F.G.H.
7/2/85

LispCourse #40: Using the Display: Bitmaps, DisplayStreams, & Windows

Introduction

Interaction with the display from Interlisp-D programs involves 3 basic types of Interlisp objects ž *bitmaps*, *display streams*, and *windows*.

A ***bitmap*** is a two-dimensional array of bits (1s and 0s) in your computer's memory.

You deal with bitmaps by setting specific bits to 1 or 0, or by copying rectanglular arrays of bits around from bitmap to bitmap.

The Interlisp-D display screen is just a special bitmap (called the SCREENBITMAP) that is 808 bits high and 1024 bits wide that is displayed on the screen with every 1-bit is black and every 0-bit is white (or vice versa if you change the VIDEOCOLOR parameter as per page 19.7 in the IRM).

Display streams represent an interface that allows you to deal with bit maps at a level higher than bits, i.e., in terms of characters, fonts, lines, circles, regions, etc.

You can call functions like DRAWCURVE on a display stream. The result will be a curve drawn on the destination bitmap of that display stream.

A display stream is a datatype that represents some destination bitmap. Stored in this datatype are fields that contain things like the current font to be used for writing characters to the destination bitmap, the X-Y position of the "cursor" in the bitmap, left and right margins for writing and drawing on the bitmap, etc.

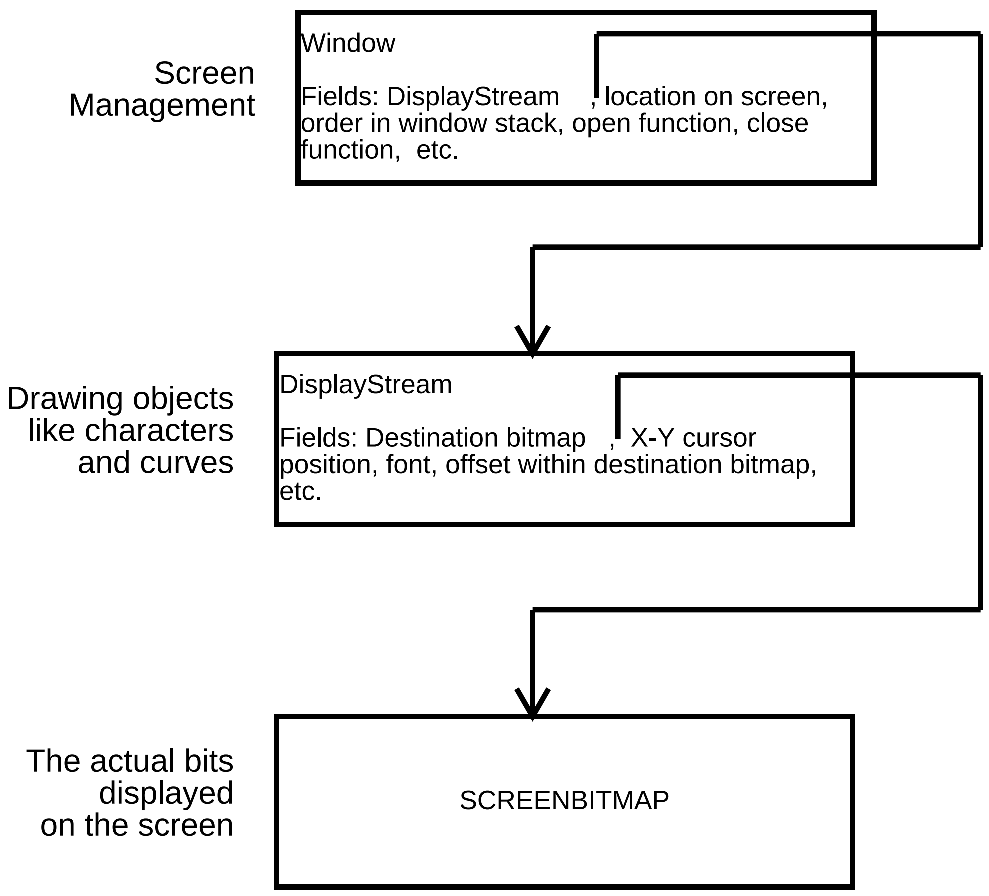
Windows are a way of managing what is being displayed on the most important bitmap of all, the Interlisp-D display screen.

A window is a datatype that represents the window object that can be displayed on the Interlisp screen.

Stored in the fields of the window datatype are all kinds of functions that specify what is to be done when various operations are carried on the window, e.g., when the window is opened, closed, shrunk, or moved.

Also stored in each window datatype is a display stream through which characters, lines, curves, etc. are drawn on the window's bitmap (i.e., the display stream's destination bitmap) which is always SCREENBITMAP.

In summary, the data structures underlying each window you see on the screen are as follows:


Screen Management
Window
Fields: DisplayStream , location on screen, order in window stack, open function, close function, etc.
Drawing objects like characters and curves
DisplayStream
Fields: Destination bitmap , X-Y cursor position, font, offset within destination bitmap, etc.
The actual bits displayed on the screen
SCREENBITMAP

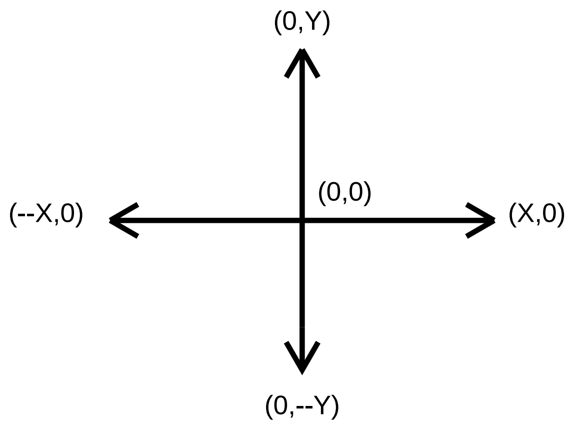
Coordinate Systems, Positions, & Regions

Coordinate Systems for Specifying Locations in Bitmaps, et al.

Each bitmap, display stream, and window has its own coordinate system used to specify locations within the object.

When dealing with the display, these coordinate systems are always measured in bits (or screen units or the area that it takes to display 1 bit on the screen or 1/72nd of an inch).

For all three objects, the coordinate system is a standard Cartesian system in the standard orientation.



Bitmaps have a finite size. Thus, for bitmaps the origin of the coordinate system is placed at the lower-left corner of the bitmap and only the upper-right quadrant (positive X and Y) is used to specify locations in the bitmap.

Display streams and windows are considered to look onto an infinite plane. Thus the origin is arbitrarily placed (see below) and the entire coordinate system is used.

The coordinate system for a window is the same as the coordinate system for its underlying display stream. The coordinate system for a display stream is mapped onto the coordinate system for its destination bitmap using X and Y translation parameters as discussed below.

Positions and Regions

Positions and regions are data structures that represent X-Y locations and rectangles, respectively, in an arbitrary coordinate system.

A ***POSITION*** is a record with two fields, XCOORD and YCCORD. Most functions that take an X-Y location as an argument require a POSITION record.

To create a POSITION record:

(create POSITION XCOORD _ *X* YCOORD _ *Y*)

A ***REGION*** is a record with four fields: LEFT, BOTTOM, WIDTH, and HEIGHT specifying the lower-left corner and extent of a rectangular region in some coordinate space.

To create a REGION record:

(CREATEREGION *Left Bottom Width Height*)

There are several functions available to manipulate positions and regions, including the following:

(INSIDEP *Region Position*) ž returns T if *Position* is inside *Region*.

Examples:

1_ (INSIDEP (CREATEREGION 100 100 10 10)(create POSITION XCOORD _ 150 YCOORD _ 100))

NIL

2_ (INSIDEP (CREATEREGION 100 100 100 10)(create POSITION XCOORD _ 150 YCOORD _ 100))

T

(INTERSECTREGIONS *Region1 Region2 ... RegionN*) ž returns the region that is the intersections of *Region1, Region2, ..., and RegionN*. NIL, if there is no intersection.

(UNIONREGIONS *Region1 Region2 ... RegionN*) ž returns the region that is the union of *Region1, Region2, ..., and RegionN*. The union is the smallest (rectangular) region that contains all of the given regions.

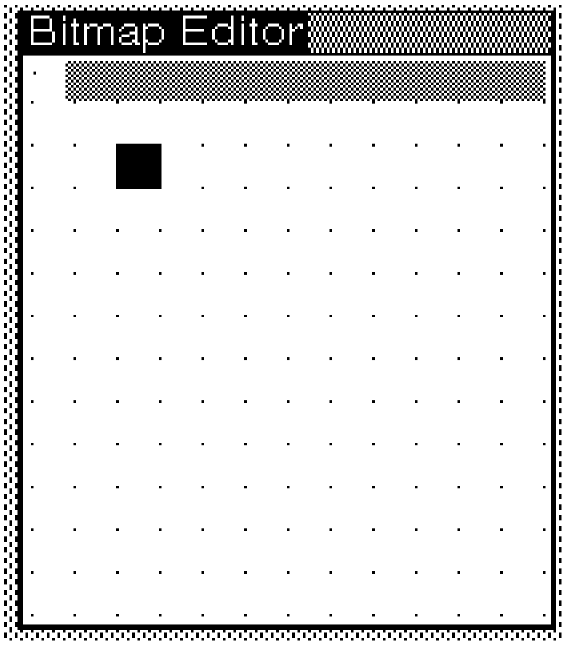
BITMAPs

Introduction

A bitmap is datatype that represents an N by M array of bits in memory.

