Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

G. van Rossum

STDWIN - A standard window system interface

Computer Science/Department of Algorithmics & Architecture Report CS-R8817 April

69D22, 69K34, 69K36

STDWIN – A Standard Window System Interface

Guido van Rossum

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
E-mail: guido@cwi.nl or mcvax!guido

ABSTRACT

STDWIN is an interface layer placed between an application written in C and arbitrary window system, making the use of windows both easier and more portable. For applications using STDWIN for their window management requirements, adaptation to a different window system is as easy as linking with an appropriate version of the STDWIN library. So far, STDWIN libraries are available for the Apple Macintosh, for the Whitechapel MG-1 (running Oriel), for MIT's X Window System version 11, for the Atari ST, and (subsets) for alphanumeric terminals on UNIX† and MS-DOS. New implementations are easily written.

Like STDIO, C's Standard I/O library, STDWIN's aim is to give a simple interface, high-level functionality, and portability. It does not attempt to allow access to all possible features of window management systems; rather, it provides the programmer with a model which allows easy construction of that part of the program which is concerned with window management.

STDWIN's high-level operations include automatic window positioning and resizing, scrolling, menus, keyboard shortcuts, and multiple-click detection.

1980 Mathematics Subject Classification: 68B20.
1982 CR Categories: D.2.2, I.3.4, I.3.6.
Key Words & Phrases: window systems, user interfaces, portability.
Note: This paper has been submitted for publication elsewhere.

1. Introduction

First, some history. STDWIN's conception was motivated by the desire to add a more modern user interface (i.e., one using windows and a mouse) to the programming environment for the language ABC, developed here at CWI.[1] The ABC programming environment, consisting of a syntax-directed editor, an interpreter and a source file manager, is a large body of C code which, through careful isolation of system-dependent modules, has proven to be quite portable, both to different versions of UNIX and to non-UNIX systems such as MS-DOS and the Apple Macintosh. Naturally, we did not want to lose its portability by tying it closely to one particular window system.

Only after having looked closely at a few existing window systems, we became fully aware of the problems. Most window systems offer a large range of facilities, apparently intended to enable programmers to create beautiful user interfaces, but often resulting in total chaos.[2] One of the problems appears to be the low level of most window system interfaces. For example, on the Apple Macintosh, all tools are provided to work with scroll bars (bit-scrolling operations, routines to draw scroll bars, routines to detect user

† UNIX is a Trademark of AT&T Bell Laboratories.

Report CS-R8817
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

interaction with a scroll bar), but the amount of code needed to glue these together and create a scrollable view on a document is horrendous.[3] Similarly, again on the Macintosh, double-clicking the mouse button is a frequent form of user input, but there is no library routine available to detects double-clicks, leading to much code duplication and gratuitous differences between programs.*

With these considerations in mind, we set out to design a 'generic' window system interface.

- The interface should be general enough to suit the needs of many different programs. Thus, it should be reasonably rich in functionality, e.g., provide both textual and graphical output, handle keyboard, mouse and menu-based input, support multiple windows, etc.
- It should be simple to use. Including one header file and calling a small number of routines (with not too many parameters!) should suffice for the creation of a full-function window and the definition of its contents. As much as possible, the programmer should only be bothered with issues that matter from the program's point of view. In other words, the interface should be 'high level'.
- And most of all, it should be realistically portable; each potential feature should be weighed in the light of its implementability using different systems, including several popular micros.

The requirement of portability is necessarily both good and bad. It is bad because it can sometimes make an elegant solution unfeasible, imposing seemingly random restrictions. But it is good because it makes the design stick to reality, and limits it to the 'essence' of window systems, rather than allowing the designers to invent yet another incompatible paradigm. And sometimes the 'helicopter view' gained from looking at solutions chosen by vastly different window systems for a particular problem has shown the way to an entirely new view, simplifying it by generalization.

A large part of the paper is devoted to a detailed description of STDWIN's functionality from a programmer's point of view. First the 'core' of the package is described, explaining the basic output and input facilities; then some extra facilities are briefly discussed. Interspersed are comments on the rationale for particular solutions, some hints on the implementation, and warnings about non-portable uses. At the end the paper returns to the more philosophical issues: experiences, future developments, food for thought.

2. Description

2.1. Header file

Applications wishing to use STDWIN must place a line saying `#include <stdwin.h>` near the top of their source file(s). All user-visible external names defined in this header file start with `w` or `W`. external names used internally by implementations begin with `_w` or `_W`.

2.2. Initialization and clean-up

Before starting to use STDWIN, the initialization routine `winit()` must be called. Before exiting, the application should call `wdone()` to perform any necessary clean-up operations.

These calls can't be repeated; after `wdone()` has been called the application cannot call `winit()` again and 'return to life'.

