SCOUT - a definitive notation for SCreen layOUT

SCOUT is a definitive notation for describing screen layout. By means of definitions, SCOUT primarily describes the geometry of the layout of windows in a display. SCOUT also serves as a link with other definitive notations. For instance, one can define a SCOUT window in which some part of a DoNaLD (the line drawing notation) picture is displayed. SCOUT windows can also be sensitive so that events (mouse or keyboard actions) happening in that window can be detected. Therefore SCOUT can also support the development of simple user interfaces.

1 Understanding the SCOUT Notation

SCOUT describes a display as (potentially) overlapping windows. For example, if display disp is defined as:

```
disp = < win1 / win2 >
```

this means that display `disp` consists of two windows `win1` and `win2`, should `win1` and `win2` overlap, `win1` overlays `win2`.
The best way of understanding what SCOUT windows are is through the formula below:

```
window = region X content X attributes
```

A window defines a region in which something will be displayed in a certain way.

1.1 Different types of SCOUT windows

There are four kinds of windows in the current SCOUT notation: text window, textbox window, DoNaLD window and image window. Because of the different nature of the windows, their definitions of region, content and attributes may differ.

1.1.1 Text windows

- region (called a frame) – list of boxes. The string is filled into the first box, the remaining characters are filled into the second box and so on.
- content – a character string. A character string.
- attribute – {fgcolour, bgcolour, border, bdcolour, relief, alignment}. These attributes indicate the colour of the text string, the colour of the background, whether the boxes have borders, the border colour, the kind of border and the alignment of strings in relation to the boxes respectively. The acceptable reliefs are raised, sunken flat, ridge and groove.

1.1.2 Textbox windows

- region – a box
- content – a character string
- attribute – {fgcolour, bgcolour, border, bdcolour, relief, alignment}. As for the text window, but the alignment attribute defaults to LEFT.

1.1.3 DoNaLD windows

- region – a box. A box.
- content – a drawing (name of the drawing). A drawing.
- attribute – {xmin, ymin, xmax, ymax, fgcolour, bgcolour, border, bdcolour relief}. xmin, ymin, xmax, ymax defines the coordinate system of the drawing; fgcolour and bgcolour defines the foreground and background colour, border determines whether to draw borders of the box, bdcolour defines the border colour and relief how the border is drawn.

1.1.4 Image windows

- region – a box. A box.
- content – an image (name of the image variable). An image.
- attribute – {bgcolour, border, bdcolour relief}. bgcolour defines the colour to be filled when the image is not large enough to cover the whole area of the window. border determines whether to draw borders of the box. bdcolour defines the border colour and relief how the border is drawn.

1.2 Mouse sensitive windows

The sensitive attribute is common to all four types of windows. It is used to declare that a window is sensitive to mouse and keypress actions. When this attribute is ON, a mouse action or a keypress action within the region of this window will cause a definition to be generated. If a mouse action occurs in a window and it is a DoNaLD or image window, then the window name concatenated with `_mouse` will be the name of the variable to be defined; if it occurs in a text window, the window name concatenated with `_mouse_` followed by the box number will be the variable name. The value assigned to the appropriate variable records the nature and the location of the mouse action. It is a 5-tuple of (button, type, state, x, y) where

1. button – the button number pressed or released;
2. type – the button action (4 = pressed, 5 = released);
3. state – the state before the button action occurred (shift (+1), caplock (+2), control (+4), meta (+8) and was-pressed (+256)). For example, if a button is released while the shift and control keys are depressing, state will be 1 + 4 + 256 = 261;
4. x – the x- coordinate of the mouse in the coordinate system of the window in which the mouse action occurred.
5. y – the y- coordinate.

As with mouse events, a stroke on the keyboard will generate a definition. Instead of `_mouse` or `_mouse_` followed by a box number, the variable name of the generated definition will end with `_key` or `_key_` followed by a box number. The value defined will also be a 5-tuple: (key, type, state, x, y), where key

is the ascii code of the key pressed.

In principle, there could be many types of windows, many more attributes and many ways of defining regions. The current notation only demonstrates the principle of using definitions in describing screen layout.

2 Data Types and Operators

Because there is such flexibility in the design of the window data type, the set of data types and operators in SCOUT may be extended in the future. There are, however, some essential data types in SCOUT: integer, point, window and display. Associated with them are basic operators for integer arithmetic, vector manipulation, list manipulations, construction and selection. The following shows the basic SCOUT operators and functions for the four essential data types. All the operators of the SCOUT notation can be found in the next section.

- Operators: +, -, *, /, % (remainder), - (unary minus)
  Meaning: Normal integer arithmetic
  Example: 10 % 3 gives 1

- Constructor: x, y Meaning: Construct a point
  Example: 10, 20 is a point with x-coordinate 10 and y-coordinate 20

- Operators: +, -
  Meaning: Vector sum and vector subtraction
  Example: 10, 20 - 20, 5 gives -10, 15

- Selector: .1, .2
  Meaning: Return the 1st (x-) coordinate and the 2nd (y-) coordinate respectively
  Example: 10, 20.1 gives 10

- Constructor: field-name: formula, field-name: formula, ..., field-name: formula
  Meaning: Constructing a window
  Example: type: DONALD, box: b, pict: "FIGURE1"

- Selector: .field-name
  Meaning: Return the value of the field
  Example: type: DONALD, box: b, pict: "FIGURE1".box gives b