The bits in a bitmap are identified using a positive integer coordinate system whose origin (0,0) is the lower-left corner of the bitmap.

For example, (10,2) represents the bit that is 10 to the left of and 2 up from the bit in the lower-left corner of the bitmap.



Creating Bitmaps

To create a bitmap use the following function:

(BITMAPCREATE *Width Height*) ž creates and returns a bitmap *Height* bits high and *Width* bits wide.

BITBLT

The major operation on bitmaps is the moving of bits from one bit map to another using the BITBLT function:

(BITBLT *SourceBitMap SourceLeft SourceBottom DestBitMap DestLeft DestBottom Width Height SourceType Operation Texture*) ž copies some bits in *SourceBitMap* and combines them with some bits in *DestBitMap*, resulting in a change to these bits in *DestBitMap*.

The bits copied from *SourceBitMap* are those in the region defined by *SourceLeft*, *SourceBottom*, *Width*, & *Height*.

The bits effected in the *DestBitMap* are those in the region defined by *DestLeft*, *DestBottom*, *Width*, & *Height*.

If either of these regions overflows the edges of its bitmap, then *Width* and/or *Height* are decreased until both regions fit into their bitmaps.

The way in which the bits are copied from the *SourceBitMap* is determined by *SourceType* and *Texture* as follows:

If *SourceType* is INPUT, then the bits are copied directly from the region in *SourceBitMap*.

If *SourceType* is INVERT, then the bits are copied from the region in *SourceBitMap*, but each bit is inverted (i.e., 1s become 0s and vice versa).

If *SourceType* is TEXTURE, then *SourceBitMap*, *SourceLeft*, and *SourceBottom* are ignored and the bits to be copied are taken from the bitmap specified by *Texture*.

If the *Texture* bitmap is smaller than *Width* by *Height*, then it is repeated as many times as necessary to make a rectangle of bits that is of size *Width* by *Height*.

Note: the global variables ***WHITESHADE***, ***BLACKSHADE*** and ***GRAYSHADE*** are small bitmaps for white, black, and gray, respectively.

SourceType defaults to INPUT.

The way in which the copied bits are combined with the bits already in *DestBitMap* is determined by *Operation* as follows:

If *Operation* is REPLACE, the bits in *DestBitMap* are replaced by the copied bits.

If *Operation* is PAINT, the bits in *DestBitMap* are logically ORed with the copied bits.

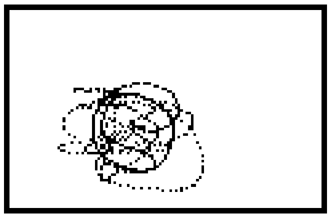
If *Operation* is INVERT, the bits in *DestBitMap* are logically XORed with the copied bits.

If *Operation* is ERASE, the bits in *DestBitMap* are logically ANDed with the inversion of the copied bits.

Operation defaults to REPLACE.

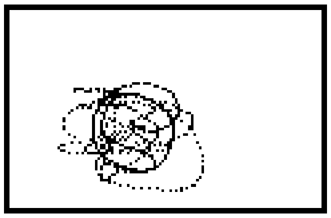
Examples:

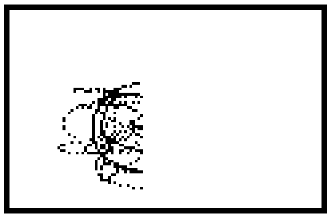
START

BitMap1:


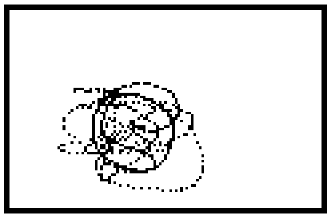
BitMap2:

(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 ’INPUT ’REPLACE)

BitMap1:


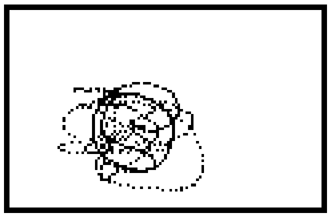
BitMap2:


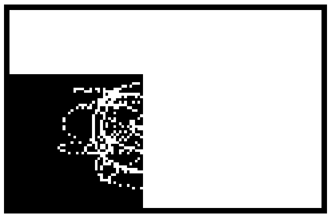
(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 ’INPUT ’ERASE)

BitMap1:


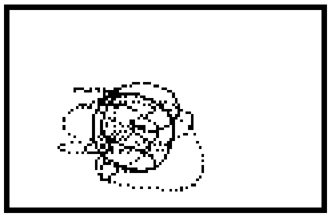
BitMap2:

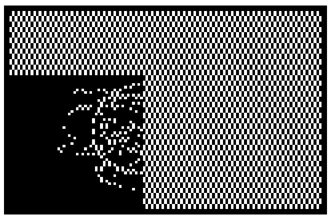
(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 ’INVERT ’PAINT)

BitMap1: 

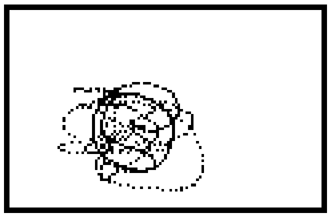
BitMap2: 

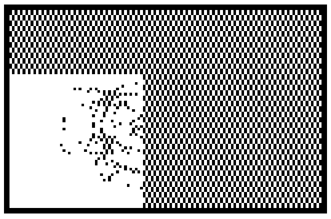
(BITBLT BitMap1 0 0 BitMap2 0 0 125 175 ’TEXTURE ’PAINT GRAYSHADE)

BitMap1: 

BitMap2: 

(BITBLT BitMap1 0 0 BitMap2 0 0 50 50 ’INVERT ’ERASE)

BitMap1: 

BitMap2: 

Miscellaneous Bitmap Functions

The following are other miscellaneous functions that operate on bitmaps:

(BITMAPBIT *BitMap X Y NewValue*) ž If *NewValue* is either 0 or 1, then sets the value of the (*X*,*Y*)th bit in *BitMap* to have value *NewValue* and returns its old value (either 0 or 1). If *NewValue* is NIL, just returns the value of the (*X*,*Y*)th bit in *BitMap*. If *NewValue* is anything else, then its an error.

(BITMAPCOPY *BitMap*) ž returns a new bitmap that is an exact copy of *BitMap*.

(BITMAPHEIGHT *BitMap*) ž returns the height in bits of *BitMap*.

(BITMAPWIDTH *BitMap*) ž returns the width in bits of *BitMap*.

The following functions return pointers to the "special" bitmaps:

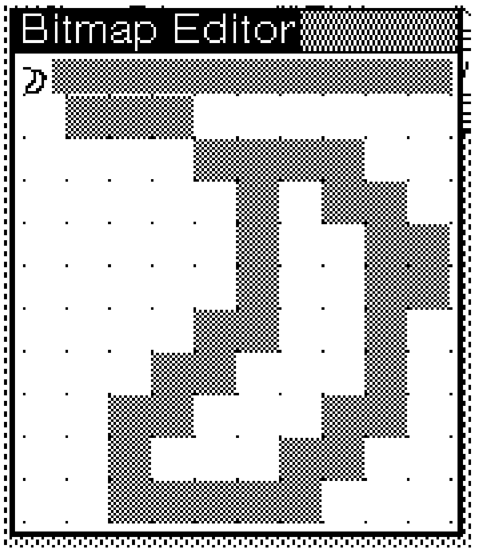
(SCREENBITMAP) ž returns the screen bitmap.

(CURSORBITMAP) ž returns the bitmap that is the cursor.

Hand Editing Bitmaps

(EDITBM *BitMap*) ž calls an editor to edit *BitMap*. If *BitMap* is NIL, will create a bitmap after asking for the height and width of the desired bitmap.

The bitmap editor runs in a window which you will be asked to place. The window is shown below.



In the bar just below the title bar flush to the left side of the window, the bitmap being edited in displayed at normal resolution.

To the right of this is a gray area ÿ clicking the MIDDLE mouse button in this gray area will bring up a menu of commands, including "OK" which will allow you to exit the editor.

The gridded area represents a portion of the bitmap blown up ÿ each square corresponds to one bit.

Clicking the RIGHT mouse button in a square makes the corresponding bit black.

Clicking the MIDDLE mouse button in a square makes the corresponding bit white**.**

To change the portion of the bitmap being displayed in this blown-up area, LEFT or MIDDLE click in the normal resolution bitmap above and choose the "Move" option (i.e., the only option) from the menu that appears.

Notes

Most of the above functions can take either a display stream or a window in place of a bitmap argument. In this case, the operation is carried out on the bitmap associated with the given display stream or window.