2.3. Creating and destroying windows

A new window is created by calling `wopen(title, drawproc)`. *Title* is a string identifying the window to the user; it is usually displayed by STDWIN in the window's border, e.g., in a title bar. *Drawproc* is the address of the window's draw procedure (see next section), or NULL if the window is not to have a draw procedure. STDWIN windows look like windows in the usual style of the underlying window system, usually with a title bar, scroll bars etc. Position, size and other characteristics of the new window are determined by STDWIN (but see below).

`Wopen` returns a *window pointer*, of type `WINDOW *`, to be used to identify the window in subsequent

* In all fairness it should be said that the Macintosh is still miles ahead of most of its competitors, simply because there are at least standards for many aspects of the interaction between application and user, such as the placement of scroll bars, the use of double clicks or the shape of the mouse cursor.

operations. If creation of the window failed, a NULL pointer is returned.

STDWIN allows an application to have multiple windows open simultaneously. Implementations usually impose a limit on the number of open windows; when this limit is reached, `wopen` returns NULL, and the application should try to close other windows (or prompt the end user to close them).

A window is deleted permanently by calling `wclose(win)`. Windows can be deleted only by the application. The end user can send a request to the application to close a window, but the application may ignore the request or postpone its execution.

There is no explicit way to iconize a window (i.e., to temporarily close it, leaving an icon in its place). On systems where window iconization is built into the window system, STDWIN may support it silently; all the application notices is that no input is received for iconized windows.

2.3.1. Changing defaults

When a window is opened STDWIN determines a default size and position for it. Usually this is convenient for the application (which needn't have its own algorithm for placing multiple windows, for example), but sometimes finer control is desirable. Therefore, a number of default-setting routines are provided:

`wsetdefwinsize(width, height)`

Changes the default window size. This sets the net size, excluding borders, scroll bars etc.

`wsetdefwinpos(h, v)`

Changes the default window position. This default is usually not a constant but a dynamically computed value. The next opened window will be placed at *(h, v)*; the position of windows opened after that may be a more complicated function of *h* and *v*.

These routines may be called at any time; they affect only windows opened after their call. A negative or zero parameter restores the default for that dimension. Other values are clipped or rounded to reasonable and implementable values; these routines are best seen as giving hints to STDWIN, which may be ignored by some implementations.

2.4. The output model

A STDWIN window is a view on a possibly much larger area, a rectangle referred to as its *document*, in which the application draws its output. The document's size is chosen by the application, and can be changed at any time by calling `wsetdocsize(win, width, height)`. It is not limited by window or screen size, nor indeed by available memory; the entire document's contents are not stored directly. The end user has the freedom to 'pan' the window over the document's surface, using scroll bars or a similar mechanism. When a particular part of the document is to be visible in the window, STDWIN asks the application to repaint that area. It is not forbidden to draw outside the document, but the end user normally can't pan outside the document (unless the window is larger than the document).

There are two mechanisms for repainting: a low-level mechanism using DRAW events, and a higher-level mechanism using a *draw procedure*.

DRAW events are merged with the general event stream (see below); when no other events are in the event queue, STDWIN looks to see if there is any window needing a repaint, and if so, it passes a DRAW event for that window to the application. A DRAW event includes as additional information the rectangle that is to be repainted. The application should react by erasing and repainting that rectangle (or a larger part of the document).

Normally, however, windows have an associated *draw procedure*. This is a procedure (defined by the application) which knows how to draw the entire document, or any sub-rectangle of it. When STDWIN is about to generate a DRAW event for a window with a draw procedure, it prepares the window for drawing, erases the rectangle that needs repainting, and calls the draw procedure with the window and the rectangle as parameters. The advantage of this mechanism over DRAW events is the possibility for certain STDWIN implementations to clip the output to a smaller, non-rectangular area that really needs a repaint; also somewhat simpler event decoding logic for the application.

Usually, the end user controls which part of the document is visible in the window (by manipulating the scroll bars). However, there are times when an application wants to display a particular part of the document, e.g. to show the effect of a search operation. It can then call `wshow(win, <rectangle>)` to indicate that the given rectangle should be visible, if at all possible. STDWIN will check whether this is already the case, and if not, move the window with respect to the document to make it visible. There is also a lower-level call, `wsetorigin(win, <point>)` which makes the given point in the document the top left corner of the window.

When the application wants to change part of the document, it can directly paint the changes (after preparing for drawing in that particular window). However, it is often more appropriate to delay the actual painting until after other input has been processed. It is possible to tell STDWIN that a particular area of the document needs repainting by calling `wchange(win, <rectangle>)`. At the appropriate time, a DRAW event for this rectangle (possibly merged with other areas that need repainting) will be generated, or the window's draw procedure will be called.

When the repaint area is non-rectangular (e.g., it is the union of several rectangles), the application is asked to repaint the smallest rectangle that encloses the repaint area. This may occasionally cause more repainting than absolutely necessary, resulting in extra delays; since the repainting is limited to the window size, however, the costs won't be excessive in most cases. The choice was made here for a simple interface to the draw procedure, avoiding dynamic data structures. For the needs of the highest-demanding applications, an enquiry routine returning the exact repaint area may have to be be added (or a function telling whether a particular rectangle intersects the repaint area).

