To: Distribution

From: Bernie Greenberg and Jim Davis

Date: 10/01/80

Subject: Towards a Windowed Video System

1. INTRODUCTION

This MTB introduces the concept of a video window system. A video window system supports the special features of video terminals in a terminal independent fashion, while ameliorating their disadvantages. A subsequent MTB will propose an implementation. This MTB serves only to define the issues.

Unfortunately, there are still some problems to be solved. This MTB hopefully is a complete description of the things such a system should do, and of the issues involved, and offers some answers for the problems we do understand. More thinking can follow from the foundation we lay here.

Send comments to:

the System M continuum >udd>m>jrd>mtgs>tv

or by Multics mail, on MIT or System M, to
Greenberg.Multics or JRDavis.Multics

or by phone
Greenberg: (617)-492-9330 HVN 261-9330
Davis: (617)-492-9382 HVN 261-9382

2. MOTIVATION

Video terminals are becoming universal on Multics. They are cheaper, quieter, faster, and more reliable than printing terminals. In addition, the advent of Emacs has brought even more video terminals to Multics. Unfortunately, except for Emacs, Multics currently treats video terminals in almost exactly the same way it treats printing terminals (other than the most rudimentary End of Page processing).

Used in this way, video terminals have two significant drawbacks. The first is that as lines are sent to the terminal, older, perhaps still useful information quickly scrolls off the top of the screen and is lost. The second weakness is that almost all video terminals are unable to overstrike. Multics goes to great lengths to ensure that "what you see is what you get", but the rules appropriate for an overstriking printing terminal simply cannot work on video terminals in use today.

The existence of Emacs shows that video terminals can be used in ways that more than compensate for their weaknesses and in fact far surpass anything possible on a printing terminal. This power is based on the ability of video terminals to selectively write, clear, and re-write different areas of the screen, offering a chance to display the information that the user wants to see continuously.

Current readily available video terminals offer no feature to compensate for the inability to overstrike. This is due to the economics of memory for a terminal. There is no inherent reason why overstriking cannot be done on a video device, but the memory needed to hold the additional characters would make the terminal cost more than people are willing to pay for the feature. We can look to the time when costs will go down, and this feature will be available again. In the meantime, this suggests that rules for treating overstriking should change incompatibly for video terminals. This is discussed below.

3. THE CONCEPT OF A WINDOW

The basis for the use of screens is the idea of a window - a (rectangular) area of the screen that is itself a "virtual screen". There may be many windows visible on one terminal screen at one time. Different windows may contain different interactions (possibly from different programs) for different purposes.

There is much flexibility possible in the definition of a window; Windows that do not extend clear across the screen, and windows that exist but lie partially or fully "buried" under other windows are all reasonable to consider. In some systems, windows may be nested within other windows.

Output in a window is confined to that window. Output stops when a window is filled, until the user has read it all, then it resumes. (Optionally unseen output may be discarded.) Input is edited on a real time basis (as in Emacs today). The rest of this section describes the features of a window system.

3.1. Output Conversion

Output Conversion is the conversion of characters in text to a displayable form, for example, displaying ASCII Form Feed (14 octal) as \014, or, on an IBM 2741, displaying Left Bracket as a cent-sign followed by a less-than character. Output Conversion also includes wrapping over-length lines, and placing a "\c" on the beginning of the new line.

The ring zero TTY DIM does output conversion now.

3.2. End of Page

The system must remove output from a window to make room for new output. There are three ways to do this.

One option is to scroll output off the top of the window. Almost all video terminals, even those without cursor addressability features, will scroll continuously when fed output consisting of lines ending in linefeeds. Most scroll by one line, at least one scrolls by four. Scrolling a window smaller than the full screen is only feasible if the terminal has the ability to insert and delete lines or the line speed is 9600 baud or above. Without the insert-delete lines feature, the entire window must be re-written. In the single case where there is one window which covers the whole screen, the terminal's inherent ability to scroll can be used. It isn't obvious how to use this feature though, given that windows can change size, shape, location, and number after creation. A window that starts out covering the full screen may not always do so.

A second option is to move the cursor to the top of the window, and begin output from there, overwriting lines. There are some who feel this is easier to read than scrolled output, since the lines don't move across the screen. In many cases, the user will be perusing a large file. This mode resembles reading output a page at a time.

A third option is to clear the window, and begin from the top.

3.3. MORE Processing

"MORE" processing is that window management function which is invoked when an attempt is made to sequentially output more text than will fit in a certain window. More processing halts the output at the bottom of the window to allow the user to inspect (read) the window contents, and acknowledge having read it, before the next windowful is displayed.