Thus, BITBLT et al. can be used to manipulate the contents of a window. (See below).

Display Streams

Displays streams are a special type of Stream (see LispCourse #38, page 3) designed to provide an interface to bitmaps (most importantly the display scrren bitmap).

Display streams allow you to access bitmaps in higher-level terms than just BITBLT and BITMAPBIT. In particular, display streams are designed to deal with objects like characters and geometric objects (lines, curves, circles, etc.).

Display Stream Properties (& creating a display stream)

Basically, a display stream has a destination bitmap and set of properties that specify how higher-level objects like characters and lines should be drawn on this bitmap.

The most important properties of a display stream are the following:

Destination ž the bitmap that this display stream refers to. Initializes to (SCREENBITMAP).

XOffset, YOffset ž the origin of the display stream’s coordinate system expressed in terms of the destination bitmap’s coordinate system. Initializes to 0 and 0, so that the display stream origin is at the lower-left corner of the destination bitmap.

ClippingRegion ž a region *in the display stream’s coordinate system* that limits where anything can be written or drawn to the display stream. If a display stream has a clipping region, then commands that request characters and/or lines to be drawn outside of this clipping region will simply be ignored.

XPosition, YPosition ž the X and Y coordinates of the "current" position of the display stream. The "current" position is an invidsible cursor that determines where things will be drawn unless otherwise specified. (Initially 0 and 0, i.e., at the display stream’s origin.

Texture ž the backround pattern used in the display stream (see Texture description under BITBLT above). Initializes to WHITESHADE.

Font ž a font descriptor that specifies the font to be used for printing characters on this display stream. Initializes to font descriptor for Gacha 10. (See LispCourse #21 for a discussion of fonts.)

Operation ž the BITBLT operation (see BITBLT above) used by default when printing or drawing on the destination bitmap. Must be one of REPLACE, PAINT, INVERT, or ERASE as above. Initializes to REPLACE.

For other properties of a display stream, see section 19.9.1 of the IRM..

To create a display stream, use the DSPCREATE function:

(DSPCREATE *Destination*) ž returns a display stream object using Destination as its destination bitmap. If Destination is NIL, then (SCREENBITMAP) is used. All of th properties of the display stream are initialized as described above.

The properties of a display stream can be manipulated using a set of functions that work as follows:

Each property is is manipulated by a selector/mutator whose name is "DSP" followed by the property name in all caps.

For example: the Font property is accessed using the function **DSPFONT**.

Each of these functions takes two arguments: the new value of the property and the display stream.

If the new value is NIL, then the function just returns the current value of the property.

If the new value is a value, then the function returns the old value and sets the property to the new value.

Example:

(DSPFONT NIL DS) returns the Font property of display stream DS.

(DSPFONT (FONTCREATE ’TIMESROMAN 16) DS) changes the Font property of DS to be TimesRoman 16 and returns the old font of DS.

(DSPTEXTURE GRAYSHADE DS) changes the background texture of DS to be gray and returns the old background texture.

Moving the Current Position in the Display Stream

When printing or drawing to a display stream, the default is to print/draw at the current position, i.e., at (XPosition, YPosition) in the display stream.

DSPXPOSITION and DSPYPOSITION can be used to independently change the X and Y coordinates of the current position.

The following functions can also be used to change the current position:

(MOVETO *X Y DisplayStream*) ž moves *DisplayStream*’s current position to point (*X*,*Y*).’

(RELMOVETO *DeltaX DeltaY DisplayStream*) ž moves *DisplayStream*’s current position to a point *DeltaX* units to the right and *DeltaY* units up from its previous position.

Printing and Drawing on Display Streams

To print characters on a display stream, use the standard printing routines discussed in LispCourse #38 (starting on page 12).

For example: (PRIN1 "ABC" DS) will print ABC on display stream DS. It will do the printing starting at the DS’s current point and then will move the current point to the end of the last character printed.

The font used for printing the characters is determined by DS’s font property.

To draw straight lines on a display stream use the following functions:

(DRAWTO *X Y Width Operation DisplayStream*) ž draws a line on *DisplayStream* from the current point to location (*X*,*Y*). The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*’s operation property). At the end, *DisplayStream*’s current point is set to (*X*,*Y*).

(RELDRAWTO *DeltaX DeltaY Width Operation DisplayStream*) ž draws a line on *DisplayStream* from the current point to the location *DeltaX* to the right and *DeltaY* up. The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*’s operation property). At the end, *DisplayStream*’s current point is set to the end of the line.

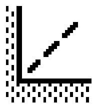
(DRAWBETWEEN *Position1 Position2 Width Operation DisplayStream*) ž draws a line on *DisplayStream* from *Position1* to *Position2*. The line is *Width* bits wide and is drawn using the BITBLT operation specified by *Operation* (defaults to the *DisplayStream*’s operation property). At the end, *DisplayStream*’s current point is set to *Position2*.

To draw curved lines on a display stream use the following functions.

These functions all take a Brush argument. The Brush argument is a two item list containing the ***shape*** and the ***width*** of the "brush" that will be used to draw the curve. *Shape* is one of: ROUND, SQUARE, VERTICAL, DIAGONAL. *Width* is the thickness of the line to vbe drawn in bits.

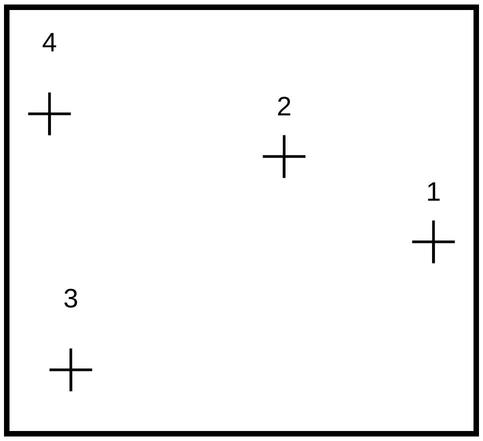
These functions also take a Dashing argument that determines how to dash the line. If Dashing is NIL, no dashing will be done. Otherwise, Dashing is a list containing an even number of positive integers. The first integer specifies how long (in bits) the brush should be "on", the second integer then specifies how long the brush should be "off", the third integer specifies how long the brush should be "on" again, etc.

Example: A Dashing of (5 2) specifies that the line should have 5 bits on then 2 bits off, then 5 bits on, etc.

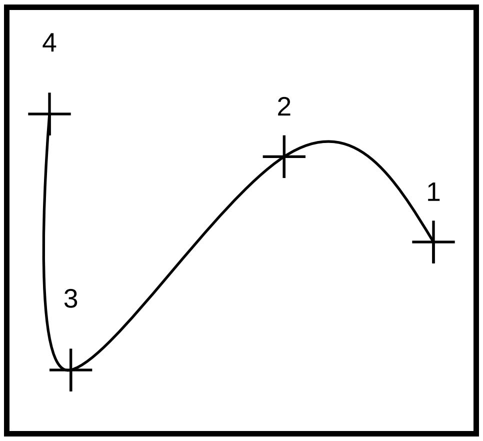


(DRAWCURVE *Knots ClosedFlg Brush Dashing DisplayStream*) ž
Knots is an ordered list of POSITIONs. DRAWCURVE draws a spline curve on *DisplayStream* that is fit to these POSITIONs. If ClosedFlg in NIL, the spline will be an open curve; otherwise it will be a closed curve. Brush and Dashing are as described above. The current position is left unchanged.

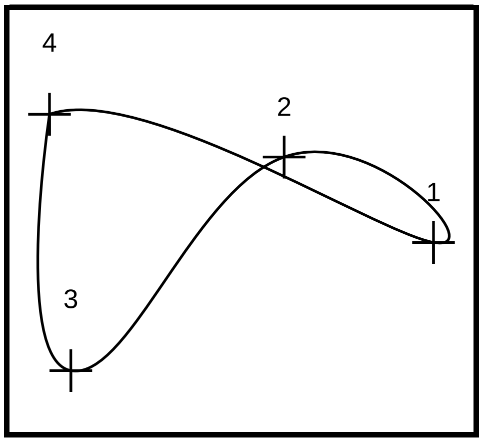
The knots:



An open curve using drawn unsing these knots:



A closed curve using drawn unsing these knots:



(DRAWCIRCLE *X Y Radius Brush Dashing DisplayStream*) ž Draws a circle on *DisplayStream* centered at (*X*,*Y*) and having radius *Radius.* The current position is left at (*X*,*Y*).

(DRAWELLISPSE *X Y MinorRadius MajorRadius Orientation Brush Dashing DisplayStream*) ž Draws an ellipse on *DisplayStream* centered at (*X*,*Y*) and having a minor radius *MinorRadius* and a major radius *MajorRadius.* The orientation of the *MajorRadius* is determined by *Orientation,* which is in degrees from upright in the counterclockwise direction. The current position is left at (*X*,*Y*).

Windows & the Window Package

The window package provides two basic services to Interlisp-D programs:

1) It manages the "space" on the Interlisp-D display screen, allowing multiple programs using multiple display streams to all access the screen simultaneously without interfering with each other.