2.5. Drawing in a document

2.5.1. The coordinate system

STDWIN provides a single coordinate system per window. Coordinates are integers, with the X axis pointing right and the Y axis pointing down. In order to avoid confusion with other conventions, the axes are never called X and Y axis but h and v axis. H coordinates are always listed first. The origin (0, 0) is the top left corner of the document. Unit size equals pixel size on the screen; thus, documents inherit the screen's aspect ratio. Pixels on different machines can vastly differ in size; e.g., on alphanumerical terminals, pixel size might well equal character cell size. Therefore, applications should scale their drawings accordingly. STDWIN provides enquiry functions to tell the physical size of a pixel. An alternative approach, suitable for applications that display mostly text, is to scale the drawing accordingly to the dimensions of characters drawn on the screen. Text measuring functions are available for this purpose (see below).

2.5.2. Preparation for drawing

Since a picture is usually built out of a large number of calls to primitive drawing operations, it would be annoying to have to specify a window parameter on each call. STDWIN requires the application to say in which window it wants to draw before using any drawing primitives, by calling `wbegindrawing(win)`. After the drawing is done, the application should call `wenddrawing(win)`, telling STDWIN to flush the output to the screen.

In a draw procedure these calls are unnecessary; there, all drawing operations apply to the given window, and output is flushed when the draw procedure returns.

2.5.3. Graphical primitives

STDWIN currently provides a small set of graphical primitives. This set will be extended when the need arises. All primitives except `werase` and `winvert` draw in OR mode, i.e., they only add black pixels to the drawing and never erase pixels. Note that points are actually given as two integer parameters, h and v; rectangles are given as four integer parameters: left, top, right and bottom. Rectangles always refer to the area enclosed by infinitely thin boundary lines; e.g., the rectangle (0, 0, 1, 1) encloses a 1 by 1 square whose top left corner is the origin (0, 0).

Functions currently defined are:

`wdrawline(<point1>, <point2>)`

Draws a line from point1 to point2.

`wdrawbox(<rectangle>)`

Draws a box (i.e., a rectangle) inside the given rectangle.

`wdrawcircle(<point>, radius)`

Draws a circle with the specified radius around the given point as center.

`wpaint(<rectangle>)`

Paints the area inside the given rectangle black.

`werase(<rectangle>)`

Erases the area inside the given rectangle.

`winvert(<rectangle>)`

Inverts the pixels in the given rectangle.

`wshade(<rectangle>, percentage)`

Adds a shading pattern to the given rectangle, approximately making the given percentage of all pixels black. Thus, a percentage of 0 has no effect; a percentage of 50 sets every other pixel; a percentage of 100 is equivalent to `wpaint(<rectangle>)`. The exact shading pattern used is implementation-dependent, as are the values to which percentages are rounded.

2.5.4. Text drawing primitives

STDWIN supports the drawing of characters in a font which may be proportionally spaced, depending on the implementation. The exact shape and size of the characters are implementation-dependent. STDWIN does not use the notion of a 'base line' on which characters are drawn; rather, when a character or string is to be drawn, the top left corner of the box around it is given. All boxes have the same height, and a width appropriate for the character, so characters drawn in adjacent boxes 'look right'. This approach has the advantage that the application needn't be concerned with such font parameters as base line, ascent, descent and leading; it can simply start drawing characters at (0, 0) and they will come out 'right'. (This advantage for simplistic applications may turn into a disadvantage for programs wishing precise control over the placement of characters. In that case, additional enquiry functions will have to be defined to remedy this situation.)

The call `wdrawchar(<point>, character)` draws the given character with its top left corner at the given point. It returns the h coordinate of the right edge of the box in which the character is drawn; this is the 'natural' h coordinate for a character to be drawn next to it.

The call `wdrawtext(<point>, string, length)` draws the characters of the given string starting with the top left corner at the given point. *Length* indicates the number of characters in the string; if negative, the string ends with a NUL character. `Wdrawtext` returns the h coordinate of the right edge of the box in which the last character is drawn. Note that no special interpretation is given to characters like `'\n'` or `'\t'`; they may be displayed as spaces or funny graphics.

2.5.5. Text measuring primitives

The dimensions of characters drawn by the above functions depend on the font used. Future versions may implement font and size changes under application control; currently these are fixed by the implementation. For applications that want to know in advance how big the strings they are drawing will be, there are functions to measure text dimensions. Unlike the drawing primitives, the text measuring primitives and the style-changing primitives described in the next section can be called anywhere.

The following text-measuring functions are defined:

`wlineheight()`

Gives the vertical height of the boxes in which characters are drawn. This is the same for all characters, and the value delivered gives a 'natural-looking' line spacing when lines are drawn at v coordinates with increments of this value.