Multics does a crude form of this on terminals in "page length" mode by printing "EOP" when output has filled a screen (which is a degenerate case of a window). The user acknowledges having "read" the window by hitting a formfeed or newline. ALL process output stops until that formfeed has been hit.

Most systems that claim to support video terminals say "MORE?" or something equivalent when MORE processing is invoked, telling the user that more follows. A space is a common acknowledgement character, due to its ease of typing. Proper MORE processing involves the option of hitting some other character to say "no more, I don't want to see the rest of this thing." On Multics today, this can only be done by hitting QUIT, which is treacherous, because unanticipated error messages or, worse yet, console messages, can be thrown away with no indication that they were lost.

3.4. Input Editing

By now the idea of real-time input editing should be familiar. One of the fundamental concepts of the Multics input system is that "what you see is what you get". It is possible to implement that concept by using the ability of display terminals to selectively erase unwanted input. In Emacs, # and @ really remove characters from the screen, and this should be true in the window system as well.

The editor should support erase and kill, an input escape character, and offer some way of recovering a killed line. The editor can be arbitrarily powerful. Word-delete is a reasonable next step. So is the ability to insert text in the middle of a line (rather than just at the end). One hard problem is deciding how to control the editor. Emacs uses ASCII control characters, but these are reserved for line, device, and protocol control by the standard. It seems even less good to use printing characters for editing.

Real time erase and kill processing is an INCOMPATIBLE, though minor, change to input canonicalization. Under the present scheme, an erase character can itself be overstruck with a printing character, thus changing its effect from deleting the character to the left to deleting the character it was overstruck with. In effect, this is a way of cancelling an erase character. The difference between current input schemes and a real-time editor is that in the current scheme you overstrike an erase character to "discard" it, in a real-time editor you undo what it did by recalling the text.

Similarly, under the current scheme a kill character can be overstruck with an erase character. We don't say here how to un-kill a line, but one possibility is that a line may be un-killed by typing an erase character on a killed line.

The audit editor shows that it is useful to make previous input lines editable, so that the user can correct errors and resubmit command lines with minimal typing.

4. SCENARIOS

Here we give a few of the ways that windows might be employed:

In the simplest scenario, a window is just a stream output device. Output appears in the window, and scrolls towards the top. This is what Multics offers today, augmented with MORE processing and input editing.

The next step is to create multiple windows.

The probe debugger could utilize multiple windows very effectively. Interactive dialog might go in one window, the current source in a second window, and the values of variables might be displayed in another. Octal dumping might well go in a fourth, optional, window, stack traces in a fifth, etc. The Emacs Lisp-debug mode demonstrates how useful and powerful such an environment is. Similar debuggers exist on other systems. A sample screen appears at the end of this document.

The most powerful use of the window system will be by programs that take advantage of the ability to move the cursor about, re-write selected areas of the screen, and so on. Such programs will "know" that they have a video device at their disposal, and will perhaps also have to find reasonable ways of behaving on non-video devices as well.

A mail reading program might keep a list of the contents of the mailbox in one window, with a moving cursor to indicate which mail was being read, while the mail itself was in a second window. A reply would be entered in a third window, with the original still visible for reference as the response was entered.

Emacs could take advantage of window support, which it provides for itself now.

Finally, there is the menu system, which uses one window for display of a menu and a second for output from the commands.

The common factor in all these scenarios is that the screen is divided spatially. Information that is important to the user remains on the screen, in a fixed location, and is easy to locate. Transient or unimportant location is somewhere else, out of the way, in a separate window, and it scrolls out of its window without affecting the important data.

5. LIMITS OF WINDOWS

A Window Video system can do things that can't be done without one, but the added features don't come for free. The kinds of windows that can be supported with tolerable efficiency depends strongly on the terminal and the line speed.

It is an absolute requirement that the terminal and communications line be asynchronous and full duplex. At a minimum, the terminal must have an addressable cursor. If the line speed is 9600 baud or greater, then other features may be absent. At any lower speed, the ability to clear to the end of a line is required. MTB 419 sets out the requirements of a video environment for terminal features.

A terminal without the ability to insert and delete lines cannot fully support more than one window unless the line speed is greater than 9600 baud, because inserting or deleting lines may (in the worst case) involve transmission of hundreds of characters. At line speeds below 9600 baud, the transmission delay is unacceptable.

Given the ability to insert and delete lines, multiple windows can be supported reasonably well at 1200 baud, provided that the windows extend across the full width of the screen. These "vertically stacked" windows are the kind provided by Emacs.

