprocessor, Brown's Ramtek, has eight 1024 x 1280 bit planes which are accessible only through the Ramtek instruction set. A color lookup table allows user definition of up to 2^8 colors from the 4096 color range of the Ramtek. The Ramtek is equipped with its own keyboard and a Summagraphics BIT PAD data tablet and puck which maps one-to-one with the physical screen. When a hit is sensed on the data tablet (the button on the puck is pressed) an interrupt is generated on the VAX. The terminal version runs both on the VAX and the BBN C/70.

The terminals are generally low resolution devices (approximately 24 x 80 characters), with optional graphics character sets facilitating the drawing of boxes. Cursor tracking is provided by cursor keys; interrupts are generated by sending UNIX signals.

4.2. Global Data Structures

All information relevant to a window and associated virtual terminal is entered in three data structures as described in Fig. 4.

These data structures are collected in an array, of a size determined by system constraints on the maximum number of concurrent windows. Each window is identified by its index into this array. Since this index also indicates the position of the window's title in the title menu, it is called the menu position.

In order to handle screen space conflicts, each window is given a priority number. A window's priority is analogous to its coordinate on a z-plane. The higher a window's priority, (its z-coordinate), the closer it is to the top of the stack of windows, occupying any screen space which overlaps the x-y boundaries of lower priority windows. Windows which are at least partially obscured by some other window with a higher priority are considered covered. The window of the current terminal always has the highest priority; it is closest in the z-plane and is therefore totally uncovered.

A window's priority is by no means constant throughout its life. Any time the current terminal is changed, the window of the new current terminal must be moved to the front of the z-plane and the priorities of the other windows decremented (z-coordinates moved further back in the z-plane). To do this, two arrays are maintained. When indexed with a window's identifying number (menu position), the value of *priority [menu_pos]* will be that window's priority. Inversely, when indexed with a window's priority, the value of *menu_position [pr]* will be that window's menu position. These two arrays, then are simply different representations of the same information. A value of -1 in the priority array indicates that the menu position and data structure with that index are currently unused. Fig. 5 graphically represents these relationships.

Pen and paper colors are standard combinations which are indexed by a window's menu position. There are two constant global arrays, *pen[]* and *paper[]* which contain the index into the color table of the colors in the standard combination. These of course degenerate to a binary choice in a monochrome display.

4.3. Display Manager Implementation

On the highest level, the display manager runs the user command input loop. The execution of each command entails entering information in the window data structure and updating the display per this information.

In order to save costly computation of the contents of a partial window, our implementation only redraws a window in its entirety. Our implementation also uses a static system window. Consequently, only the minimum graphics routines already mentioned in the general design are defined.

The user interface relies heavily on the data tablet as a pick/locator device. Each time a command is input or a window specified, a routine compares the coordinates returned by a pick with those of the command spaces and the windows to determine which item was picked. The tracking locator driven by the puck is also used to specify window boundaries. Besides invoking the user accessible functions, the display manager also handles the display of user prompts.

Each command fills in fields of the window data structures, either by computation or by user input. In addition, the *menu_position* and *priority* arrays are used to change the priority ordering of the windows each time a command is executed, making the window involved in the command the current window.

The **change**, **move**, and **destroy** commands require the old window be erased, moving it to the back of the z-plane and drawing it into the background. An erasure may lead to the problem of determining the minimal number of windows redrawn -- the redrawing chain. Initially, the window being erased is **the problem rectangle**. The general strategy for minimizing redrawing chains is to find the closest window which completely covers the problem rectangle, thus eliminating the need to be concerned with the problem rectangle. This window becomes the new problem rectangle, and the algorithm recurses. The algorithm is explained in detail in Fig. 6.