`wcharwidth(character)`

Computes the width of the box in which the given character will be drawn.

`wtextwidth(string, length)`

Computes the width of the box in which the string will be drawn. *Length* indicates the number of characters in the string; if negative, the string ends with a NUL character.

`wtextbreak(string, length, width)`

Computes the number of characters from the string that will fit in a box of the given width (in pixels). *Length* is interpreted as above.

2.5.6. Text style

Future versions of STDWIN will have to worry about mixing fonts, type sizes and text styles. Currently applications have no control over the font and size used, and can only control one aspect of text style; different window systems differ so much in their support of font names, font scaling, style combinations and so on, that it seemed wise to avoid these issues in the first version (however, some implementations have a way to influence the font, size or style used at initialization time). The only calls currently available are those to change between normal, black on white characters and inverse, white on black characters; this is needed to display the focus in the text-editing package (see below).

The call `wsetinverse()` sets the text style to inverse characters; the call `wsetplain()` reverts the text style back to normal. The text style is a global attribute, so draw procedures that change it should reset it to normal before leaving.

2.5.7. Scrolling

Applications like text editors often have a need for deleting a horizontal or vertical slice from their document; e.g., after a text editor has deleted a couple of lines, the remaining lines must be moved up in the document. Although it is theoretically possible to do this by calling `wchange` for the remaining part of the document (assuming the draw procedure knows that the v coordinates of the affected lines have changed), this often involves a lot of drawing which could have been avoided by applying a 'bit copy' operation as available in many systems, combined with only a little bit of redrawing (e.g., for lines 'scrolled in' from below the window border).

The call `wscroll(win, <rectangle>, dh, dv)` is provided to help in situation. It should be called outside the drawing procedure, where the call to `wchange` would otherwise be placed. If the particular STDWIN implementation supports the requested type of bit scroll operation, it will scroll the bits inside the given rectangle by an amount of *dh* to the right and by *dv* downward. (Negative values mean scrolling to the left or upward, respectively.) No bits outside the given rectangle are affected or used: bits 'scrolled out' of the rectangle will simply be thrown away; for the area that is to be 'scrolled in' from outside the rectangle, `wchange` is called internally. If the particular form of bit scrolling required isn't supported, the entire call is equivalent to `wchange(win, <rectangle>)`, relying on the normal repaint mechanism to update the window.

2.6. The input model

Interactive input is presented to the application in the form of *events*. Examples of events are 'a character has been typed' or 'the mouse button has been pressed'. Some other information generated asynchronously by STDWIN is also passed in the form of events.

Events are queued internally; the routine `wgetevent` gets the next event from the queue and passes it to the application. If the queue is empty, it waits until an event arrives first. (Certain events, like DRAW events, are not really queued but constructed on the fly when the queue is empty.)

Some applications don't want to wait when no event is ready, but do want to process events that are already queued. For such cases there is the alternative routine `wpollevent` which acts like `wgetevent` when an event is available from the queue, but returns immediately with a dummy NULL event when the queue is empty.

An event always applies to a particular window. This means that an application which has no window open is blind and deaf. When an application calls `wgetevent` in this state, it is terminated. Therefore, applications should make sure to always open a window before calling `wgetevent`.

STDWIN implementations may limit the size of the event queue; when the queue is filled up events may get lost without notification. (There is no way to prevent this, since the problem usually occurs in the underlying operating system.)

2.7. Events

Events are typically read in a 'main event loop', which might look something like this:

```
int stop= 0;
while (!stop) {
    EVENT e;
    wgetevent(&e);
    switch (e.type) {
        ...
    }
}
```

The variable `e` is called the *event record*. The information placed in the event record depends on the event type. For all event types, the type is available as `e.type`, and the window to which the event applies as `e.window`; additional information is listed with the individual event descriptions. This additional information is stored in a union named `e.u`, e.g., `e.u.character` for character input events.

For clarity, events are always referred to by their 'informal' names in this paper, e.g., MOUSE DOWN. The actual constants defined by STDWIN are derived from the informal name by prepending `WE_` and replacing spaces by underscores, yielding, e.g., `WE_MOUSE_DOWN`.

Events can be classified as mouse events, other user input events and STDWIN-generated events.

2.7.1. Mouse events

Mouse events are generated when the user presses a mouse button inside the visible part of a document displayed in a window. There are separate event types for a press of a button, moves while a button is held down, and a release of a button. The position of the mouse cursor at the time the event was generated is reported in (`e.u.where.h`, `e.u.where.v`). The button number (1, 2 or 3 on a three-button mouse; always 1 on a one-button mouse) is reported in `e.u.where.button`.

