The Maryland Window Library

Chris Torek

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

There are many applications for windows, but the implementation of window software is not an easy task. The job can be considerably simplified by eliminating some of the capabilities of a full system, for example, requiring that windows do not overlap, but this reduces the usefulness of the result. This paper describes the University of Maryland Window Library, an implementation that supports a very complete character-based window system.

1. Introduction

What exactly is a window? It is merely a rectangular region that can display text. In fact, it is very much like a standard terminal's display. However, since it exists as a subarea of a real display, it can be moved around, or covered by other windows. Windows can never be bigger than the real display.

One useful analogy for windows is glass over a room full of text. This room is called the buffer. The buffer can be bigger than the window. However, only the area that can be seen through the window is visible.

Two separate windows can look into the same room (buffer). The same room is revealed through both windows, but not necessarily the same part of that room. These windows are called "linked windows" although "linked buffers" might have been a better phrase.

Thus there are two basic "parts" to a window: its window descriptor (its position and size); and its text buffer or textbuf (holding the text to be displayed within the window). It is also useful to have a portion of a window for non-textual information. This is called the winbuf, or "window buffer". Normally the winbuf contains only "glass" (empty space), but it can be written on. For example, an outline, or frame, may be written onto the glass, to show how much space is available within the window (see Figure 1).

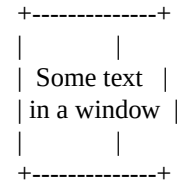


Figure 1: Framed window

Figure 1 shows a sixteen by six window with a frame. (The frame characters shown are those used on most displays. On displays that have some graphics capability, often the frames are cute little corners and sides that join together, giving the window a very solid appearance.) The text is in the text buffer, and the frame in the window buffer. This window has a descriptor indicating that the window is sixteen characters across by six characters high, and points to the window's winbuf and text buffer. The text buffer contains two blank lines, the words "Some text"

on the third line, the words "in a window" on the fourth line, and two more blank lines. The winbuf contains the frame characters, with the inside of the winbuf being "glass" that displays the text buffer.

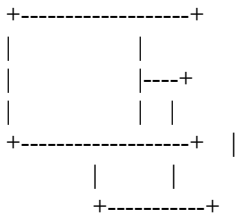


Figure 2: Overlapping windows

Windows can overlap. In Figure 2 two windows overlap. The only difference between overlapped characters and non-overlapped characters is that overlapped characters are invisible. If a window is covered, it can be brought to the "front" (where it will cover whatever used to cover it). Windows may also be hidden to make them totally invisible. Hiding a window that covers others will expose the others: hiding the upper window in Figure 2 would make the lower one completely visible. Hiding a window has no effect on the operations that can be performed on it; one can’t see the results, however, until the window is unhidden.

In addition to windows, there is a single "box" that may be turned on. It is a hollow rectangular region on the display that is made of inverse video characters. It can be used either to outline a window, or as a roving block cursor. As a window outline, the box can be used to display where a window may be created or moved. As a cursor, the box can be used with a mouse or a bitpad as a pointer for selecting windows.

2. What’s in a window?

2.1. Cursors

Like a terminal, a window needs a cursor. The window's cursor shows where the next character will be placed within that window, though characters are actually written into the text buffer. The cursor is normally visible as a blinking block. It can be turned off if desired.

Since the window actually has two pieces, the winbuf and the textbuf, it also has more than one cursor. The normal cursor ("window cursor") is the one just described. There are two others: the aux cursor and the buffer cursor. The aux cursor and the buffer cursor will be discussed later; for now it is enough to say that they are never visible, and that all three cursors are mutually independent. In this document, "the cursor" refers to the window cursor.

2.2. Text

Characters within a window can be in any combination of four modes: bold, underscored, blinking, and reverse video. There is a "current mode" associated with each window. This is the mode with which future characters will be printed. In addition, there is the "glass factor" for characters within the winbuf: are they real characters, or are they just empty slots through which the text buffer should be displayed? Frame characters are not glassy; the rest of the winbuf is normally filled with glassy blanks. The only way to put things into the winbuf is with the aux cursor or one of the special functions, e.g., frame window.

2.3. Text Buffers vs. Window Buffers

A window buffer, as stated earlier, is normally just "glass" through which the text buffer is displayed. However, it is possible to write characters, such as those making up a frame, into the winbuf. These characters then obscure some of the characters in the text buffer. It is also possible to write modes into the winbuf. These modes can be considered "tinted glass": they affect the display of characters being viewed, though the characters themselves are not changed. For example, if inverse video has been "painted" onto the "glass", all the characters in the text buffer under that paint are displayed in inverse video. If the glass is wiped clean of its inverse video paint, the characters will be restored to normal. The modes on the glass are exclusive or'ed with those in the text buffer.

