Windows for ML.
Revised.

David C.J. Matthews

16 December 1987

This revised version of the windows system incorporates various changes and suggestions. It is still open to change but I hope it will soon converge on a standard. Any suggestions are welcome.

The main aim is that it should be reasonably portable while allowing ML programmers to do most of what they want directly in ML. It has been implemented on the Sun but there may have to be a few changes for other implementations. An implementation under X-windows is probably the next step.

The design generally follows the **Bitmaps Proposal** of Mike Fourman who in turn based it on bitmaps in Luca Cardelli's language, Amber. It has been revised in the light of suggestions from Mike Fourman and others at Abstract Hardware and IST.

1 Figures

```
structure Figures :
    sig
    datatype point = point of {x: int, y: int}
    val origin : point
    val add_pt : point * point -> point
    val sub_pt : point * point -> point
    val scale_pt : point * int -> point

    datatype line = line of point * point

    datatype rect =
```

```
   rect of {bottom: int, left: int, right: int, top: int}
(* Points in the rectangle *)
val left : rect -> int and right : rect -> int
val top : rect -> int and bottom : rect -> int
(* Height and width *)
val height : rect -> int and width : rect -> int

(* create_rect - make a rectangle enclosing two points.
   The pixel in the bottom left is in the rectangle,
   the pixel in the top right is not. *)
val create_rect : point * point -> rect
val top_left : rect -> point
val top_right : rect -> point
val bottom_left : rect -> point
val bottom_right : rect -> point

val left_of : rect * rect -> bool
val above : rect * rect -> bool
val sects : rect * rect -> bool (* Rectangles intersect *)
val offset : rect * point -> rect (* Move a rectangle *)
val size : rect -> rect (* Moved to origin *)
val union : rect * rect -> rect (* Surrounding rectangle. *)
val intersection : rect * rect -> rect
exception intersection : unit
val within : point * rect -> bool
end
```

The `Figures` structure contains types implementing points, lines and rectangles, and operations on them. The origin is assumed to be at the lower left hand side of the screen so rectangles are created with the value of bottom less than or equal to top and left less than or equal to right. To allow for empty rectangles the bottom left point is assumed to be in the rectangle but the top right is not.

2 Bitmap

```
structure Bitmap:
   sig
   type bitmap and mode

   val create_bm: Figures.rect -> bitmap
```

```
exception bitmap: string

val draw:
   {source: bitmap, dest: bitmap, from: Figures.point,
    to: Figures.rect, draw_op: mode} -> unit
val fill:
   {dest: bitmap, to: Figures.rect, draw_op: mode} -> unit
val draw_line:
   {dest: bitmap, line: Figures.line, draw_op: mode} -> unit
val draw_pattern:
   {source: bitmap, dest: bitmap, from: Figures.point,
    to: Figures.rect, draw_op: mode} -> unit
val region: bitmap * Figures.point * Figures.rect -> bitmap
val bounds: bitmap -> Figures.rect
val init_bitmap: bitmap * int -> unit

datatype font = font of int
val draw_text:
   {message: string, dest: bitmap, font: font,
    where: Figures.point, draw_op: mode} -> unit
val font_default : font
val char_box: font -> Figures.rect

val mode_src: mode and mode_dst: mode
val mode_set: mode and mode_clear: mode
val mode_not: mode -> mode
val mode_or: mode * mode -> mode
val mode_and: mode * mode -> mode
val mode_xor: mode * mode -> mode

(* The kind of access allowed. *)
val readable : bitmap -> bool
val writeable : bitmap -> bool
val freeze: bitmap -> unit
end
```

The `Bitmap` structure contains operations for operating on bitmaps. Bitmaps are rectangular areas of bits which can represent pixels. There are two kinds of bitmap: Those which are associated with areas of the screen which can only be written to, and bitmaps in memory which can be read from or written to.

The bounding box for a bitmap, returned by `bounds`, gives the height and width and also an origin for drawing operations on the bitmap.

A bitmap of a given size and origin is created with `create_bm`. This creates a new bitmap in memory initialised to zeros. A bitmap can also be made from an area of an existing bitmap. `region` creates a bitmap which shares the same bits but with a new origin and rectangle. The point is the origin of the region relative to the parent, and the rectangle is the area of the region. The region must lie entirely within the parent. Drawing operations on a region are clipped to its edges. As well as providing a clipping window a region can also be used to shift the origin of a bitmap without changing its size.

The primitive drawing operation is `draw` which copies from an area of one bitmap to an area of another. `from` is a point relative to the `source` and `to` a region in the `dest`. The way in which the copying is done is controlled by the `mode`. Each bit in the destination is set to a value formed from the corresponding source bit and the previous value of the destination bit by application of the appropriate function.

`fill` can be used to fill an area. It is the same as `draw` with a source which is all zeros.

`draw_line` draws a line using the appropriate drawing operation. The line is clipped to the region of the bitmap.

`draw_pattern` fills an area of the destination with copies of the source laid side-by-side and end-to-end. The `from` point allows the pattern to be aligned in the destination.