Mouse events allows easy detection of *multiple clicks*, to which many applications want to assign a special meaning. Successive presses on a mouse button are considered to be part of a click sequence if they are 'close together' in space and time. When a mouse button is pressed, STDWIN checks whether it is close enough to the previous press to be considered a continuation of the same click sequence, and if so, notes the number of the current click in `e.u.where.click`. A click that is unrelated to previous clicks has click number 1; a following related click has click number 2, the next one has number 3, and so on, until the mouse is moved too far away or the user waits too long, in which case the click number is reset to 1 at the next mouse event. This way of reporting multiple clicks requires no delay to see whether a click is part of a multiple-click sequence; mouse events are reported as soon as they happen.

Not all STDWIN implementations run on machines whose mouse has more than one button; it is therefore unwise to write an application which can perform certain operations only through buttons 2 or 3. If multiple buttons are held down simultaneously, only events for the button pressed first are generated.

The mouse event types are MOUSE DOWN for a button press, MOUSE MOVE for a move of the mouse cursor while a button is still depressed, and MOUSE UP for a button release. The click number for MOUSE MOVE events is always zero. In order to prevent filling up the event queue, multiple MOUSE MOVE events may be collapsed to a single event, giving only the last mouse position. When the user moves the mouse outside the window with a button held down, the mouse remains associated with the window, and its position is reported relative to the origin of the window's document. The click number for a MOUSE UP event is the same as that of the corresponding MOUSE DOWN event if the mouse wasn't moved too far from its original position, or zero if it was moved further (and in this case this event is the end of its click sequence).

2.7.2. Other user input events

CHAR
: The user has typed a character at the keyboard. Its ASCII value is reported in `e.u.character`. Note that some special keys (like RETURN, TAB, BACKSPACE) do not send CHAR events but COMMAND events.

COMMAND
: This event is sent for special keys on the keyboard, and for certain special actions recognized by STDWIN. Some keys do not generate CHAR events but COMMAND events, because they do not send

the same ASCII code on all keyboards (e.g., Enter), or because there are no standard ASCII codes for them (e.g., arrows and function keys). A code telling which special command was meant is reported in `e.u.command`. Possible values represent the following keys and standard actions: CANCEL, TAB, RETURN, BACKSPACE, LEFT, RIGHT, UP, DOWN and CLOSE; this list may be extended in the future. The constants are actually called `WC_CANCEL` etc.

CLOSE is to be interpreted as a request to close the window; the key or other action that generates it is system-dependent. The application should close the window, possibly after verifying that any changes the user has made to the file displayed in the window have been saved, in which case it may ignore the request, or put up a dialogue box asking what should be done to the file.

MENU

A menu item was selected. The menu id and item number of the selected item are reported in `e.u.m.id` and `e.u.m.item`; menu items are numbered starting at 0 (see below for the definition of menus).

The interaction technique used to select menu items is not defined by STDWIN; a suitable technique is chosen by each implementation, e.g. pop-up, push-down or permanently present menus. Keyboard shortcuts are usually also available. The application cannot distinguish between the various ways of selecting a particular menu item; all it sees is which item is selected.

2.7.3. STDWIN-generated events

NULL

Nothing happened. This is a dummy event reported only by `wpollevent` when the event queue is empty.

ACTIVATE

A window has been 'activated'. This is usually caused by the end user selecting an inactive window with the mouse. Only one window can be active at any time. This usually means that all subsequent keyboard input applies to the active window; some applications want to change the highlighting of selected objects in a document when its window is active. (Highlighting of the window's title, etc. is done automatically by STDWIN.) After a window is opened, the first event applying to it is usually an ACTIVATE event (because windows are opened in an unactivated state). Applications needn't monitor ACTIVATE events if all they want is determining to which window keyboard input applies; the relevant window is reported with each event in `e.window`.

DEACTIVATE

A window has been 'deactivated'. This usually occurs just before another window is activated. In many implementations of STDWIN it is possible for the user to activate a window not belonging to the current application; in this case the current application receives only a DEACTIVATE event until one of its windows is reactivated. Note that closing a window does not generate a DEACTIVATE event for it, since the window has already disappeared by the time the application can call `wgetevent`.

SIZE

A window's size has changed. This is usually done by the user explicitly resizing the window; in some ('tiling') STDWIN implementations it can also be caused by opening or closing other windows.

Some applications want to format their documents to fit exactly in the window. SIZE events make it possible for such applications to monitor window size changes. The new window size is not reported in the event record; the application can use the enquiry function `wgetwinzize` for this purpose (see below).

Note that window moves don't generate events (except possibly DRAW events).

DRAW

This event is reported only for windows without an associated draw procedure. It means that part of the window needs to be repainted. The smallest rectangle enclosing the area to be repainted is reported in `e.u.area`, a struct with four fields `left`, `top`, `right` and `bottom`.

TIMER

The window's alarm timer has gone off. For each window, an alarm may be set with the call `wsettimer(win, dsecs)`. The alarm will go off, causing a TIMER event, aproximately *dsecs/10* seconds in the future (*dsecs* meaning deciseconds). Only one alarm per window is maintained; a new call overrides the previously set alarm. A value of 0 cancels the alarm. Timer values may be rounded up to whole seconds by some implementations. The maximum timer value that is guaranteed to be supported is 32000 dsecs.