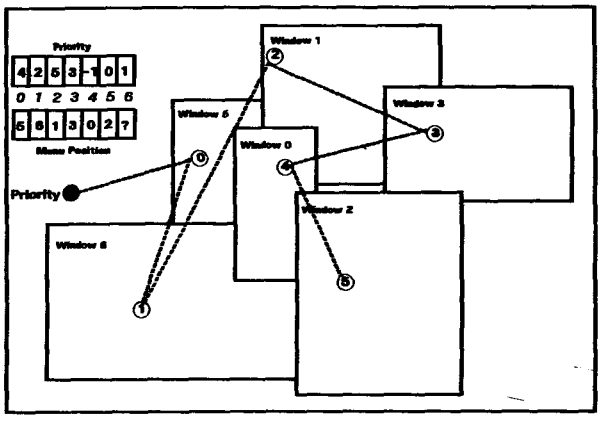


Fig. 5 -- **Priority and menu_position configuration**

```
procedure min_redraw
    (problem_limits []  -- boundaries of problem rectangle
     priority   -- priority of problem rectangle
     intersect_flag []  -- which windows intersect problem_limits
     to_be_drawn [])          -- which windows will have to be redrawn


 -- Assume the problem rectangle is to be redrawn until a window is found
 -- which covers it.

answer := true

-- Loop through all windows with a higher priority than the problem rectangle
-- starting with the highest priority.

i := num_of_windows
while (i > priority and answer = true) loop

  id = menu_position(i)
  if (intersect_flag(id) then

       if (boundaries of window(id) surround problem_limits) then
         -- Window[id] boundaries contain the problem rectangle. Therefore,
         -- redrawing window[id] will cover the problem rectangle.

         -- The problem rectangle does not have to be redrawn.

         answer := false

         -- The boundaries of window[id] are a new problem rectangle
         -- Min_redraw is called recursively to decide which of the
         -- windows that overlap window[id] will have to be redrawn. (If
         -- flag for this window is set, then it was already computed.)

         if not (to_be_drawn[id]) then
           new_prob := boundaries of window(id)
           new_prty := i
           new_intrsct := intersect_flag array of window
           min_redraw (new_prob,new_prty,new_intrsct,to_be_drawn)
           to_be_drawn(id) := true
         endif


       else
         -- Window(id) boundaries do not contain the problem rectangle but
         -- they do intersect it.

         -- Window(id) need not be redrawn if a window of higher
         -- priority contains the intersection of the problem rectangle
         -- and window[id]. Since this intersection is a sub-area of the
         -- problem rectangle, consider it a new problem rectangle.

         new_prob := intersection of problem_limits, limits of window(id)
         new_prty := i
         new_intrsct := intersect_flag array of new_prob

         if (min_redraw (new_prob,new_prty,new_intrsct,to_be_drawn)) then

           -- No windows cover the intersection of window(id) and the
           -- problem rectangle. Window(id) must be redrawn. The
           -- the boundaries of window(id) are a new problem rectangle.
           -- Min_redraw is called recursively to decide which of the
           -- windows that overlap window (id) will have to be redrawn.
           -- (If the flag for this window is set, then it was already
           -- computed.)

           if not (to_be_drawn(id)) then
                 new_prob := boundaries of window(id)
                 new_prty := i
                 new_intrsct := intersect_flag array of window
                 min_redraw (new_prob,new_prty,new_intrsct,to_be_drawn)
                 to_be_drawn(id) := true
           endif
         endif
       endif
  endif
endloop
```

Fig 6 -- **Algorithm for minimizing redrawing chains**

4.4. Virtual Terminal Emulator Implementation

In the BRUWIN Ramtek implementation and the BRUWIN Terminal version, the virtual terminal emulator is identical. The vte is split into two parts: the vte_input routine, which accepts input and writes it to the screen appropriately, and the vte_output routine, which accepts the output from other processes and updates the screen accordingly. Both the input and output parts are written to emulate a VT 52-type terminal. On output, the special codes are trapped, and the appropriate software emulation updates the appropriate map. On input, the vte performs special line correction actions upon receiving **backspace, carriage return, line feed, and tab**. Other characters are simply echoed in the appropriate place in the window map and passed through to the input path.

4.5. Tasking Implementation

Both the Ramtek and Terminal Versions of BRUWIN use the same active tasking mediator. Every *n* seconds in both PROCESS mode and command mode, the mediator loops through all of the open output paths (*from_r_path*), checking to see if anything needs to be read from the path. If so, the path is read and the characters are sent to the vte to be updated.

4.6. Vte/Task Interface Implementation

The paths described above are implemented in the BRUWIN versions through the use of UNIX pipes. For each vte, there exist two pipes, one which goes from the vte to the mediator and one which goes from the mediator to the vte. The vte checks every *n* seconds to see if new characters have arrived at the current window's keyboard. If so, the vte writes on the *to_w_path*, sending the characters down the pipe to be read by the shell. For the BBN Terminal version, the pipes were replaced with a *pseudo-terminal device driver* (pty), which looks much the same as the hardware terminal drivers without the hardware interface. Reads and writes to ptys take place in exactly the same way as reads and writes to ttys.