`draw_text` writes a string in a given font. The fonts are identified by integers and at present there is only one. `where` gives the location of the *origin* of the first character in the string. The y co-ordinate is the position of the base-line for the characters so characters with descenders will go below this line. All characters in a font must have the same width and the size of a character can be found with `char_box`. The rectangle returned is the size of a character and is located relative to the base-line. Its top line will have a positive y co-ordinate, being above the base-line, and its bottom line will have a negative y co-ordinate to allow for descenders.

`init_bitmap` allows a bitmap to be filled in with bits from a number so that a bitmap produced externally can be read in.

It is useful to be able to freeze the data in a bitmap in memory once it has been created so that no changes can be made to it. `freeze` can be used to do this. Once frozen a bitmap can only be used in the source of a drawing operation.

3 Windows

```
structure Window:
   sig
   type window
   (* Get a default value for the shape of a window. *)
   val next_win_posn : unit -> Figures.point * Figures.rect

   (* Create a top-level window. *)
   val create_win : Figures.point * Figures.rect -> window
   (* Create a sub-window *)
   val sub_win : window * Figures.point * Figures.rect -> window
   (* Get rid of a window and any sub-windows. *)
   val destroy : window -> unit

   (* Get and set the size and position of the window. *)
   val get_origin : window -> Figures.point
   val set_origin : window * Figures.point -> unit
   val get_bounds : window -> Figures.rect
   val set_bounds : window * Figures.rect -> unit

   .....

   end
```

The `Window` structure is the most complex part of the windows package. Windows are thought of as forming tree structures with top-level windows as roots. Sub-windows can be formed from parts of other windows and moved around within them.

`create_win` and `sub_win` make windows with a given origin position relative to the parent, and a given rectangle relative to the origin. The position of the rectangle on the screen or in the parent can be changed, either by changing the origin or by changing the bounds. Changing the bounding rectangle can resize the window.

`next_win_position` returns a default position and rectangle for a new top-level window. Repeated calls may return different values so that windows stack up rather than completely covering one another.

```
structure Window:
   sig
   ....
   (* Get the current bitmap. *)
   val image : window->Bitmap.bitmap
```

```
   (* Cursor for the window. *)
   type cursor (* =  Bitmap.bitmap * Figures.point * mode *)
   val cursor_size : Figures.rect
   (* A default cursor. *)
   val arrow : cursor
   (* Set the cursor *)
   val set_cursor : window * cursor -> unit

   ...
   end
```

Drawing to the window is done by writing to the image bitmap of the window. The drawing takes effect immediately onto the area of the window but is not retained so if a window is moved on the screen it may be necessary to redraw some or all of the image.

Each window and sub-window has a cursor which is displayed when the mouse is in the window. This can be set by `set_cursor`. A cursor is a readable bitmap of size `cursor_size`, together with a hot-spot and a mode in which to draw it. The hot-spot is a point relative to the cursor which is treated as the actual cursor position.

```
structure Window:
   sig
   ....
   val is_top : window -> bool
   (* Set the name stripe *)
   val set_name : window * string -> unit

   (* Convert between open and iconic forms. *)
   val close_win : window -> unit
   val open_win : window -> unit
   val is_open : window -> bool
   (* An icon which can be drawn. *)
   val icon_size : Figures.rect
   val poly_icon : Bitmap.bitmap
   (* Set the icon for the window. *)
   val set_icon: window * Bitmap.bitmap -> unit
   ...
   end
```

Certain operations can only be performed on a top-level window. On the Sun a top-level window is created with a name stripe at the top and a border. Mouse button clicks in this area can be used to ask for the window to be moved or resized. Other implementations might have resize or drag regions.

A top-level window can be closed into an inactive form in which it is represented by an icon. The icon is a 64 by 64 bit readable bitmap. `poly_icon` is a picture for the Poly system and is the default. The window can be made iconic using `close_win` and opened by `open_win`. Drawing to a closed window has no effect and changing its origin or bounds will only take effect when it is reopened. The system may allow the icon to be moved on the screen but this will not be reported to the ML program.

```
structure Window:
   sig
   ....
   (* Events which can be returned. *)
   datatype key_press =
      key_down | key_down_again | key_up | key_click
   datatype win_event =
      win_redraw of Figures.rect list | win_state |
      win_move of Figures.point | win_resize of Figures.rect |
      win_enter | win_exit | win_pointer of Figures.point |
      win_key of
         {key: string, location: Figures.point, state: key_press} |
      win_mouse of
         {button: int, location: Figures.point, state: key_press}

   type time_stamp (* Time when an event occurred. *)
   val before: time_stamp * time_stamp -> bool

   val read_win : window -> win_event*time_stamp (* Get an event *)

   (* Event set - An event will only be read if it is the mask for
      that window. Otherwise it will be passed on to the parent. *)
   datatype event_set =
       event_set_key | event_set_move | event_set_enter_exit |
       event_set_resize | event_set_pointer | event_set_state |
       event_set_mouse;
   (* Get and set the event mask. *)
   val get_mask : window -> event_set list
   val set_mask : window * event_set list -> unit
   ...
   end
```

Events may occur in the window and are read by `read_win`. A time stamp is returned with an event to allow events on different windows to be put into the correct sequence.