2) It provides an interface that dispatches the user’s mouse "actions" to the Interlisp-D programs that these actions were intended to effect.

We will discuss only the first function in this section. Mouse action dispatching will be covered in the section on the Mouse below.

Space Management: The Window Stack

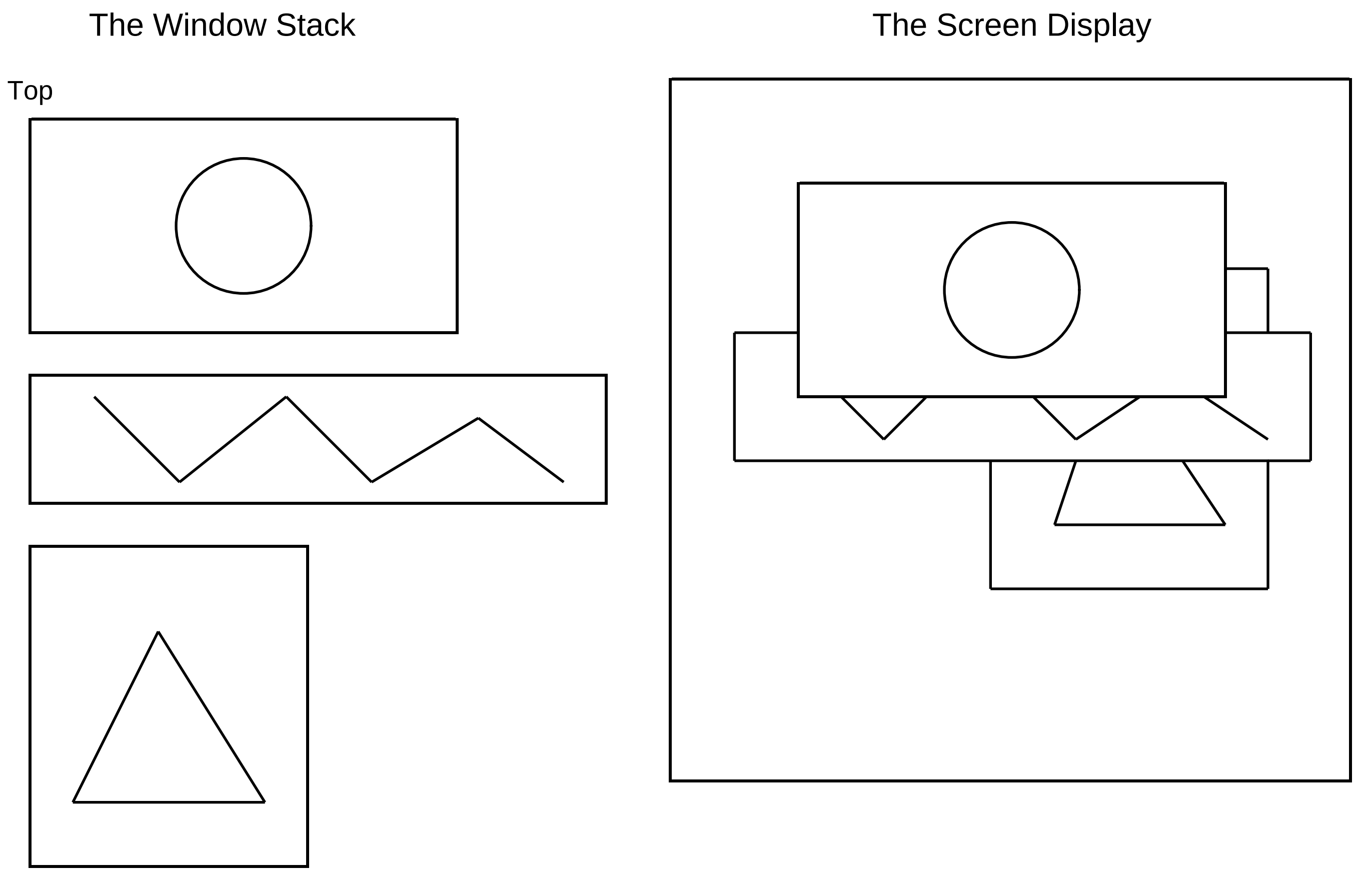
Space on the Interlisp-D screen is managed using an occlusion stack of open windows.

Each open window represents a rectangular region of the screen in which a display stream (or part of a display stream) is to be displayed.

All of the open windows are placed on a stack (i.e., an ordered list), ordered by "depth".

If the display regions of two or more windows intersect, then the actual screen display in the intersection region will reflect the contents of the window nearest to the top of the stack (i.e., earlier in the ordered list).

The windows that are lower (deeper, later) in the stack will have the part of them corresponding to the intersection region "occluded" by the windows higher in the stack.



The window at the top of the stack is always visible in its entirety on the screen.

Windows later in the stack may also be entirely visible (if they do not intersect with any other open windows), but this cannot be guarenteed.

Any operation on a window brings it to the top of the stack. Thus all operations are carried out on windows that are entirely visible on the screen.

The function call **(OPENWINDOWS)** returns the window stack as an ordered list.

Windows

A window is just a data structure that contains all of the information necessary for the window package to display the window on the screen.

Open and Closed Windows

Windows can be ***open*** or ***closed***.

A *closed* window is just a data structure and is not part of the window stack. It therefore cannot be displayed on the screen.

An *open* window is on the window stack and is therefore displayed on the screen providing that it is not completely occluded by other open windows on the stack.

Whenever you operate on a window (e.g., draw in the window), the window is opened (and brought to the top of the stack).

(OPENW *Window*) opens the closed window *Window*, placing it at the top of the window stack.

(CLOSEW *Window*) closes the open window *Window*.

(OPENWP *Window*) returns *Window* if it is an open window; NIL otherwise.

Creating Windows

To create a window use:

(CREATEW *Region Title Border NoOpenFlg*) ž creates and returns a new window.

The created window will be displayed (if opened) in screen region *Region*. If *Region* is NIL, the the user will be asked to specify a region on the screen.

If *Title* is non-NIL, a title bar will be created at the top of the window and *Title* will be printed left-flush in this title bar.

If *Border* is a number it is the number of bits to use as a border around the edge of the window; otherwise the window will have a border width of 4.

If *NoOpenFlg* is NIL, the created window will be opened and placed at the top of the window stack. Otherwise, the created window will not be opened.

(WINDOWP *Window*) ž returns *Window* if it is a window, NIL otherwise.

Windows and Display Streams

When a window is created, a corresponding display stream is created and stored in the window data structure.

The function: **(WINDOWPROP *Window* ’DSP)** will retrieve the display stream associated with *Window*. (See explanation of WINDOWPROP below.)

This destination of a window’s display stream is always (SCREENBITMAP). Thus, a program should never alter the Destination property of a window’s display stream.

The window package automatically takes care of setting the XOffset, YOffset and ClippingRegion properties for the window’s display stream as the window is scrolled or moved about the screen.

A program should never alter the XOffset, YOffset, and ClippingRegion properties of a window’s display stream.

Otherwise, all properties and operations applicable to display streams are also applicable to windows.

For example, to change the font used to print in a window, you use DSPFONT. It is not necessary to specify the window’s display stream directly since all the DSPxxx functions know how to coerce a window into its corresponding display stream.

(DSPFONT BigFont Window) has the same effect as (*DSPFONT BigFont (WINDOWPROP Window ’DSP))*

Printing and Drawing in Windows

Printing and drawing in a windows is identical to printing and drawing in display streams ž use the print functions and the *DSPxxx* functions described above.

Just use the window as the argument in place of a display stream.

Doing Things to Windows

The following functions carry out various operations on windows.

(MOVEW *Window Position*) ž moves *Window* so that its lower-left corner is located at POSITION *Position* on the display screen. If *Position* is NIL, then the user is asked for to specify a position.

If *Window* is closed, it will not be opened unless *Position* is NIL.

(SHAPEW Window NewRegion) ž rehapes and moves *Window* so that it is displayed in screen region *NewRegion*. If *NewRegion* is NIL, the user is asked to specify a new region. Opens *Window* if it is closed.

(TOTOPW *Window*) ž places *Window* at the top of the window stack and adjusts the display accordingly. *Window* is opened if it was closed.

(BURYW *Window*) ž places *Window* at the bottom of the window stack and adjusts the display accordingly.

(CLEARW *Window*) ž clears the window, filling it with the Texture of its underlying display stream. Also sets the current position to the upper-left corner of the window.

(REDISPLAYW *Window*) ž causes *Window* to be redisplayed. Used if, for example, the window's display gets messed up. For this to work correctly the window must have a REPAINTFN property (see below).

(SHRINKW *Window*) ž closes *Window* and replaces it on the screen with a smaller icon window. The icon window is actually another window that is associated with the window being shrunk.

Unless the window's properties specify otherwise (see below), the icon window is just a black bar containing *Window*'s title or the date and time if *Window* has no title.

You cannot shrink an icon window.

SHRINKW is a no-op if *Window* is closed.