2.8. Pushing events back

Occasionally, an application may want to postpone processing of an event till later. E.g., a subroutine may be getting events in a loop until it receives an event which shouldn't be handled locally but in the main event loop. The routine `wungetevent(&eventrecord)` allows an event to be pushed back onto the event queue; the next call to `wgetevent` or `wpollevent` will report the event just pushed back. Only a single event can be pushed back (some implementations save the pushed back event in a separate buffer). It is possible to modify the event before pushing it back, or to synthesize events entirely.

2.9. Getting and setting the active window

A pointer to the active window is returned by the function `wactive()`. The application can also make a different window active by calling `wsetactive(win)`. This call does not take effect immediately; some time in the future, a DEACTIVATE event for the currently active window and an ACTIVATE event for the newly activated will be received.

2.10. Menus

Most window systems provide a simple way to set up and manipulate menus, in their simplest form lists of text strings which can be selected by the user by clicking on a string with the mouse. Menus may 'pop up' when a particular mouse button is pressed in a particular screen area, or be 'pulled down' from a 'menu bar', etc. STDWIN provides a consistent, simple way for the application to interface with standard menus, or with menus defined entirely by the STDWIN library (if the window system provides no usable menus).

A *menu* contains a number of *items*, numbered starting at 0. A menu has a *title*, a text string displayed to identify the menu to the user, and a *menu id*, a small positive integer identifying the menu to the application. Each item contains a text string, an optional *check mark* (which may be set by the application to indicate whether an option controlled by a menu item is active), and can be *enabled* or *disabled*. Only enabled items are selectable. When the user selects an enabled item, a MENU event is queued containing the menu id and item number in the event record. Because of the way events are queued, it is possible to receive MENU events for disabled menu items (if the selection was made before the menu item was disabled); applications should be prepared to receive spurious menu selection events.

A menu is created by a call to `wmenucreate(id, title)`; this returns a *menu pointer* which must be used for all further manipulations with the menu. *Id* is the menu id, which should be in the range [1..255]. Menu ids should be unique within an application.

Initially, a menu contains no items. Items are added by calling `wmenuadditem(mp, text, shortcut)`. The new item's number equals the number of items in the menu before this call; it is returned as the function value. *Mp* is the menu pointer; *text* is the item text. The item is initially enabled and unchecked. *Shortcut* is a character used to construct a 'keyboard shortcut' for the menu item; −1 means the item is not to have a shortcut. (The interpretation of keyboard shortcuts is implementation-dependent. In a typical STDWIN implementation, a menu item with shortcut 'X' might be selected by typing ESC-X or Meta-X (but not Control-X). All printable characters are acceptable as shortcuts, but on some systems lower case and upper case are indistinguishable.) Adding an item with an empty string as text adds a disabled 'separator' item.

The text of an existing menu item can be changed by calling `wmenusetitem(mp, number, text)`. Items can be enabled or disabled by calling `wmenuenable(mp, number, flag)`. The check mark for an item can be set or cleared by calling `wmenucheck(mp, number, flag)`.

A menu can be deleted by calling `wmenudelete(mp)`. Note that individual menu items, once added, cannot be removed, nor can new items be inserted in the middle. This is due to restrictions in many window systems' menu interfaces; usually menus are sufficiently static that it doesn't matter.

For a menu's items to be selectable, the menu must be attached to a window and the window must be activated. Normally, STDWIN automatically attaches all menus to all windows, so all menus become selectable as soon as the first window is activated. To change this behaviour, the call `wmenusetdeflocal(TRUE)` causes subsequently created menus to be 'local', requiring explicit attachment and detachment. The call `wmenuattach(win, mp)` attaches the menu *mp* to the window *win*.

The call `wmenudetach(win, mp)` reverses this effect. A menu may be attached to multiple windows; multiple menus may be attached to a window. After calling `wmenusetdeflocal(FALSE)`, future menus will be 'global' again, i.e., automatically attached to all (existing and new) windows.

3. Additional facilities

3.1. Enquiry functions

Some enquiry functions are available to interrogate the system state.

`wgetscrsize(&width, &height)`
  Returns the screen size measured in pixels into the integer variables whose addresses are passed.

`wgetscrmm(&mmwidth, &mmheight)`
  Returns the approximate screen size measured in millimeters. By combining this information with the outcome of `wgetscrsize`, pixel size and aspect ratio can conveniently be computed. In some (most?) implementations, the numbers returned may be approximations or guesses.

`wgetwinsize(win, &width, &height)`
  Returns the size of the drawable area of a window, measured in pixels. (Due to the presence of borders, a maximally-sized window is usually smaller than the screen.)