4.7. Mgr/Vte Implementation

In both the Ramtek and Terminal versions of BRUWIN, the vte interfaces with the display manager through the *x_map*, *y_map*, and *puttext* routines. *X_map* and *y_map* take a virtual terminal cursor and correlate that point to a physical device coordinate on the screen. The *puttext* routine uses does the physical write of a piece of text to the screen at the correlated coordinate. For the Ramtek, this routine calls a Ramtek graphics instruction which sends down-loaded fonts to the screen; for the Terminal version, the routine simply does direct cursor addressing before calling the system write routine.

5. CONCLUSIONS

The BRUWIN project has shown that a viable window manager system can be built in conjunction with an existing operating system with no structural changes to that operating system. Though users may sacrifice some efficiency (as the code is on a higher-level than pure operating system primitives), they gain the ability to use all previously written and compiled program with no changes. Moreover, with little work, the design of BRUWIN is adaptable to a variety of devices and operating systems. The restructuring of the RAMTEK version of BRUWIN for the Terminal version took less than a day. The restructuring of this Terminal version to run under BBN's C/70 UNIX took less than two hours, including the replacement of the pipe vte/task interface with the pty vte/task interface.

5.1. Further work

Representation of graphics in a window is common in Xerox PARC's Smalltalk, where graphics images can be stored as bit maps and maneuvered. For high resolution graphics, however, a general purpose computing system cannot afford to keep large bitmaps like the current

character maps for a terminal screen. Often, too, graphics instructions, unlike textual data, are written directly to the graphics device, making it impossible to intercept some representation for storage in a virtual graphics terminal emulator. Research needs to be done to develop a way in which to conveniently store and manipulate graphics data in the context of a window manager.

An important extension to a window manager is a program (rather than user) interface, so that programs, rather than users are able to access and manipulate windows and associated virtual terminals.

A higher-level research consideration is the design of a window manager-manager. This entity would allow one supervisor program to determine the format of, send data to, and receive data from specified users' screens. Such a supervisor would allow, for example, lessons to be dynamically broadcast and rearranged by a professor in a computer-science equivalent of a language laboratory [BROW80].

6. ACKNOWLEDGEMENTS

We wish to thank several people for their help on the BRUWIN project: Andy van Dam, for his initial support of the idea and subsequent encouragement through BRUWIN's many iterations; Tom Doeppner, for spending countless hours as our system sounding board; Steve Reiss, for his helpful advice and expedient code changes; Steve Feiner, for his often unsolicited but nevertheless extremely valuable suggestions from BRUWIN's inception; Nicole Yankelovich, for careful reading of final drafts; Bill Smith, for his aid in the formative days of the project; and Dave Johnson, for several suggested improvements to our basic algorithms.

REFERENCES

[Apol81] Apollo Computers Inc., *Apollo System User's Guide*, 19 Alpha Road, Chelmsford, MA 01824, July 1981.

[Bauw78] Bauwens, E. and Magnee, F., "The virtual terminal approach in the Belgian University Network," *Computer Networks* 2, 4/5 (September/October 1978), 297-311.

[Brin78] Brinch Hansen, Per, "Distributed processes, a concurrent programming concept," *Comm. ACM* 21, 11 (November, 1978), 934-941.

[Brow80] Brown University Computer Science Department, *Brown University Instruction Computer Environment*, Brown University, Providence, RI 02912, 1980.

[Engl68] Englebart, Douglas C. and English, William K., "A research center for augmenting human intellect," *Proc. 1968 AFIPS FJCC* 33, 1 (Fall, 1968), 395-410.

[Feld79] Feldman, Jerome A., "High Level Program for Distributed Computing," *Comm. of the ACM* 22, 6 (June 1979), 354-368.

[Fole82] Foley, James and van Dam, Andries, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

[GSPC79] GSPC, "Status Report of the Graphic Standards Planning Committee," *Computer Graphics* 13, 3 (August, 1979).

[Gold79] Goldberg, Adele and Robson, David, "A Metaphor for User Interface Design," *Proceedings of the Twelfth Hawaii International Conference on System Sciences* 6, 1 (1979), 148-157.

[Hoar78] Hoare, C.A.R., "Communicating sequential processes," *Comm. ACM* 21, 8 (August, 1978), 666-677.

[Hone79] Honeywell,, *Rationale for the Design of the Green Programming Language*, Honeywell Systems and Research Center, Minneapolis, MN 55413, March 15, 1979.

[Hone80] Honeywell,, *Reference Manual for the Ada Programming Language*, Honeywell Systems and Research Center, Minneapolis, MN 55413, November, 1980.

[Joy81] Joy, William and Horton, Mark, "TERMCAP," *UNIX Programmers Manual, Seventh Edition, Berkeley Release 4.1* (June, 1981).