Text buffers also have cursors, namely the "buffer cursor". In linked windows, where two window descriptors share a text buffer, there is only one buffer cursor. In other words, the buffer cursor is shared along with the text buffer.

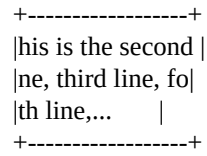
2.3.1. Margin Settings and the Aux Cursor

Window buffers also have margin settings. These are mainly used internally to keep the window cursor from overwriting the frames (if there are any), but can be set if desired. Margins determine the amount of glass in the winbuf. Anything outside the margins is strictly off-limits; the space inside the margins is the glass. However, unwritable margins would not be particularly useful. The aux cursor is permitted to roam over the entire winbuf. It can be used to print characters directly into the winbuf. Any character in the winbuf will take precedence over the character underneath it in the text buffer. Essentially, the aux cursor writes on the glass.

While the aux cursor can be used to write into the winbuf, it can also be used in a sneaky way to "un-write" characters: the aux cursor is permitted to erase from the glass. Erasing a character from the winbuf re-exposes the character underneath in the text buffer. Tinted glass is created by erasing with a mode other than normal (i.e. any combination of blinking, bold, underscored, and reverse video).

Text buffers do not have margin settings. No part of the text buffer is protected. Quite often, however, some of the text buffer is not visible. In a typical framed window, the entire top and bottom lines of the buffer, and the left and right edges, are not visible, since they are covered by the frame characters. (Note that since the aux cursor could at any time erase part of the frames, they must remain invisible until explicitly exposed. They can’t be automatically "moved down and right".) Figure 3 should clear some of the mud.

A window



and its text buffer

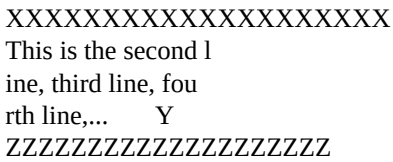


Figure 3: A window and buffer

2.3.2. Operations

Since a buffer can be larger than its window, there are "buffer operations" and "window operations". In general, window operations deal with the visible portion of the text buffer, and in many cases with the glass through which said text is viewed. Buffer operations deal exclusively with the text buffer, and are not limited to the visible portions[1]. Figure 4 shows what happens when a line delete is performed as both a window operation and a buffer operation on a window that is smaller than its buffer.

[1] "Visible portions" can include covered or hidden portions. "Covered" is used to describe windows or characters that are obscured by other windows. "Hidden" is used to describe windows not displayed at all by specific request. The visible portion of a window's textbuf is the area that is inside the window's margins; this is irrespective of whether the window is hidden and/or covered.

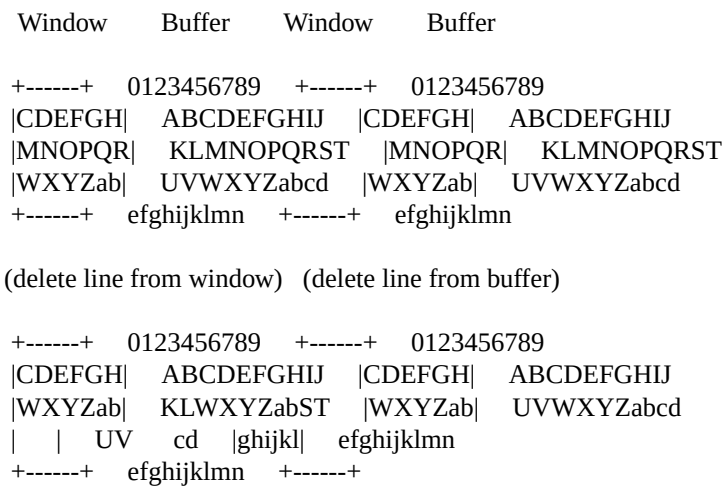


Figure 4: Window and Buffer Operations: Line Delete

When the cursor position is relevant to the operation, window operations are done at the window cursor, and buffer operations at the buffer cursor. Only two operations (aux print and aux read) are done at the aux cursor position.