A second limit to windows is the size of the screen. A 24x80 cannot comfortably be used with more than three windows, because there isn't enough room for an interesting amount of data in the windows. Some terminals, (e.g., the HISI VIP7801 (some models) and the Delta Data 4000), support "multiple pages", i.e., memories larger than the screen, but this does not increase the visible area at all. At best, given very sophisticated display management, such memory can be used to optimize user waiting time, but its use as a user-visible interactive feature seems inadvisable.

It is also appropriate to consider bit-map displays. A bit-map display is a type of video display composed not of characters but of points, arranged in a matrix, typically 1024x512 points. Although no such device is used with Multics today, there is reason to believe one may be in the future. Bit-map displays can display text in arbitrary fonts, sizes, positions, and orientations. Because they have more points on them then ASCII CRTs do, they can display a full page of text; and many more windows than will fit on a 24x80 CRT. Finally, both line and grey-scale graphics may be freely mixed with text. Window systems were originally devised for bit-map displays. If Multics ever gets bit-map displays, the window system must be able to support them.

6. THE BACKSPACE PROBLEM

As mentioned above, video terminals now available can't overstrike. Overstrikes are used for three main purposes: characters are overstruck by the erase character to erase them; overstruck with underscores to underline them; and overstruck to form characters the terminal can't print (i.e. APL).

Input editing meets the need for rubout, it remains to deal with underlining and overprinting.

First, although some terminals (e.g. the VIP7801) have a "forms" capability that can represent underlined text, these features are so difficult to use and so variable from terminal to terminal that they are useless. There is no special case for underlining of which we can take advantage.

Overprinting is done using the ASCII BS character. On a printing terminal, this character moves the "cursor" backward. If the cursor is moved backwards on a video terminal, the previous contents will be destroyed by the next character output. We propose to display the BS as an escape sequence. Rather than display the octal escape ("\010"), which is fairly meaningless, we propose a new, more mnemonic sequence: "\BS". This convention can be extended for all other non-printing ASCII characters.

One alternative for input of BS is to treat it as an erase character. This has been proposed by users of Multics before, and does ensure that "what you see is what you get". It is in the MR9 PFS. The disadvantage of this is that this changes the meaning of BS, leaving no way to input text that will be overstruck. Also, when video terminals that can overprint appear, users will have to change thir habits again.

A second alternative is to move the cursor to the left (like Emacs ^B) Printing characters would self-insert, rather than overstrike. This is only possible if the editor is extended to allow middle-of-line editing.

One option offered by Emacs is to suppress the display of backspace. Successive overstruck characters will appear next to each other. For example, the word "Seem" overstruck with underscores, would be represented as "S__e_e_m".

This has the advantage of keeping all characters on the screen, but it takes up extra screen positions. Formatted text lines displayed in this way seem to be overlength, though they fit in their specified lengths when printed. In addition, the user can't tell whether a string is really overstruck, or just "funny looking". (For example, the string "S_e_e_m" used above is indistinguisable from Seem, to an Emacs user is this mode.)

The best we can do for APL is to define printable output escape sequences. This will be hard to understand, but better than any other possible result.

7. FURTHER EXTENSION

7.1. Piece of Paper Management

Useful information stays visible longer when multiple windows are used then when they are not, but it will always be the case that there will be more worth saying than room to say it in. On a printing terminal the user can search back through the paper for something previously printed. On a video terminal the information is gone (unless the audit_dim is being used). A window system can address this problem by what we call "Piece of Paper Management" (PPM).

The basic idea here is to implement a virtual screen larger than a real screen, or in general, larger than the virtual screen on which it is displayed. A "piece of paper" is like an editor buffer, it has a given content at all times, coordinates within it, and a "current position". A given piece of paper can be on display or not at any given time. The dynamic bindings of pieces of paper to windows allows a great deal of flexibility.

To implement PPM with any efficiency requires a Redisplay function. Redisplay is the procedure which updates a window (or screen) contents by comparison of its known contents with an image of "what it should look like", and using character-by-character, line-by-line, or better, comparison techniques to determine how best to make the window look like it should. Redisplay minimizes the number of characters sent to the terminal to update the screen, because of line speed limitations. Redisplays generally involve tremendous complexity to minimize terminal output, cursor motion, and computation time.

It isn't clear what functions PPM should try to perform. It should be possible to scroll backwards through paper, to see previous output. Should the user ask the application to scroll, or should s/he communicate directly with the window system for this. Does scrolling backwards also move the cursor? If so, where does output go when the cursor is not at the end of the paper? If not, does the cursor vanish? If the paper contains user input requests, it would clearly be desirable to be able to "pick up" previous input for re-entry. Is it meaningful to alter a transcript of user_output? The design we present allows for editing an input line. An extension to editting buffers is reasonable (think of send_mail). Does this commit us to re-implementing Emacs, or at least its lower foundations?