[LRG76] Learning Research Group, *Personal Dynamic Media*, Xerox Palo Alto Research Center, Palo Alto, CA 94304, March 1976.

[Lant79] Lantz, Keith A. and Rashid, Richard F., "Virtual terminal management in a multiple process

environment," *Proceedings of the Seventh Symposium on Operating Systems Principles* (December 10-12, 1979), 86-97.

[Lant80] Lantz, Keith A., "Uniform Interfaces for Distributed Systems," TR63, Ph.D. Thesis, Computer Science Department, University of Rochester, Rochester, NY 14627, May 1980.

[List79] Lister, A.M., *Fundamentals of Operating Systems*, Springer-Verlag, New York, 1979.

[McCr78] McCrossin, J.M., O'Hara, R.P., and Koster, L.R., "A time-sharing display terminal session manager.," *IBM System Journal* 17, 3 (1978), 260-275.

[Newm79] Newman, William and Sproull, Robert, *Principles of Interactive Computer Graphics*, McGraw-Hill, 1979.

[Rash80] Rashid, Richard F., "An Interprocess Communications Facility for UNIX," Technical Report, Carnegie-Mellon University, Pittsburgh, PA., March 1980.

[Schi78] Schicker, P. and Duenki, A., "The virtual terminal definition," *Computer Networks* 2, 6 (December 1978), 429-441.

[Shaw76] Shaw, Alan, *Operating Systems Fundamentals*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Symb81] Symbolics, Inc., *Symbolics Software*, 21150 Califa Street, Woodland Hills, CA 01367, 1981.

[Teit77] Teitelman, Warren, "A Display Oriented Programmer's Assistant," Report CSL-77-3, Xerox Palo Alto Research Center, Palo Alto, CA 94304, March 1977.

[Teit81] Teitelman, Warren and Masinter, Larry, "The Interlisp Programming Environment," *Computer* 14, 4 (April 1981), 25-33.

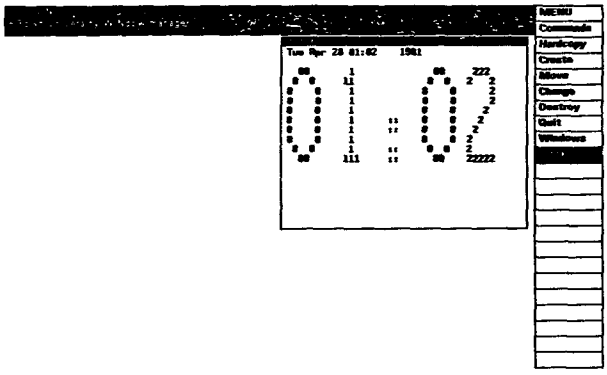
[Tesl81] Tesler, Larry, "The Smalltalk Environment," *BYTE* 6, 8 (August 1981), 90-147.

[Tren81] Trent, Barry A. and Meyrowitz, Norman, *ALTU: An Ada-like Tasking Facility for UNIX*, Brown University, Providence, RI 02912, 1981.

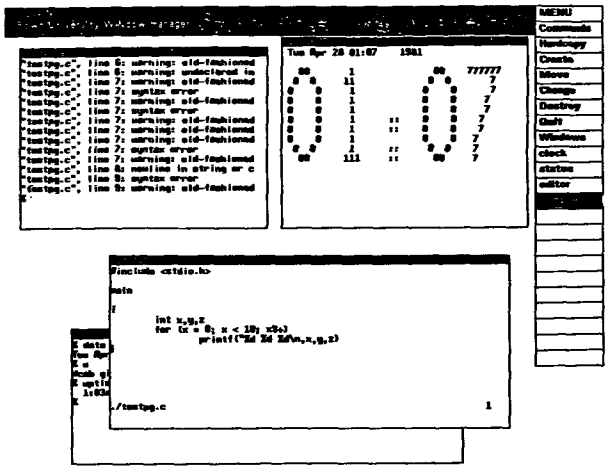
[Wegn80] Wegner, Peter, *Programming with Ada: An Introduction by Means of Graduated Examples*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

Appendix A

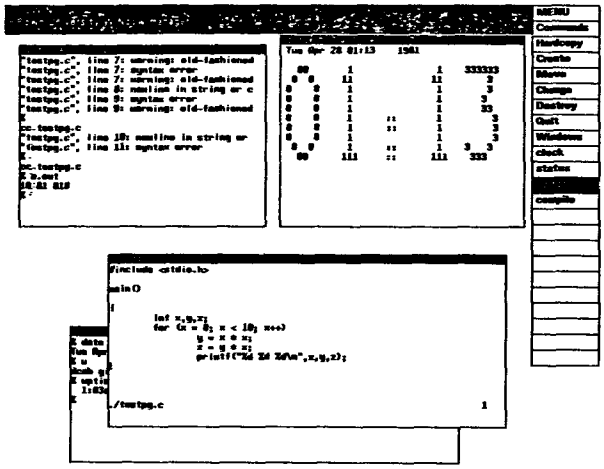
A view of BRUWIN at the start. The user has the *hardcopy, create, move, change, destroy, and quit* commands to chose from.