2.4. Scrolling

The real reason for having separate text buffers and window buffers is to allow the storage of more text than is visible. That is, buffers can be much larger than the window through which they are viewed. This means that a window can be moved over the buffer. To go back to the room-of-text analogy, the wall is flexible, and the glass of the window can be pulled around on it. In the window library, moving the window over its text buffer is called scrolling. Thus you can scroll a window up, down, left, or right, over its buffer. There are limits, however: the amount of scrolling permitted depends on just how much more space there is in the buffer than the window. You can’t pull the glass past the edge of the wall.

The more conventional definition of scrolling is moving the existing text upward or downward, discarding one line at the top or bottom, and making room for new text. The window library provides this by making an exception to the scrolling limits. If the window is scrolled past the end of the buffer, the buffer is "extended" one line at the

bottom, and a line is "erased" from the top, making the window fit once again. (The actual algorithm is to copy the text upward, keeping the window at the bottom of the buffer.) A similar exception is made for scrolling past the beginning of the buffer. This action (moving the text upwards or downwards, losing a line off the top or bottom) is called "scrolling the buffer", and may be done directly, without moving the window past the edge of the buffer.

Windows automatically scroll when necessary. When this happens, the number of lines scrolled is controlled by the window's "popup". Popup defaults to one, but can be set anywhere from zero to the size of the window. Scrolling by *n* lines ($n > 1$) is exactly equivalent to scrolling by one line *n* times. Scrolling by zero lines is a special case. It is not scrolling at all. Rather, the cursor is brought to the top of the window and the top line is cleared. Each time the cursor advances to the next line, that line is also cleared. When the cursor reaches the bottom of the window, the process is repeated. This is much faster on some terminals, but tends to be confusing.

2.5. Printing Text

There are three ways to print text into a window. The first is to print at the window cursor, the second to print at the buffer cursor, and the third to print at the aux cursor. The first and second are generally more useful than the third, and have less "drastic" effects.

The aux cursor can be anywhere within the window buffer, and characters printed with it are required to have a mode. This mode includes (in addition to the standard four mode bits) the "glassiness" information. The character being printed is not interpreted in any way, but simply shoved onto the glass. If the character is not standard ASCII (codes 32 to 126 decimal), the results are terminal-dependent. Some terminals use special codes for the window frame characters, and look for such codes; others do not and may end up clearing the screen, repositioning the cursor, or doing other undesirable things. Therefore, the aux cursor must be used very carefully.

Both the window cursor and the buffer cursor check the character being printed, and take special actions for non-ASCII characters. Most of these codes are ignored. The following are not:

BELL

ASCII 7, or bell, rings the terminal's bell. If desired, this may be set to flash the terminal's screen instead,

providing that the terminal has this capability. Visible bells are often nicer in a crowded working environment.

BS ASCII 8, or backspace, moves the cursor backwards one character. If the cursor is at its leftmost permitted column, it does not move.

TAB ASCII 9, tab, moves the cursor forward one tab stop. Tab stops are set at every eight columns. If there are no remaining tab stops, the cursor is moved to the rightmost column.

LF ASCII 10, linefeed, moves the cursor down one line without changing its column. (If CR/LF mapping is turned on, it moves it to the leftmost column.) If the cursor is on the bottom line, the window is scrolled. If this is the buffer cursor, then one line at the top of the buffer is thrown away, and a new blank[2] line is introduced (i.e. the buffer is scrolled). If this is the window cursor, then the window is scrolled by its popup.

CR ASCII 13, carriage return, moves the cursor to the leftmost column. (If CR/LF mapping is turned on, it also performs a linefeed).

For the window and buffer cursors, autowrap and CR/LF mapping can be turned on and off. Autowrap means that when a character is written in the last column of the window or buffer, it is wrapped to the next line, as if a carriage return and line feed had been printed. This mode defaults on. CR/LF mapping makes both carriage return and line feed go to the first column of the next line; i.e., either code performs the actions normally done by both. This mode defaults off. The aux cursor always wraps, and, upon reaching the last column of the last line, wraps to the first column of the first line. Since it does not translate special codes, it ignores CR/LF mapping. It should not normally be used to print CRs or LFs.

3. Update Optimization

[2]The blanks introduced are in the current mode of the window; i.e. if the current mode is inverse video, a line of inverse video blanks is made. This is true for all functions that introduce blanks in the buffer.

The Maryland Window Library keeps track of what the terminal's screen looks like, and when asked to update ("refresh") the screen, attempts to do it with the greatest possible speed. If a section of lines on the current screen matches a section on the desired screen, line insert and delete operations may be done to move the old lines to their new position. If subsections of an old line match those of a new one, insert and delete character may be used. Any number of arbitrarily complex operations may be done between screen refreshes; the update is computed at refresh time.