(EXPANDW *Window*) ž expands or "unshrinks" *Window* if it was previously "shrunk". If *Window* is an icon window, expands the window that the icon window represents. If *Window* is neither shrunk nor an icon window, EXPANDW is a no-op.

Tailoring Windows to Special Applications: Window Properties

Every window has a large set of properties that determine how it looks and, more importantly, how it behaves in various circumstances.

You needn't set any of these properties, they will all default to something reasonable if you just want an ordinary window.

But if you want a fancy window, you need to set one or more of these properties to get the looks or behavior you are interested in.

Most of the properties are intended to have functions or lists of functions as values. These functions are called (with the window as an argument) whenever some operation is done on the window (e.g., the window is closed) or some event occurs in the window's environment (e.g., the user clicks a mouse button in the window).

In addition to these functional properties, each window has a few properties that are not functions, e.g., the window's title, and a few that are read-only (i.e., that the window package sets and that the user can query but not change).

Looks Properties

TITLE ž a title to be printed left-flush in the title bar at the top of the window. If TITLE is NIL, the window will have no title bar.

If TITLE is changed from NIL to a value, then a title bar is created resulting in a slight enlargement of the window. Defaults to NIL.

BORDER ž an integer indicating the width of the border around the edge of the window. The border will have (at most) 2 bits of white around the inside with the remaining border width being black; subject to the constraint that the white bits never make up more than half the border width. Defaults to 4.

WINDOWTITLESHADE ž a texture (see texture under BITBLT above) that is used as the background in the title bar to the right of the end of the title. Where the title is printed, the background is black. Defaults to value of global variable WINDOWTITLESHADE, which defaults to BLACKSHADE.

Read-only Properties

DSP ž the window's display stream.

HEIGHT, WIDTH ž the height and width in bits of the interior of the window (i.e., not including the border and title bar). The interior part of the window is the user accessable portion of the window ÿ you cannot draw or print directly on the title bar or border.

REGION ž a REGION that describes (in the SCREENBITMAP coordinate system) the region occupied by the entire window (title bar and all) on the screen.

Processes Property

PROCESS ž the process that is associated with this window. This is the process that will become the TTY process when GIVE,TTY.PROCESS is called using this window as an argument.

By default (see WINDOWENTRYFN) this is the process that becomes the TTY process when you button down inside the window.

Functions to Be Invoked by Operations on the Window

CLOSEFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just before the window is closed. If any function is the atom DON'T or returns the atom DON'T, then the window will not be closed.

Warning: You cannot call CLOSEW inside of a CLOSEFN function otherwise you will get infinite recursion.

OPENFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just after the window is opened. If the atom DON'T appears anywhere on the OPENFN list, then the window will not be opened and the OPENFNs will not be called.

TOTOPFN ž a single function that will be called (with the window as the only argument) whenever the window is brought to the top of the window stack.

MOVEFN ž a single function or a list of functions that will be called (with the window and the new position of the lower-left corner of the window as arguments), in order, just **before** the window is moved. If any function is the atom DON'T or returns the atom DON'T, then the window will not be moved. If any function returns a POSITION record, then the window will be moved to that position.

AFTERMOVEFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just **after** the window is moved.

REPAINTFN ž a single function or a list of functions that will be called (with the window and the region of the window to be repainted as arguments), in order, whenever the window is redisplayed (i.e., during scrolling or when REDISPLAYW is called). The function should redraw the contents of the specified region in window.

Warning: You cannot call CLEARW inside of a REPAINTFN. Use DSPFILL instead.

Functions to Be Invoked by Mouse Events in the Window

BUTTONEVENTFN ž a single function that will be called (with the window as the only argument) whenever there is a change in state of the mouse buttons (a mouse button goes up or down) while the cursor is inside the window and the window is associated with the TTY process.

While the BUTTONEVENTFN is being processed, further mouse events do not reinvoke the BUTTONEVENTFN.

As a general convention a BUTTONEVENTFN is called when a mouse button goes down but does not do its work until the mouse button goes up. This convention is accomplished by writing the "correct" kind of BUTTONEVENTFNS - i.e., functions that wait until the mouse button is up before continuing their work.

RIGHTBUTTONFN ž a single function that will be called (with the window as the only argument) instead of the BUTTONEVENTFN whenever only the RIGHT mouse button goes down in the window.

If RIGHT mouse clicks are to be treated the same as other mouse clicks, then just make the BUTTONEVENTFN and the RIGHTBUTTONFN be the same function.

Defaults to DOWINDOWCOM, which brings up a menu of the standard window operations.

WINDOWENTRYFN ž a single function that will be called (with the window as the only argument) whenever a button goes down in the window and the process associated with the window is not the TTY process.

Default is to call GIVE.TTY.PROCESS on the window and then call the window’s BUTTONEVENTFN.

CURSORINFN, CURSOROUTFN, CURSORMOVEDFN ž a single function that will be called (with the window as the only argument) whenever the cursor moves into (out of, about inside of) the window.

Properties that Support Icons and Shrinking

SHRINKFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just before the window is shrunk. If any function is the atom DON’T or returns the atom DON’T or if any of the CLOSEFNs is DON’t or returns DON’T, then the window will not be shrunk.

ICONFN ž a single function that will be called (with the window and the previous icon, if any), just before the window is shrunk. The ICONFN should return a bitmap or a window that will be used as the basis for making the icon window.

EXPANDFN ž a single function or a list of functions that will be called (with the window as the only argument), in order, just after the window is expanded from the shrunk state. If any function is the atom DON’T, then the window will not be shrunk and the rest of the EXPANDFNs will not be called.

Manipulating a Window’s Properties: WINDOWPROP, et al.

The function WINDOWPROP is a selector/mutator that can be used to manipulate properties of a window:

(WINDOWPROP *Window Property NewValue*) ž sets the *Property* property of *Window* to be *NewValue*. Returns the old value of the specified property.

WINDOWPROP is a LAMBDA/No-spread function. If *NewValue* is omitted from the function call, then

WINDOWPROP will simply return the old value of the specified property. (Note: Omitting NewValue is not the same as specifying NIL as the NewValue. The former will not change the specified property, the later will set the property's value to NIL.)

Important note: WINDOWPROP can be used to add any arbitrarily named property to a window ÿ the *Property* argument need not be one of the properties supported by the window package (and described above). Thus WINDOWPROP can be used to cache any sort of information on the window that may be useful to the program. See the scrolling window example below.

The functions WINDOWADDPROP and WINDOWDELPROP can be used to add and remove items for properties whose values are lists (e.g., lists of functions as for the CLOSEFN property):

(WINDOWADDPROP *Window Property ValueToBeAdded*) ž adds *ValueToBeAdded* to the list that is the value of the *Property* property of *Window*. If *ValueToBeAdded* is already on that list it is not added again. If the current value of the specified property is not already a list, it is made into a list before *ValueToBeAdded* is added. *ValueToBeAdded* is always placed at the end of the list. Returns the old value of the specified property.

(WINDOWDELPROP *Window Property ValueToBeRemoved*) ž removes *ValueToBeRemoved* from the list that is the value of the *Property* property of *Window*. If *ValueToBeRemoved* was actually on that list, WINDOWDELPROP returns the old value of the specified property. Otherwise, WINDOWDELPROP returns NIL.

Making Scrollable Windows

The window package provides some support for making scrolling windows.

In particular, the window package provides and manages the scroll bars that pop up whenever the mouse rolls through the left edge of the window.

The window package also supports a default scheme for doing the actual scrolling of the window contents ÿ although the default scheme requires some amount of progamming setup.

The SCROLLFN Property

Every window has a SCROLLFN property.

If the SCROLLFN property is NIL (the default), then the window will not be scrollable.

If the SCROLLFN has a non-NIL value, then the window will be scrollable as follows:

Whenever the cursor moves from inside the window to outside the window across the window’s right border, the window package (in particular, the scroll handler) will bring up a scroll bar.

If the user presses a mouse button while the scroll bar is up, then the window package will call the function that is the value of the SCROLLFN property. If the user continues to hold down the mouse button while in the scroll bar, the SCROLLFN will be called repeatedly every few milliseconds until the button is released.

The SCROLLFN will be called with the following 4 arguments: 1) the window, 2) the distance to scroll horizontally, 3) the distance to scroll vertically, and 4) a flag that indicates whether the button is still being held.

The distances to scroll depend on which mouse button was pressed as follows:

LEFT ž the distance in screen units (bits) from the cursor to the top (left if horizontal scroll) of the window.

RIGHT ž negative of the distance in screen units (bits) from the cursor to the top (left if horizontal scroll) of the window.

MIDDLE ž a FLOATP that describes the ratio of the distance between the cursor and the top (or left) edge of the window to the length of the entire scroll bar. (In other words, proportion of the way down (or across) the scroll bar that the cursor is.)

The SCROLLFN should take care of changing the contents of the window as appropriate for the distances specified by the user’s mouse press.

Every window has a SCROLLFN property.

SCROLLBYREPAINTFN

The window package provides a default SCROLLFN, called **SCROLLBYREPAINTFN**, that can be attached to the SCROLLFN property of any window that has a non-NIL REPAINTFN property.