Each window is a rectangular area and the ML program can draw over the whole of it. What appears on the screen may only be part of this because other windows may be overlapping it. The system clips all the drawing operations to the area which is actually exposed. If one of the overlapping windows is now moved to expose more of the ML window the ML program will have to fill in the new area. A `win_redraw` event indicates that the image should be redrawn. It includes a list of rectangles which should be filled in to completely redraw the image. The ML program can use this information to avoid redrawing the whole of a window if the cost of redrawing is high. This event is also generated if the window size is changed by a call to `set_bounds.`

The `win_move, win_redraw` and `win_state` events are all produced as a result of the user requesting changes to the window. On the Sun this is done using mouse clicks in the border of a top-level window. They are all treated as requests to the ML program to make the appropriate change, but it is up to the program to interpret and respond to the request. The user cannot force a particular action. The `win_move` event requests that the origin of the window be moved to the given point. The effect of setting the origin there will be to move the window without changing its size or affecting drawing within the window. The `win_resize` event requests a change in the shape of the window. Responding by setting the bounds to the given rectangle will keep the origin at the same place on the screen but change the area being displayed. This response would be appropriate for graphics where one might want to look at a particular area while keeping the origin in the same place on the screen. In a window containing text it might be more appropriate to keep the origin at a fixed place within the window and so the correct response to a `win_resize` would be a combination of calls to `set_origin` and `set_bounds.` The `win_state` event requests the window to be opened or closed as appropriate.

The `win_enter` and `win_exit` events indicate that the mouse has entered or left the window. Within the window the position is returned by `win_pointer` events. The co-ordinates used are relative to the window. There is no way on the Sun to explicitly read the location of the mouse except by tracking events.

Key and mouse button presses are returned as `win_key` and `win_mouse` events. The buttons on the Sun mouse are numbered zero, one and two counting from the left. As well as the key or button pressed the position of the mouse at the time is returned. Some applications make use of repeated clicks of keys, particularly of the mouse buttons. `key_down` and `key_down_again` indicate that the key or button has been depressed and `key_up` and `key_click` that it has been released. `key_down_again` is given instead of `key_down` if that key had been released a "short" time before. Similarly `key_click` is given instead of `key_up` if a key is pressed and released quickly. In all cases when a key is pressed and released an event will be generated for both the press and release. What constitutes a "short" time depends on the system.

Events are directed at the window which underlies the mouse, but a window can choose to mask out some or all events, in which case they will either be ignored, in the case of `win_enter` and `win_exit` or passed to the parent window, in the case of `win_pointer`, `win_key` and `win_mouse`. `win_redraw` cannot be masked out. Which events a window is interested in is controlled by an event mask which can be read and set by `get_mask` and `set_mask`. Initially the mask is empty so all events are either ignored or passed to the parent. The set of events can be combined and events added to the mask or removed from it.

```
structure Window:
   sig
   ....
   val select_item_from_menu :
      window * string list * Figures.point -> int
   end
```

The final operation in the `Windows` structure allows a menu to be displayed and an item selected from it. The menu is a list of strings which are displayed and the number of the string selected is returned. The strings are counted from 1 and a result of 0 means that no item was selected. The menu is displayed at a position relative to the window. This will normally be the location of the mouse when a button is pressed to bring up the menu.

4 Discussion

There are a few outstanding questions which need to be thought about which may have a bearing on how easily the system can be ported.

A major change since the previous document is that redrawing a window after it has been uncovered is now the responsibility of the ML program. The previous version had a bitmap in memory for each window and all drawing was done to this. Redrawing the screen after damage was the responsibility of the system. It is still open to an implementation to do this in which case the `win_redraw` event would never be returned except when the window size is changed. The advantage of having the ML program redraw the window is that it may require less memory to hold a higher-level data structure than a complete bitmap. There is a trade-off in terms of CPU time but the ML program can always keep a memory bitmap and draw that to the screen each time.

A more difficult problem has to do with icons. On the Sun an icon is simply a window which represents a closed top-level window. It can be moved on the screen but it behaves like any other window, covering some windows and being covered by others. On the Macintosh however an icon has a more active role. Moving an icon into a window or onto another icon is interpreted as an operation involving both. A file is deleted by moving its icon to the icon representing a rubbish bin. This interpretation of an icon is difficult to implement on the Sun and probably other machines as well and so it has not been considered.

Sub-windows are areas within other windows which can be moved and resized within them. They are drawn separately from their parent window. They could be implemented using regions. Should they be primitive?

Menu selection is provided as a primitive operation. It could be implemented as, say, a sub-window, but that would probably be slower. Also the menu selection function as implemented can go over the boundary of the window. A more general function allowing pictures as well as strings is probably required.