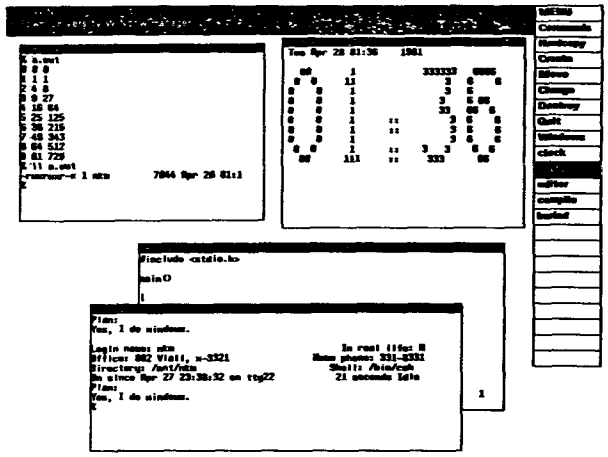
Here, the user has created a window titled "clock" by picking two points on the window's diagonal, and has executed a program which updates a clock once per minute.



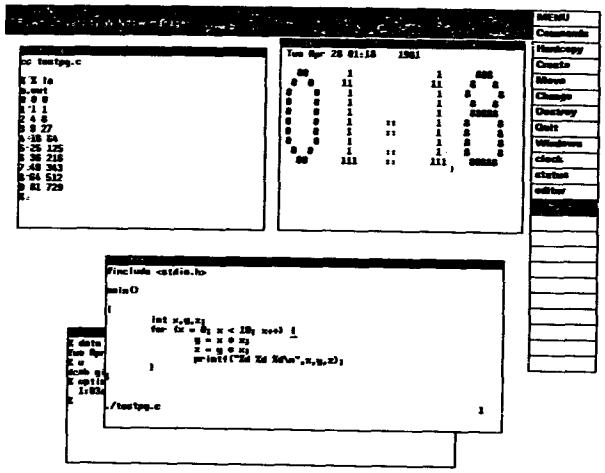
The user has created the status window and subsequently buried it with the editor window. The editor window is running a full-screen UNIX editor. Note that the program in the editor has been written and saved away; the user can leave this window and enter the *compile* window with assurances that the state of the editor window will remain the same. Note that the *compile* window has several error messages concerning *testpg.c* -- which is still visible on the screen.



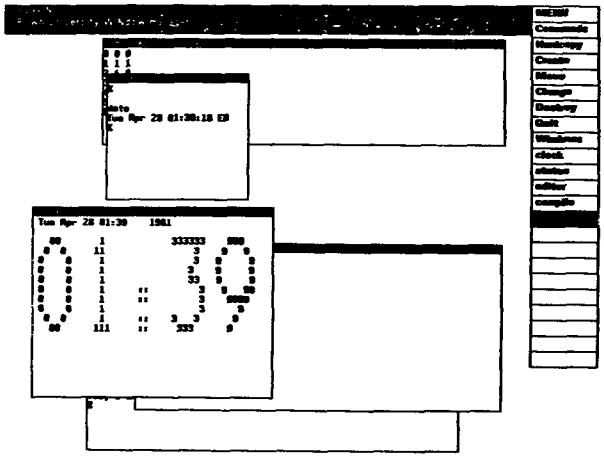
Here, some changes have been made to the program in the editor window and the *compile* window has again been entered. The program runs, but produces erroneous output.



Here, the status window is brought up on top. Note the stripe in the title menu which indicates the current window. Note also that the window called *buried* is listed in the title menu but is obscured.



The user corrects the mistakes, and again runs the program in the *compile* window. This time, the program runs properly. Note that throughout this session, the clock has been updating steadily.



Here by touching the buried entry on the title menu, we bring the buried window on top. Next we pick the move command and move the window to the lower left. We then touch the change command and make the shape of *compile* short and wide.