7.2. DESK MANAGEMENT

Windows are created, can change size and position, and are destroyed. Since all windows share the space of one terminal, changes in one window effect other windows. Desk Management coordinates this for the user.

We don't know what Desk Management should do.

When a new window is created, where does it get its space from? Does it replace one or more existing windows, or do some windows shrink to provide space for it? How can the user control the size and position of windows? Is the size and position of windows controlled by the application, the user, or both?

When a window is destroyed, what becomes of its space? Is it divided evenly among all, or split with its neighbors? Is there a way to bequest space to the window that "most deserves" the space?

Emacs attempts to answer some of these questions with "pop-up" windows and the "window editor". It's not clear what the answers are.

One service the Desk Manager can perform is to optionally display visual window separators.

We don't know how subsystems that use windows should behave with respect to each other. This isn't a problem we have to solve, except that we have to ensure that it is possible to do whatever is "right". For example, if there are some windows on the screen, and the user invokes read_mail, does read_mail have the right to use the entire screen? If so, does it have the obligation to restore the previous window contents (or at least configuration) when through? If read_mail doesn't have the right to the whole screen, how does it know its limits? How do you use video_probe on video_read_mail?

7.3. Visual Attributes

Many terminals today can display text in ways other than white letters on a black screen (e.g. high intensity, low intensity, inverse video, underlined, blinking, etc.) These attributes are often used for forms input. It is important to support these features in a terminal independant way, since Multics needs a forms facility.

Terminals also have simple graphics capability. This is perhaps better addressed by the Multics Graphics System.

We don't know best to define a "virtual" visual attributes terminal. The design should be easy to use, and implementable on most terminals. A second problem is that terminals are highly variable in their support of these features, and it will be hard to find a scheme for describing terminals suitable for the TTF.

7.4. Hierarchical Windows

Nested windows can be useful. For example, read_mail might want to display a mode line showing author and subject of the current piece of mail. This is most easily done with a window within a window. It's hard to define the behavior of heirarchical windows. If a window is cleared, should all inferior windows be cleared? How can the user see output "under" an inferior window?

There are many more questions about heirarchical windows, but since no device with enough resolution to be useful with overlapping windows is likely soon, we won't consider them further.

8. SAMPLE SCREENS

```
+-----------------------------------+-----------------------------------------+
| 6   thrint_ (line 259)            | bmp = 432|1127(27)                      |
| 5   command_processor_|11006     | i = 17                                  |
| 4   abbrev_|7507                  |                                         |
| 3   listen_|7756                  |                                         |
| 2   process_overseer_|37773      |                                         |
| 1   user_init_admin_|42370       |                                         |
|                                   |                                         |
+-----------------------------------+-----------------------------------------+
|  if p -> std_symbol_header.identifier = "bind_map" then do;                 |
|     bmp = addrel (p, p -> std_symbol_header.area_pointer);                  |
|     i = bmp -> bindmap.n_components;                                        |
|     if comp ^= 0 then                                                       |
|          if comp > i then call com_err_ (0, "thrint_", "only ^d components in|
\c ^a", i, ent);
|     else call pst_head (addrel (p, bmp -> bindmap.component (comp).symb_star|
\ct),
| addrel (p, bmp -> bindmap.component (comp).name_ptr),                       |
| fixed (bmp -> bindmap.component (comp).name_lng, 17, 0));                   |
+-----------------------------------------------------------------------------+
|  stack                                                                      |
|  v bmp                                                                      |
|  v i                                                                        |
|                                                                             |
|                                                                             |
|                                                                             |
+-----------------------------------------------------------------------------+
```

```
  m#  lines   From                                Retief Series Sequel
  1   (15)    Keith Laumer (Laumer.Multics)
 >2    (3)    Carry.Multics                       Carry of vtie.pl1
  3   (11)    Olin Sibert  (Sibert.MultAdmin)     Bug in Probe
  4   (25)    Sendak.Wild                         Forest Access Needed
```

```
From: Carry.Multics
Date: 23 March 1981
Subject: Carry of vtie.pl1

Sorry, I could not carry vtie.pl1 to MISL-Multics, because I don't have
access to the containing dir (>udd>m>Ossining).
```

```
To: Carry.Multics
From: Ossining.Multics (James Ossining)
Subject: Re: Carry of vtie.pl1

 Oops I will fix that.  Please try again.
```

9. QUESTIONS

How does this fit into DSA?

If the terminal has additional character sets, how can they be accessed?

How can the terminal status line ("25th line ") be accessed in a uniform way?