3.2. The text caret

In documents that deal with text it is often useful to have some form of 'text cursor', indicating the position where characters typed at the keyboard will be inserted. The call `wsetcaret(win, h, v)` causes a 'caret' to appear just to the left of the character position (*h*, *v*) in the document. The caret appears immediately before any character that would be drawn by `wdrawtext(h, v, ...)`. The caret has a system-defined shape; it is often a blinking vertical bar.

Each window has its own caret; the caret in the active window may be the only one that is visible, or it may blink while the carets in other windows are static. At any time a window has at most one caret; the old caret is removed when a new one is specified. The caret can be removed altogether with the call `wnocaret(win)`.

3.3. Dialogue tools

A *dialogue box* is a 'mini-window' containing a simple message or question, and requiring the user to respond, e.g. by pressing a key or clicking the mouse in a particular area. As long as the dialogue box is present, the application is blocked. After answering the question or acknowledging the message, the dialogue box disappears and normal interaction with the application continues. Dialogue boxes may be presented even when no windows are open yet. The following calls put up dialogue boxes and wait for a response:

`wmessage(string)`
  Displays a message and waits until the user acknowledges it. The precise form of acknowledgement required is implementation-dependent; it could be pressing the Return key or clicking an 'OK button' with the mouse.

`waskstr(question, replybuf, buflength)`
  Displays a question and waits until the user has finished typing a reply. The initial contents of the reply buffer are used as a default reply. The function normally returns TRUE; if the user aborts the dialogue (e.g., by pressing the CANCEL button) it returns FALSE.

`waskync(question, dflt)`
  Displays a question which gives the user the possibility to answer with Yes, No or Cancel only. The return value is 1 (Yes), 0 (No) or -1 (Cancel). *Dflt* is the suggested (default) return value.

`waskfile(prompt, replybuf, buflength, new)`
  Displays a dialogue box asking for a file name. *Replybuf* initially contains a default or suggested file name. The boolean parameter *new* specifies whether a new (not yet existing) or old (existing) file is required. When a new file is asked for, the user may specify an existing file, but in this case explicit permission is asked to overwrite it. The function returns TRUE, or FALSE if the user aborts the dialogue. The file name is returned in a form acceptable to the STDIO function `fopen`. STDWIN

implementations may provide additional support, e.g. file name completion or file system browsing; the fact that some systems provide elaborate standard file-selection dialogues (which is highly appreciated by the end users) was a strong motivation to include this function in STDWIN.

`wperror(string)`

Displays an error message similar to that printed by the standard C function *perror*(3), and waits for an acknowledgement as for `wmessage`.

It should be noted that `waskstr` is the most general of the above functions; in theory, versions of the others can be implemented with the help of `waskstr` and other existing tools.

3.4. The text-editing package

The *text-editing* package is a set of routines implemented entirely 'on top of' STDWIN, without using any implementation-dependent functions or data structures. The availability of this package is important because it provides a standard way to tackle the non-trivial problem of editing multi-line text blocks. It is clearly influenced by the TextEdit routines available in the Apple Macintosh's ROM Toolbox (but contains only original code).

The text-editing package displays a paragraph of text in a rectangle of a given width, breaking the lines at spaces between words as necessary. It gives the application complete control over what happens to the text, but provides an easy way to handle user input intended to edit it.

The call `tecreate(win, <rectangle>)` returns a pointer to a text-editing block at the specified position in the given window's document. (A text-editing block is not a portion of the document but a data structure.) Any number of text-editing blocks may be created, although usually at most one block per window should be editable at any time.

Initially, the block contains no text. The call `tesettext(tp, string)` sets the text to be edited, replacing any existing text in the block. The call `tegettext(tp)` returns a pointer to the text string (which remains valid only until the next call to a text-editing routine).

The text block is not automatically drawn. When a text-editing routine changes the edited text (or other aspects of its appearance), it calls `wchange` for the appropriate area of the window; the window's draw procedure should call `tedraw(tp)` for each block which overlaps the repaint area.

Besides the edited string, a text-editing block contains a *focus*, indicating which text is selected for deletion or at which position new text will be inserted. The focus can be set by the application with the call `tesetfocus(tp, first, last)`, telling that the characters in the range [first..last-1] are selected, or, if first equals last, that the text insertion point is at that position (characters are counted starting at 0). If the focus is an insert position, the window's caret is set at that position in the document. Text can be inserted at the focus (replacing its previous contents) by calling `tereplace(tp, string)`; specifying an empty string deletes any text in the focus.

The simplest way to let the user edit the text in a text-editing block is to call `teevent(tp, &eventrecord)` for each event. This call returns TRUE if the event is applicable to the text-editing block (e.g., it is a CHAR event, or a mouse click within the block's bounding rectangle), and in that case the event is processed by the text-editing package (e.g., a character is inserted, or the focus is moved to the point where the mouse was clicked). If the event is not applicable to the particular block, the function does nothing and returns FALSE; in this case the application should further decide what to do to the event. Of course, the application is free to decide whether to offer an event to a text-editing block at all; e.g., it might have a different interpretation for the Return key (for which the text-editing package inserts a new-line character in the text string).

