M E M O

To: Alto and PARC Lisp groups

Date: November 18, 1974

From: P. Deutsch

Location: Palo Alto

Subject: Display primitives in Lisp

Organization: PARC/CSL

File: <LPD>LSDISP.PUB;18
Archive category: Lisp

****** DRAFT!!! COMMENTS PLEASE!!! YOUR IDEAS ARE NEEDED!!! ******

Several conflicting goals must be resolved in deciding on a set of display facilities for Lisp: ease of use, efficient access to hardware facilities, and device- and system-independence. This memo suggests a set of facilities constructed in two layers: a lower layer that gives direct access to the Alto bitmap capability, while retaining Lisp's tradition of freeing the programmer from storage allocation worries, and an upper layer that uses the lower (on the Alto) or a character-stream protocol (for VTS, on MAXC) to provide for writing strings, scrolling, editing, etc. on the screen.

1. Bitmap level

At this level we introduce two new types of object: the bitblock and the slice. Bitblocks (or simply blocks) are just rectangular chunks of display memory. Bitblock primitives allocate bitblocks, and provide for writing strings and individual bits. Slices describe how pieces of bitblocks are mapped onto the screen, i.e. giving vertical position, background color, indentation, magnification, etc. Slice primitives create and set the parameters of slices, and couple and decouple them to (parts of) bitblocks.

1.1. Bitblock primitives

bitblock[width;height]

Creates and returns a new bitblock of specified width and height. Width and height are given in bits. The block is initialized to zeros.

bitblockwidth[block]

Returns the width of block.

bitblockheight[block;height]

Returns the height of block.

Blocks are garbage-collected like any other data type, when there are no references to them from either Lisp data structures or slices. Thus there is no need for a "release block" operation.

clearblock[block;value;xpos;ypos;width;height]

Clears the given subrectangle of block to 0 (value=NIL or 0) or 1 (value=T or 1). Error if any parameter is invalid, i.e. xpos<0 or ypos<0 or width<0 or height<0 or xpos+width>blockwidth[block] or ypos+height>blockheight[block]. Xpos and ypos default to 0; width defaults to blockwidth[block]-xpos; height defaults to blockheight[block]-ypos.

copyblock[srcblock;xsrc;ysrc;width;height;destblock;xdest;ydest;mer

Copies the given subrectangle of srcblock into a subrectangle of destblock. Merges bits if merge≠NIL, otherwise replaces bits.

modifyblock[block;xpos;ypos;data;mode]

Modifies a string of bits in block extending leftwards from position (xpos,ypos) according to data and mode. If mode=NIL, sets bits selected by 1's in data; if mode=T, clears bits selected by 1's in data; otherwise, mode must be a number, and replaces that many bits with the right-hand bits of data.

blockprin[block;xpos;ypos;datum;xlimit;flag;font]

Writes datum into block at position (xpos,ypos). If flag=NIL, does PRIN1, otherwise takes flag as a readtable and does PRIN2. If font=NIL, uses the standard font; otherwise, font must be an array organized as a standard Alto font (see the Alto manual for details). If fn=NIL, skips any characters which do not appear in the font; otherwise, performs fn[datum;chno] where chno is the position of the character in datum (a la NTHCHAR), and continues printing after fn returns. If xlimit (or the width of block, if xlimit=NIL) would be exceeded by a given character, performs fn[datum;chno] and returns the value of this call; otherwise returns the X position just beyond the last character written.

The primitives which modify the contents of bitblocks can also operate on arrays. Such arrays must be initialized with the "width" in element 1 and the "height" in element 2, both given in bits; the data is stored 32 bits per element, row-wise, left to right, top to bottom, with the width rounded up to the next multiple of 32 (as might be expected). The implementation presently being considered also requires that the entire array fit in core, which corresponds to a size restriction of about 5K 32-bit words or 1/6 of a screenful.

There is a permanently defined block called the cursor block, and an associated slice called the cursor slice. There is nothing special about the cursor block -- its width and height are 16 bits, and all the bitblock operations work normally on it. The cursor slice, on the other hand, does have some special properties discussed under slices below.

1.2. Slice primitives

Slices are slight abstractions of the Alto display control blocks. They have the following components:

A body, which is a subrectangle of a bitblock beginning at the left edge;

A position (X,Y) on the screen, subject to the restriction that two active slices must not overlap in Y;

A background color, white or black;

A resolution, normal or halved;

An activity flag, which determines whether the slice is actually displayed on the screen.

Other hardware-imposed restrictions are discussed below.

slice[oldsl;param1;value1;...;paramn;valuen]

If oldsl=NIL, creates a new slice with parameters initialized as given; otherwise, alters parameters of the slice oldsl. Value is oldsl or the new slice. Possible values of param are:

XPOS: the X position (must be a multiple of 16);
YPOS: the Y position of the lower edge (must be even);
BLOCK: the bitblock;
YD: the Y displacement within the bitblock corresponding to YPOS;
HEIGHT: the vertical extent of the data to be displayed from the bitblock (must be even);
COLOR: the background color, 0=white, 1=black;
RESOLUTION: the resolution, 0=normal, 1=halved;
ACTIVE: the activity flag, NIL or T.

The default values for the parameters are XPOS=YPOS=0, BLOCK=NIL, YD=0, HEIGHT=0, COLOR=0, RESOLUTION=0, ACTIVE=NIL.

sliceparam[sl;param]

Returns the param parameter value of slice sl.

The cursor slice has all its parameters fixed except XPOS and YPOS, which may be set to any values whatever (not limited to multiples of 16 or 2 respectively). It is always active: the cursor data is XOR'ed with any other displayed data it happens to overlap. See the Alto manual for further details.

2. Stream primitives

Display streams (or simply streams) behave like output files with respect to the system printing functions, but write into selected portions of blocks and trap out when an attempt is made to write beyond their limits.

stream[st;param1;value1;...;paramn;valuen]

Interpretation is similar to slice; returns st or a new stream. The parameters are:

BLOCK: the bitblock, must be specified at creation time;
XD: the X displacement of the string origin within the bitblock;
YD: the Y displacement of the string origin;
WIDTH: the maximum width of the string (in bits);
FN: the function to be called on overflow, i.e. an attempt to write beyond the maximum width or to write a character not specified by the font;
XPOS: the X position for the next character, relative to XD;
FONT: the font.

Legality checking is performed after all arguments have been absorbed, so (for example) XD, XPOS, and WIDTH may be specified in any order.

streamparam[st;param]

Analogous to sliceparam.

When a stream overflows, its FN is called with the arguments [datum;chno;stream] where datum is the datum that was being printed and chno is 1+ the number of characters actually added to the display before overflowing,

i.e. the number of the next character to print. FN will normally reset stream in some manner. It then has the options of printing the rest of datum itself and returning T, doing no printing and returning NIL, or erasing some or all of the part of datum already printed and returning a character position within datum (analogous to chno) at which to restart printing. Overflowing a stream with no FN generates an error.

The operations of closef, position, openp, and output on streams are straightforward. (Closing a stream does not erase the characters.) I would be inclined to resolve two other potential issues as follows: writing merges the characters into the block, as opposed to replacing the bits (you can always use clearblock); sfptr is only legal with values of 0 and NIL. The latter restriction may be relaxed if it is deemed advisable to retain a record of the characters written.

3. String editing

At the higher level, facilities are needed for constructing windows (arrays of strings which scroll properly) and for performing editing operations on strings. Editing (backspace, insert, delete, etc.) requires that every character written on a window also be saved in a Lisp string somewhere. It is probably best to provide this facility at the lower level, per open stream.