- Constructor: ⟨ W1 / W2 / ... / Wn ⟩
  Meaning: Constructing a display, if W1 and W2 overlap, W1 overlays W2
  Example: ⟨ don1 / don2 ⟩

- List function: insert(L, pos, exp)
  Meaning: Insert the expression exp, at the position pos, into list L
  Example: `insert( ⟨w1, w2, w3⟩, 2, new)` gives `⟨w1, new, w2, w3⟩`

- List function: delete(L, pos)
  Meaning: Delete the posth element of list L
  Example: `delete(⟨w1, w2, w3⟩, 2)` gives ⟨w1, w3⟩

- Operator: if cond then exp1 else exp2 endif
  Meaning: if cond gives non-zero value (true) then returns exp1 else returns exp2, in this context, exp1 and exp2 must have the same type.
  Example: `if 1 then "Open" else "Close" endif gives "Open"`

3 Text and Image Windows

3.1 Text Windows

Unlike line drawing for which we have a separate definitive notation, we are yet to define a definitive notation for text processing. Part of the SCOUT notation is therefore designed to cope with simple text layout. To this end, SCOUT incorporates a text window subtype. This text window subtype differs from other window subtypes in that the content of the text window subtype is a string defined within SCOUT rather than a virtual screen described outside SCOUT by another definitive notation. As a result, string becomes one of the SCOUT data types.

Associated with the string data type is a set of operators useful for displaying a text string. String concatenation (//), string length function (strlen) , sub-string function (substr) and integer-to-string conversion (itos) are the basic SCOUT string manipulation functions. There are two postfix operators - .r and .c: since the basic geometric unit in SCOUT is the pixel but the size of a block of text is more conveniently specified as "number of rows by number of columns", it is convenient to introduce functions returning the row height and the column widths in pixels. .r is the function meaning "multiply by the row height" and .c is the function meaning "multiply by the column width". These functions are appropriately represented by postfix operators because they work very much like units. For examples, 10.c, 3.r refers to a point 3 rows down and 10 columns right to the origin. A similar consideration influences the design of a box, a data type for defining regions. The region associated with a box is sufficiently defined by its top-left corner and its bottom-right corner, and this is a convenient method of definition in the case of graphics. For a block of text, however, the bounding box is more conveniently defined by specifying the top-left corner and the dimensions of the box in terms of number of rows and columns. For instance, [0,0,3,10] refers to a box with the origin as its top-left corner which is suitable for displaying three rows by ten columns of text.

Because displaying a string is different from displaying an image, the way of specifying a region for displaying text is different from that for displaying an image. As attested by the fact that the earlier releases of the X Window system had only primitives for creating rectangular windows, a simple box is adequate for display purposes in most applications. But, when considering the possible application of text display to desktop publishing, we know that a piece of text may be displayed across several regions; defining a region by a box is simply not sufficient. Our solution is to define a region by an ordered list of boxes. A string should be filled in the first box first, then the second, and so on. Therefore we define a data type frame which is a list of boxes. Operators for lists can be applied to frames.

4 Textbox Windows

Textboxes contain strings that may be edited by the user. An example definition is shown below:

```
window fromText = {
    type: TEXTBOX
    frame: ([{60, 20}, {30, 1}])
    border: 1
    sensitive: ON
};
screen = <fromText>;
```

Note that sensitive must be ON for the user to be able to edit the contents of the box. Also note that the second "point" given for frame is not actually a point - for TEXTBOX only, the example above means a box 30 characters long, and 1 character deep. The same effect can be achieved in TEXT windows by specifying two integers rather than a point, for example frame: ([60, 20, 1, 30]), or perhaps calculating

the point using the .c and .r operators: frame: ([60, 20, 60, 20 + 30.c, 1.r]).

When the contents of a textbox is changed (actually, when the key is released), a variable is automatically redefined. The name of this variable is:

```
name of the window // "_TEXT_" // display box number
```

For the above example, it would be fromText_TEXT_1. Definitions or triggered procedures can be used to deal with input as shown below.

```
%eden
fromTextAppendHi is fromText_TEXT_1 // " hi";
proc p : fromText_TEXT_1 { writeln(fromText_TEXT_1); }
```

Note that data read from a text box may contain an extra appended newline (or two). This may be changed in future versions of Eden.

An alternative interface is available, which is not recommended for new models (if possible) due to its procedural nature. When a TEXTBOX is created, two Eden procedures are automatically created to enable access to the text in the box. The procedures names start with the name given to the window, and are followed by _getText and _setText. The SCOUT example given above could be used with the following Eden code:

```
%eden
test = fromText_getText();
fromText_setText(test // " and hi mum");
```

4.1 Image Windows

```
Example:

real xscale = 0.5;
real yscale = 0.5;
image source = ImageFile("gif", "logo.gif");
image sy = ImageScale(source, xscale, yscale);
window firstImage = {
    type: IMAGE
    box: [{10,10}, {490,240}]
    pict: "sy"
    border: 5
    relief: "raise"
};
window secondImage = {
    type: IMAGE
    box: [{10,260}, {490,490}]
    pict: "source"
    border: 5
    relief: "raise"
};
screen = <firstImage/secondImage>;
```

will give you two windows. The upper one shows a mini version of the image of display in the lower one. As of now, the functionality is very limited. Only two functions are defined: ImageFile() and ImageScale(). And the images have to be centred in the window.