There are more text-editing calls, e.g. to move a text-editing block to a new position, to enquire about the focus, to perform individual editing operations, to ask for the height of the rectangle minimally needed to display the text entirely, etc.

The following call displays a text string in exactly the same way as the text-editing package would do (breaking it into lines at the same places, etc.), but without creating a text-editing block, and thus without a focus: `wdrawpar(<point>, string, width)`. It returns the v coordinate of the bottom of the text paragraph. To compute the height of a text paragraph thus drawn without actually drawing it, one can call `wparheight(string, width)`.

4. A complete example

The program below is a complete STDWIN application. It is presented here to give a feel for the use of some of the routines described above. The program displays a window in which a text-edit block is placed; all events recognized by the text-edit package are handled correctly, and so are several ways of quitting. Other events are ignored.

```
#include <stdwin.h>

TEXTEDIT *tp; /* Global so drawproc can reference it */

void drawproc(w, left, top, right, bottom)
    WINDOW *w;
    int left, top, right, bottom;
{
    tedraw(tp);
}

main()
{
    MENU *m;
    WINDOW *w;
    int stop;
    int width, height;

    winit();

    m= wmenucreate(1, "Sample");
    wmenuadditem(m, "Quit", 'Q'); /* Item 0 */

    w= wopen("Sample window", drawproc);
    wgetwinsize(w, &width, &height);
    tp= tecreate(w, 0, 0, width, height);

    stop= 0;
    while (!stop) {
        EVENT e;
        wgetevent(&e);
        if (teevent(tp, &e))
            wsetdocsize(w, width, tegetbottom(tp));
        else {
            switch (e.type) {
            case WE_COMMAND:
                if (e.u.command == WC_CLOSE || e.u.command == WC_CANCEL)
                    stop= 1;
                break;
            case WE_MENU:
                if (e.u.m.id == 1 && e.u.m.item == 0) /* Quit */
                    stop= 1;
                break;
            }
        }
    }
    wclose(w);
    wdone();
    exit(0);
}
```

5. Experiences

Five distinct STDWIN implementations have been created so far: for the Apple Macintosh, for the Whitechapel MG-1, for X version 11, for the Atari ST, and a subset for alphanumeric displays (which runs both under Unix and MS-DOS).

Once a STDWIN version for a target system is available, application portability is high. Most portability problems that crop up (besides the usual problems like word size, byte order, data alignment or following NULL pointers) have to do with differences in other parts of the operating system interface, e.g. use of the file system. One portability problem encountered with the STDWIN interface was that some programs developed for alphanumeric terminals expected a fixed-width font; in general most problems were caused by insufficiently precise specification of STDWIN.

The time needed to create a STDWIN version for a particular target system is moderate. An experienced C programmer who did not know anything about STDWIN or the Atari ST in advance created a working Atari ST version in two months. So far, each version has been created more or less from scratch (except for the common parts like the textedit package). We have now gained enough experience with different target systems to be able to create an intermediate layer containing code which remains more or less constant between target systems.

6. Future developments

To date, the applications that use STDWIN have been mostly text-based. Undoubtedly, this has influenced the direction of development of drawing facilities in STDWIN. It is sufficiently easy to add graphical primitives to an implementation, though, that we expect to add several as demand grows, e.g., bitblt, clipping, line styles, filling. The existing facilities set a sort of standard for the form of future ones. The requirement that they be implementable on top of a large variety of window systems will ensure that only more or less generally accepted primitives will be included in STDWIN; a useful sort of conservatism for a package that wants to enhance application portability.

Besides the need to add more drawing primitives, there are several areas where STDWIN requires, and will probably get, extensions: fonts, sizes and styles; the mouse cursor; drawing in off-screen bitmaps (not associated with a window); error handling (which is currently virtually absent); event queue manipulations and an 'event mask'; 'clipboard' or 'cut buffer' operations.

A development in a different direction, independent of the addition of graphical primitives, may be the addition of more toolboxes built on top of the exiting facilities, like the text editing package. Tools are needed to manipulate higher-order graphical objects, to implement specific interaction techniques, to provide 'canned applications' like text-editing windows, etc.

A third, potentially very useful, extension would be the addition of drawable 'borders' to the window that aren't scrolled together with the document. In such borders, interaction tools could be placed like palettes and buttons, or rulers around the document. The design of such an extension should be the topic of further research, in order to achieve the largest possible generality.

References

1. Leo Geurts, Lambert Meertens, Steven Pemberton, *The ABC Programmer's Handbook,* CWI, Amsterdam (to be published in 1988).
2. Mike O'Dell, "What They Don't Tell You About Window Systems," in *EUUG Conference Proceedings, September 1987, Dublin, Ireland.*
3. Apple Computer, *Inside Macintosh,* Addison-Wesley, Reading, Mass. (1985).