The idea behind SCROLLBYREPAINTFN is as follows:

A window is seen as showing at any given time a small part of a much larger display. The coordinate system of this display is the coordinate system of the window’s display stream.

Each window has a property called EXTENT that describes the region in the display stream’s coordinate system that the larger display occupies. Setting the EXTENT property of the window is the responsibilty of the program that uses SCROLLBYREPAINTFN.

The ClippingRegion property of the window’s display stream determines what region of the display stream (and hence what region of the EXTENT or larger display) is being currently shown in the window.

When a window scrolls, the ClippingRegion of the display stream changes ž another part of the larger display is now being shown in the window.

When a window’s REPAINTFN gets called it is passed a window and a region in the window’s display stream. The REPAINTFN is responsible for drawing in the window that region of the display stream.

Thus, SCROLLBYREPAINTFN works as follows:

SCROLLBYREPAINTFN calls the window’s REPAINTFN to draw in the window the contents of the new ClippingRegion whenever a scroll is requested (i.e., whenever the window’s SCROLLFN is called).

SCROLLBYREPAINTFN takes care of translating the scroll distances (see the SCROLLFN description above) into a new ClippingRegion and then calling the window’s REPAINTFN with the new ClippingRegion as an argument.

The programmer need only write a REPAINTFN that can display an arbitrary region of the larger display underlying the window.   The programmer must also specify the EXTENT of the underlying display since SCROLLBYREPAINTFN needs to this to caluculate the new ClippingRegion after each scroll.

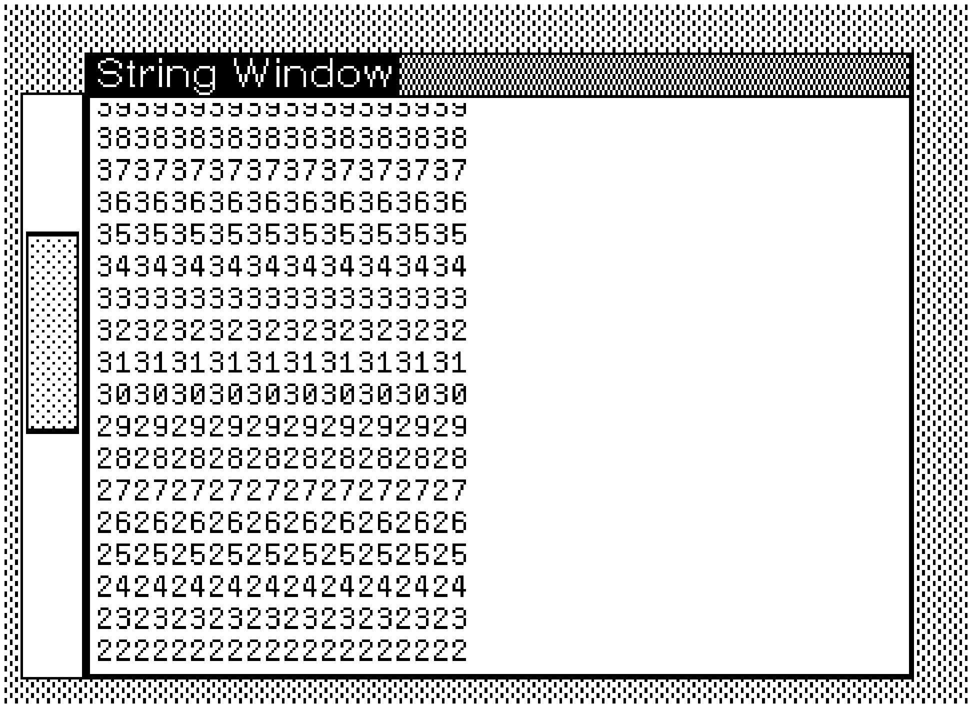
Example

The goal is to develop a scrollable window that displays a list of strings, one string per line.

```
(MakeStringWindow
  (LAMBDA (StringList)                        (* fgh: "28-Jun-85 12:11")

          (* * Create a scrollable window to display the strings in
               StringList, one string per line.)


    (LET ((SW (CREATEW NIL "String Window")))

          (* * SCROLLFN will be the default -- SCROLLBYREPAINTFN)


      (WINDOWPROP SW (QUOTE SCROLLFN)
                  (FUNCTION SCROLLBYREPAINTFN))

          (* * REPAINTFN will be StringWindowRepaintFn)


      (WINDOWPROP SW (QUOTE REPAINTFN)
                  (FUNCTION StringWindowRepaintFn))

          (* * EXTENT of underlying display for scrolling is a region whose
           lower-left corner is at 0,0 and whose width is the width of the
           window and whose height is the number of strings times the height
           of each line as determined by the height of the window's font.)


      (WINDOWPROP SW (QUOTE EXTENT)
                  (CREATEREGION 0 0 (fetch (REGION WIDTH)
                                      of (WINDOWPROP SW (QUOTE REGION)))
                                (TIMES (LENGTH StringList)
                                       (FONTPROP (DSPFONT NIL SW)
                                                 (QUOTE HEIGHT))))))

          (* * Cache the string list on the window so we can use it
           in the REPAINTFN)


      (WINDOWPROP SW (QUOTE StringList)
                  StringList)

          (* * Display the first window full -- will be region out of EXTENT
              starting from 0,0 with height and width determined by the
              height and width of the window.)

      (REDISPLAYW SW)

          (* * Return the window)

      SW)))


(StringWindowRepaintFn
  (LAMBDA (Window Region)                     (* fgh: "28-Jun-85 12:21")
          (* * Repaint a string window. StringList to print is cached on
          the StringList property of the window. Region argument is used to
          determine which strings to print in the window. Starting from 0,0
          for each LineHeight increment print 1 string. Thus, string 1 goes
          at 0,0. String 2 goes at 0, LineHeight. String 3 goes at 0,
          (2*LineHeight) etc. LineHeight is just the height of the font
          used in the window.)


    (LET ((StringList (WINDOWPROP Window (QUOTE StringList)))
       (LineHeight (FONTPROP (DSPFONT NIL Window)
                             (QUOTE HEIGHT)))
       (RegionHeight (fetch (REGION HEIGHT) of Region))
       (RegionBottom (fetch (REGION BOTTOM) of Region))
       FirstLine LastLine)

          (* * Determine the first {i.e., top most} line to print)


      (SETQ FirstLine (ADD1 (FIX (QUOTIENT (PLUS RegionBottom RegionHeight)
                                           LineHeight))))

          (* * Determine the last {i.e., bottom most} line to print)


      (SETQ LastLine (ADD1 (FIX (QUOTIENT RegionBottom LineHeight))))

          (* * Move to the left edge and the bottom of the first {top most}
            line in the window. The Times/Quotient constrution is to
            position on an exact line base, i.e., on Y poisition that is
            an exact multiple of LineHeight.)


      (MOVETO (fetch (REGION LEFT) of Region)
              (TIMES (QUOTIENT (PLUS RegionBottom RegionHeight)
                               LineHeight)
                     LineHeight)
              Window)

          (* * For the top most to the bottom most line in the Region, print
           the corresponding string then move down a line. If the calculated
           string number is not positive, then we have scrolled below the
           end of the EXTENT. In this case don't print any string.)


      (for Line from FirstLine to LastLine by -1
         do (AND (GREATERP Line 0)
                 (PRIN1 (CAR (NTH StringList Line))
                        Window))
            (DSPXPOSITION 0 Window)
            (RELMOVETO 0 (MINUS LineHeight)
                       Window)))))
```

32_(MakeStringWindow
      (FOR X FROM 1 to 50 COLLECT (CONCAT X X X X X X X X X X)))

{WINDOW}#45,5643



```
39393939393939393939
38383838383838383838
37373737373737373737
36363636363636363636
35353535353535353535
34343434343434343434
33333333333333333333
32323232323232323232
31313131313131313131
30303030303030303030
29292929292929292929
28282828282828282828
27272727272727272727
26262626262626262626
25252525252525252525
24242424242424242424
23232323232323232323
22222222222222222222
```

Menus

The Menu package is very closely related to the Window package.

Basically, a menu is a window (or part of a window) in which pressing a mouse button in certain specified regions (i.e., menu items) causes certain specified events to happen.

In Interlisp-D, a menu is implemented as window with a special BUTTONEVENFN that uses a menu data structure to determine what action to take depending on what menu item the cursor is over when the button was pressed.

Bringing a menu up on the display requires two steps: 1) creating the menu data structure, and 2) displaying the menu in the window.

Creating the menu data structure

A ***MENU*** is a system datatype. Cached on the data type is all the information the menu package needs in order to display the menu in a window and in order to carry out actions when the mouse is clicked within the menu.

The MENU datatype has the following fields:

Fields that describe the menu’s behavior

ITEMS ž a list of the items to be included in the menu. If an item is an NLISTP, then it is displayed in the menu. If an item is a list, then the CAR of that list is displayed in the menu.

The display consists of the print name of the item or its CAR unless it is a bitmap, in which case the bitmap is displayed.