4. Signal Handling

Since window display mode turns on many special Unix[3] tty modes, it needs to take special consideration of signals that normally cause it to stop or exit. (For more information on signals, see *signal*(2) and *jobs*(3j) in the Unix Programmer's Manual [Joy et al 81].) The Maryland Window Library uses the job control routines provided in Berkeley 4.1 Unix. Since many Unix systems don't have job control, the actual calls to these routines are made only if permitted by the programmer. If signal handling is not enabled, it is the programmer's job to manage the Unix signals. Signal management is not always easy, so if enabled, the window library will provide default signal handlers for those signals likely to occur, namely SIGINT and SIGTSTP. SIGINT is the interrupt signal (generated with the DEL key). The default handler for this signal exits after restoring the initial terminal modes. SIGTSTP is the terminal stop signal (generated with control-Z). The default handler for this signal cleans up the terminal settings, suspends the process, and upon resumption re-initializes the terminal to windows mode.[4]

Enabling signal handling is done with the **Winit** call (described later). If at all possible, it is recommended that signal handling be enabled. If the default signal handlers are not desired, signal handling should be enabled anyway, and the handlers changed to new functions via **sigset**.

If the target operating system does not have the jobs library, then there are two alternatives. One is to write a jobs library simulator (see Appendix 3); the other to not use it. In either case the functions **sigset**, **sighold**, **sigrelse**,

[3]Unix is a trademark of Bell Laboratories.

[4]I chose not to catch SIGQUIT since core dumps are handy for debugging. Besides, one can always call **sigset (SIGQUIT, Wexit);**.

and **sigignore** must be provided. If the functions are not being used, they may be empty (e.g. **sigset () {}**).

5. Window Library Functions

This section describes the actual functions provided by the Maryland Window Library. All return a value. Most return zero if they succeed; exceptions are listed. To compile with the library, #include the file <local/window.h>, and include "−lwinlib −ljobs −ltermlib" when making the executable. (See Appendix 4 for sample programs.) The first four functions listed are those for initializing, suspending, and cleaning up; the remainder are in alphabetical order. Note: several functions "clip" their values. This means that if they are given out-of-range values, the nearest value that is in range is used.

The window library provides several types, macros, and variables. The main type is **Win**, and is the window descriptor structure. There is also a "current" window, called **CurWin**, but it is in no way special; it is provided for convenience only. It is affected only by the functions **Wopen**, **Wlink**, **Wclose**, **Wcloseall**, **Wfind**, and **Wboxfind**. The type, macro, and variable definitions are listed in Appendix 1. Note: **bold** text indicates exact strings. *Underscored* text indicates arguments that are to be filled in.

Winit (*settings***,** *nomagic***)**
struct sgttyb ****settings*;**
int *nomagic***;**

Initializes the window system, clearing the screen and setting the tty modes. Normally, both *settings* and *nomagic* should be given as zero. To resume a suspended session, where **Winit** has already been called, *settings* may be given as **(struct sgttyb *) 1**. If special tty mode settings are desired, *settings* can be the address of a preset **struct sgttyb** structure (see *stty*(2), *gtty*(2)). (Baud rate will be replaced with the actual baud rate of the terminal.) Unless signal handling is not desired (or not necessary, e.g. in raw mode) *nomagic* should be given as zero. Otherwise signal handling will not be enabled. A value of zero will set **Wsuspend** to catch SIGTSTP and **Wexit** to handle SIGINT.

Winit returns zero if the window package can be run on the terminal. If **Winit** does not return zero, nothing has been done, and none of the other functions should be called. (**Winit** will return the same value if called again, so if it said "Ok" once it won't change its mind.)

Wsuspend ()

Suspends the process by calling **Wcleanup**, sending SIGTSTP, and then calling **Winit (1, 1)**. This is the standard way to handle control-Z.

Wcleanup ()

Cleans up the terminal settings to the way they were when **Winit** was called. This must be done before exiting, or before giving up control of the terminal (e.g. to a subshell).

Wexit (*code***)**
int *code***;**

Calls **Wcleanup** and **exit (***code***)**. This should be used instead of **exit** unless **Wcleanup** has already been called.

Ding ()

Rings the terminal's bell. This is normally just called when control-G (bell) is printed, but it can be called directly if desired. If the variable **VisibleBell** is nonzero, and the terminal supports it, this will flash the terminal's screen instead.