Under most circumstances, each item will be a list of three elements: the item *label*, the item *action*, and the item *message*. This list is interpreted by the default WHENSELECTEDFN as described below.

If a WHENSELECTEDFN other than the default is used, then each item can consist of whatever that WHENSELECTEDFN requires.

WHENSELECTEDFN ž a function that gets called whenever an item is selected in the menu by a mouse click. The function will get called with three arguments: 1) the item slected, 2) the menu, and 3) the mouse key used (LEFT, MIDDLE, or RIGHT).

The WHENSELECTEDFN defaults to the function DEFAULTWHENSELECTEDFN which operates as follows:

If the CADR of the item is non-NIL, then that CADR is evaluated and returned as the value of the function.

Otherwise, the item itself is returned.

WHENHELDFN ž a function that gets called whenever the user holds the mouse button down inside an item for more than MENUHELWAIT milliseconds (defaults to 1200). The function will get called with three arguments: 1) the item slected, 2) the menu, and 3) the mouse key used (LEFT, MIDDLE, or RIGHT).

The WHENHELDFN defaults to the function DEFAULTMENUHELDFN which operates as follows:

If the CADDR of the item is non-NIL, then that CADDR is printed in the prompt window.

Otherwise, *This item will be slected when the button is released* will be printed in the prompt window.

WHENUNHELDFN ž a function that gets called whenever the WHENHELDFN has been called and the user lets up on the mouse button or moves the cursor out of the item. The function will get called with three arguments: 1) the item slected, 2) the menu, and 3) the mouse key used (LEFT, MIDDLE, or RIGHT).

The default WHENUNHELDFN is CLRPROMPT, which clears the prompt window.

Fields that describe the menu’s looks

TITLE ž the title to appear in the title bar of the menu. If NIL, then the menu will have no title bar.

MENUFONT ž the font in which the items will be printed. Defaults to Helvetica 10.

MENUROWS *or* MENUCOLUMNS ž the number of rows or columns the menu is to have. If both of these are NIL, the menu will have one column.

ITEMHEIGHT, **ITEMWIDTH** ž the height (width) of each item in the menu. If ITEMHEIGHT is not specified, then the height will be the maximum height of any item in the menu (as determined by the sizes of any bitmaps and the height of the MENUFONT). If ITEMWIDTH is not specified, the maximum width of any item will be used.

CENTERFLG ž if non-NIL, the menu items are printed centered in their areas. Otherwise, they are left-justified.

MENUBORDERSIZE ž the width in bits of the black border around each item. Defaults to 0.

MENUOUTLINESIZE ž the width in bits of the black border that outlines the entire menu. Defaults to the maximum of 1 and MENUBORDERSIZE.

Fields that describe the menu’s positioning

MENUPOSITION ž the default position of the menu in screen coordinates (for pop-up menus) or window coordinates (for permanent menus) [See below for op-up versus permamnent menus].

If NIL, then the memu is paced at the cursor.

The point of the menu to be placed at MENUPOSITION is determined by MENUOFFSET.

MENUOFFSET ž the point inside the menu that will be placed over MENUPOSITION. Defaults to 0,0, i.e., the lower-left corner.

To create a menu data structure use the standard CREATE statement from the Record Package:

Example:

```
(SETQ MenuX
    (create MENU ITEMS _ '(Yes No) CENTERFLG _ T
    TITLE _ "Yes or No??"))
```

To change a menu data structure use the standard replace statement from the Record Package:

Example:

```
(replace (MENU TITLE) of MenuX with "Well??")
```

Bringing up a menu on the screen: Pop-up and Fixed Menus

There are two way to use a menu: ***pop-up*** and ***fixed***.

Pop-up menus are used in a program to get some information from a user. A program using a pop-up menu brings the menu up on the screen and then waits for the user to select an item. When the item is selected, the menu is removed from the screen and the menu's WHENSELECTEDFN is called. The WHENSELECTEDFN will carry out some action, return a value to the program, or both.

Fixed menus remain on the screen "permanently. Whenever the user clicks a mouse button in one of the menu's items, the menu's WHENSELECTEDFN is called to carry out some action. Since fixed menus are not part of the ongoing processing of a program, the value returned by the WHENSELECTEDFN is ignored.

The MENU data structure is identical for pop-up and fixed menus. The difference is in the function used to bring the menu up on the screen.

The function for displaying a pop-up menus is:

(MENU *Menu Position*) ž displays *Menu* at *Position* (in the screen coordinate system) and then waits for the user to press and release a mouse button.

Pressing a mouse button has the following effects:

If the cursor is an item in *Menu*, that item is inverted on the screen.

If the user holds the mouse button down inside the item for MENUHELDWAIT milliseconds, then *Menu*'s WHENHELDFN is called.

If the user lets up on all mouse buttons while the cursor is still in the item, then *Menu*'s WHENSELECTEDFN is called.

If the user lets up on all mouse buttons while the cursor is outside of *Menu*, then no action is taken.

MENU returns the value returned by the call to the WHENSELECTEDFN, unless the user releases the mouse button while the cursor is outside of *Menu* in which case MENU returns NIL.

If the *Position* argument is NIL, then the MENUPOSITION field of *Menu* is used. If the MENUPOSITION field is also NIL, then the current cursor position is used.

Example:

```
(SELECTQ (MENU (create MENU ITEMS _'(Yes No)))
    (Yes (PRINT "The answer is Yup."))
    (No (PRINT "The answer is Nope."))
    (PRINT "No answer given."))
```

The function for displaying a fixed menus is:

(ADDMENU *Menu Window Position*) ž displays *Menu* in *Window* at *Position* (in the window's coordinate system) and returns immediately. The CURSORINFN and BUTTONEVENTFN of *Window* are replaced by the function MENUBUTTONFN so that when the user presses a mouse button inside an item in *Menu* (in *Window*) the following events take place:

The item is inverted.

If the user holds the mouse button down inside the item for MENUHELDWAIT milliseconds, then *Menu*'s WHENHELDFN is called.

If the user lets up on all mouse buttons while the cursor is still in the item, then *Menu*'s WHENSELECTEDFN is called.

If the user lets up on all mouse buttons while the cursor is outside of *Menu*, then no action is taken.

If no *Position* argument is given, then the MENUPOSITION field of *Menu* is used. If the MENUPOSITION field is also NIL, then the current cursor position is used.

If no *Window* argument is given, then a window is created just big enough to hold the menu and placed at *Position* (in the screen coordinate system).

ADDMENU always returns the window the menu was placed in.

Note: you can add more than one menu to a single window. But you cannot add a single menu to more than one window!

Since fixed menus are no automatically removed, the following function must be used to remove a fixed menu from the display:

(DELETEMENU *Menu CloseFlg Window*) ž deletes *Menu* from *Window*. If *CloseFlg* is non-NIL and there are no other menus in *Window*, then CLOSEW is called on *Window*.

If *Window* is not given, (OPENWINDOWS) is searched for the window containing *Menu*. If no such window is found, DELETEMENU is a no-op.

Hierarchical Menus

You can create hierarchical menus by creating menu items that have subitems.

A menu item with subitems is displayed with a small gray arrowhead at the right edge of the item.

If the user, drags the cursor from inside the item to outside the item across the right edge, then a submenu containing the subitems will be brought up.

Selecting one of these subitem from this submenu is functionally equivalent to selecting an item from the main menu.

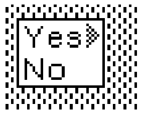
Note that the subitems may themseleves have subitems, making a fully hierarchical menu.

The MENU datatype has a field called SUBITEMFN. The value of the field should be a predicate that determines whether an item has subitems or not. The SUBITEMFN is called with the menu and the item as arguments. It should return either NIL to indicate that the item has no subitems or a list of subitems from which to use as the ITEMS field while constructing the subitem menu.

The default SUBITEMFN is the function DEFAULTSUBITEMFN which checks to see if the item is a list of 4 elements with the 4th element being a list whose CAR is SUBITEMS. If it is, it returns the CDR of this 4th element. Otherwise, it returns NIL.

Example of a menu with subitems and the default SUBITEMFN:

```
(create MENU
        ITEMS _
                ’((Yes ’Yes "Answer Yes"
                                (SUBITEMS
                                    ("Yes w/ Check" ’Yes1 "Anwser
                                    Yes, but check first.")
                                    ("Yes w/o Check" ’Yes2  "Anwser
                                    Yes, without checking first.")))
                        (No ’No "Answer No.")))
```



Menu Example

The following set of functions implements a fixed menu that sits left-flushed along the bottom of the screen. The menu has some common commands that you usually invoke from the Exec window.

Note: The function CM.MakeCommandMenu creates a fixed menu while the function CM.TEdit uses a pop-up menu.

```
(CM.MakeCommandMenu
    (LAMBDA NIL
            (ADDMENU
                (CREATE MENU
                    ITEMS _
                        ’((TEdit (CM.TEdit) "Opens up a
                        new TEdit window.")
                        (FileBrowse (CM.FB) "Opens a File
                        Browser window.")
                        (Lafite (CM.Lafite) "Starts up Lafite
                        mail program."))
                    TITLE _ "Common Commands"
                    MENUFONT _ ’(HELVETICA 14 BOLD)
                    MENUROWS _ 1
                    CENTERFLG _ T
                    ITEMHEIGHT _ 50
```

```
                                        MENUBORDERSIZE _ 2)
                                    NIL
                                    (create POSITION XCOORD _ 0 YCOORD _ 0))))
        (CM.TEdit
            (LAMBDA NIL
                (* * Ask user if New or Old file to TEdit, then open a Tedit)
                (SELECTQ (MENU (create MENU ITEMS _ ’(New Old)
                                                MENUFONT _
                                                ’(HELVETICA 14)
                                                ITEMHEIGHT _ 30
                                                CENTERFLG _ T))
                    (New (TEDIT))
                    (Old (TEDIT
                            (PROGN (PRIN1 "Enter file name: ")(READ))))
                    NIL)))
        (CM.FB
            (LAMBDA NIL
                (FILEBROWSER)))
        (CM.Lafite
            (LAMBDA NIL
                (LAFITE)))
```

Using the Mouse

Writing BUTTONEVENTFNs, CURSORxxxFNs, and WHENSELECTEDFNs

In general, programs that depend on mouse actions are best written using the Window and/or Menu packages.

The BUTTONEVENTFNs and CURSORxxFNS for windows and the WHENSELECTEDFN for menus allow you to carry out arbitrary actions whenever the user buttons down inside a window or menu item or even whenever the user moves a the cursor within a window.

For example, the GRAPHER program for editing node-link graphs is built almost entirely on BUTTONEVENTFNs that fire whenever the user buttons down inside the GRAPH window. The BUTTONEVENTFNs determine what button was pressed and where in the window it was pressed relative to the graph being displayed. Theuse this information to decide what action to carry out.

There is an important factor to pay attention to when writing BUTTONEVENTFNs, CURSORxxxFNs, and WHENSELECTEDFNs.

In particular, these functions are evaluated as part of the Mouse process (i.e., the process that tracks the cursor and interprets mouse button presses). While these functions are be evaluated, the Mouse process can't be doing anything else, i.e., it can't be carrying out its normal job of tracking the cursor and interpreting button presses.

If a BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN is going to be long running, it is good practice to spin this evaluation off of the mouse process, so that the user can go off an do other things with the mouse while the function is being evaluated.

To do this, you can use the function call **(SPAWN.MOUSE)** in the BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN.

As described in LispCourse #14 (page 16), this will make the current mouse process (the one thats going to do the long running evaluation) into a process called OLDMOUSE and then start up a new mouse process.

The BUTTONEVENTFN, CURSORxxxFN, or WHENSELECTEDFN will thus run under the OLDMOUSE process and not interfere with ongoing mouse operations.

Also, there is a convention in Interlisp-D that a events don't occur until the user releases a mouse button. For example, menu items are selected when the user releases the mouse button inside the item, rather than when she presses the mouse button.

When writing BUTTONEVENTFNs for windows, the programmer is responsible for adhering to this convention. When the BUTTONEVENTFN is called, it is good practice to wait until the user lets up on the mouse button before carryoing out any actions. If the user lets up on the mouse button outside the window, then it is considered correct return without carrying out the action.

So, at the begining of a BUTTONEVENTFN it is common practice to have a **(UNTILMOUSESTATE UP)** and **(OR (INSIDEP (WINDOWPROP Window 'REGION) (CURSORPOSITION Window)) (RETURN NIL))**. [See below for discussion of UNTILMOUSESTATE and CURSORPOSITION.

Getting Information about the Mouse and the Cursor

Many programs require information about the state of the mouse buttons or the cursor. The following functions return this information:

Position of the Cursor

(CURSORPOSITION *NewPosition Window*) ž Reads and then returns the location of the cursor in the coordinate system of *Window*. If *NewPosition* is specified, sets the cursor to be at the specified position in the window's coordinate system.

(LASTMOUSEX *Window*), (LASTMOUSEY *Window*) ž Returns the X (Y) location of the cursor in the coordinate system of *Window* as of the last time the cursor position was read. Does not actually read the current position.

LASTMOUSEX, LASTMOUSEY ž global variables that contain the X (Y) position of the cursor in the screen coordinate system as of the last time the cursor position was read.

(GETMOUSESTATE) ž Reads the current mouse state including the cursor location and sets the global variables LASTMOUSEX and LASTMOUSEY.

CURSORPOSITION calls GETMOUSESTATE before returning the cursor position. The functions LASTMOUSEX and LASTMOUSEY do not.

State of the Mouse Buttons

(MOUSESTATE *ButtonSpec*) ž Reads the mouse button state and returns T if that state matches *ButtonSpec*, NIL otherwise. *ButtonSpec* is description of the mouse buttons having one of the following forms:

LEFT, MIDDLE, RIGHT ÿ indicating the corresponding mouse button is down.

UP ÿ indicating all mouse buttons are released.

(ONLY *Button*) ž indicating that mouse button *Button* (i.e., one of LEFT, MIDDLE, RIGHT) is the ONLY button down.

(AND *ButtonSpecs*), (OR *ButtonSpec*s), (NOT *ButtonSpec*) ÿ indicating the logical combinations of other *ButtonSpecs*.

Note: MOUSESTATE is a macro which is like an NLAMBDA function in that its arguments are not quoted.

Examples:

(MOUSESTATE LEFT)

(MOUSESTATE (ONLY LEFT))

(MOUSESTATE (OR LEFT MIDDLE))

(MOUSESTATE (AND (NOT UP)(NOT (ONLY LEFT)))

(LASTMOUSESTATE *ButtonSpec*) ž Like MOUSESTATE but does not first read the current mouse state. Thus it returns the mouse state as of the time it was last read. Good for looking at exactly what caused MOUSESTATE to return T. (Does not first call GETMOUSESTATE as does MOUSESTATE.)

(UNTILMOUSESTATE *ButtonSpec Interval*) ž waits until (MOUSESTATE *ButtonSpec*) returns T or until *Interval* milliseconds have elapsed.

If (MOUSESTATE *ButtonSpec*) returns T, then UNTILMOUSESTATE returns T. If *Interval* milliseconds elapses without (MOUSESTATE *ButtonSpec*) returning T, then UNTILMOUSESTATE eretunrs NIL.

If *Interval* is NIL, then UNTILMOUSESTATE will wait indefinitely.

Example of Using the Mouse

The following two functions implement a trap window ž if the user rolls the cursor into the window, he cannot roll it out again. Every time the cursor nears the edge of the window, the window jumps to be centered around the cursor location. Thus wherever the cursor moves, the window will follow.

(Being a nice guy, there is an escape.)

```
(EX.CreateWindow
  (LAMBDA NIL                                (* fgh: "29-Jun-85 16:26")
    (LET ((Window (CREATEW (CREATEREGION 100 100 300 150))))

         (* * Add special CURSORINFN to Window)

      (WINDOWPROP Window (QUOTE CURSORINFN)
                  (FUNCTION EX.CursorInFn))

         (* * Add a warning title)
      (WINDOWPROP Window (QUOTE TITLE)
                  "Caution: This window is trap"))))


(EX.CursorInFn
  (LAMBDA (Window)                            (* fgh: "29-Jun-85 16:28")

          (* * The user has moved inside of the window, don't let him out)

    (LET ((EdgeWidth 10)
       (Region (DSPCLIPPINGREGION NIL Window))
       (HalfWidth (QUOTIENT (fetch (REGION WIDTH) of (WINDOWPROP
                                                      Window
                                                      (QUOTE REGION)))
                            2))
       (HalfHeight (QUOTIENT (fetch (REGION HEIGHT)
                                of (WINDOWPROP Window (QUOTE REGION)))
                             2))
       (Position (CURSORPOSITION NIL Window))
       NewRegion)

          (* * Compute a region just inside of the border of the window)

      (SETQ NewRegion (CREATEREGION (PLUS EdgeWidth (fetch (REGION LEFT)
                                                      of Region))
                                    (PLUS EdgeWidth (fetch (REGION BOTTOM)
                                                      of Region))
                                    (DIFFERENCE (fetch (REGION WIDTH)
                                                  of Region)
                                                EdgeWidth)
                                    (DIFFERENCE (fetch (REGION HEIGHT)
                                                  of Region)
                                                EdgeWidth)))

          (* * Forever, if the cursor moves out of the inner region move the
window to center on the cursor position)

      (while T
         do (COND
              ((INSIDEP NewRegion (CURSORPOSITION NIL Window Position))
                        (* Allow user to escape if
                           chords the left and right mouse buttons)
                (AND (LASTMOUSESTATE (AND RIGHT LEFT))
                     (RETURN NIL)))
              (T (MOVEW Window (DIFFERENCE LASTMOUSEX HalfWidth)
                         (DIFFERENCE LASTMOUSEY HalfHeight))))
                          (* Allow other processes to run)
            (BLOCK)))))
```

References

Chapter 19 of the IRM! But beware that this material is fairly old and out-of-date. Look in the Interlisp release messages for updates.