Max (*a***,** *b***)**

This is a macro which expands to

((*a***) > (***b***) ? (***a***) : (***b***))**

which is the larger of the two arguments.

Min (*a***,** *b***)**

This is a macro which expands to

((*a***) < (***b***) ? (***a***) : (***b***))**

which is the smaller of the two arguments.

WAcursor (*w***,** *row***,** *col***)**
Win ****w*;**
int *row***,** *col***;**

Sets the position of the window cursor. The position (0, 0) is as far left and up as the window cursor can get over the text buffer; this function may scroll the window to get to the specified position. The name of this function stands for "Window Absolute cursor move", meaning move the window cursor to an absolute coordinate, where "absolute" means measured from as far left and up as possible. To move only within the window (where (0, 0) is the upper left corner inside the margin settings), use **WWcursor**. *Row* and *col* are clipped to be within the window.

WAread (*w***,** *charonly***)**
Win ****w*;**
int *charonly***;**

Reads back a character and (unless *charonly* is set) its mode from window *w*'s winbuf at the aux cursor. The aux cursor is then advanced, wrapping from the last character of the last line to the first character of the first line if necessary. See **WCHAROF** and **WMODEOF**.

WBclear (*w***,** *howto***)**
Win ****w*;**
int *howto***;**

Clears (sets to blanks) the text buffer attached to window *w*. If *howto* is zero, the buffer is cleared from the buffer cursor to the end of the buffer, including the character under the cursor; if one, from the beginning of the buffer to the cursor, not including the character under the cursor; if two, the entire buffer is cleared. Other values are treated as zero.

WBclearline (*w***,** *howto***)**
Win ****w*;**
int *howto***;**

Clears the line the cursor is on in the text buffer attached to window *w*. If *howto* is zero the line is cleared from the buffer cursor to the end of the line, including the character under the cursor; if one, from the beginning of the line to the buffer cursor, not including the character under the cursor; if two, the entire line is cleared. Other values are treated as zero.

WBcursor (*w***,** *row***,** *col***)**
Win ****w*;**
int *row***,** *col***;**

Sets the buffer cursor of the text buffer attached to window *w* to row *row*, column *col*. Note: there is only one buffer cursor per buffer (i.e. two linked windows have only one buffer cursor). *Row* and *col* are clipped to be within the buffer.

WBdelchars (*w***,** *n***)**
Win ****w*;**
int *n***;**

Deletes *n* characters starting at the buffer cursor from the text buffer attached to window *w*.

WBdelcols (*w***,** *n***)**
Win ****w*;**
int *n***;**

Deletes *n* columns starting at the buffer cursor column from the text buffer attached to window *w*. In effect, calls **WBdelchars** on each line.

WBdellines (*w***,** *n***)**
Win ****w*;**
int *n***;**

Deletes *n* lines from the text buffer attached to window *w*, starting with the line containing the buffer cursor.

WBinschars (*w***, *n*)**
Win ****w***;**
int *n***;**

Inserts *n* blanks at the buffer cursor in the text buffer attached to window *w*.

WBinscols (*w***, *n*)**
Win ****w***0**

Inserts *n* columns in the text buffer attached to window *w*. In effect, calls **WBinschars** on each line.

WBinslines (*w***, *n*)**
Win ****w***;**
int *n***;**

Inserts *n* lines at the buffer cursor in the text buffer attached to window *w*.

WBputc (*c***, *w*)**[5]
char *c***;**
Win ****w***;**

Prints character *c* to the buffer attached to window *w* at the buffer cursor, and advances the buffer cursor. Note that this character may well not be visible inside any or all of the windows using this buffer; care should be taken when using this function to be sure that (if necessary and desired) a window is **Wrelscroll**ed so that the character is visible.

WBputs (*s***, *w*)**
char ****s***;**
Win ****w***;**

Puts a C style (null terminated) string into the buffer attached to window *w*, by using **WBputc** on each character in *s*.

---

[5]Note that for this and all other **putc** and **puts** functions, the window parameter comes *last*.

WBread (*w***,** *charonly***)**
Win ****w*;**
int *charonly***;**

Reads back a character and (unless *charonly* is set) its mode from the buffer attached to window *w* at the buffer cursor. The buffer cursor is then advanced, wrapping from the last character of the last line to the first character of the first line if necessary. (Note that this ignores the autowrap setting.) See **WCHAROF** and **WMODEOF**.