Lecture T2: The Design of Scout

Work of Simon Yung

insight into issues for the design of a definitive notation.

Design of SCOUT

primitives reflect nature of the screen and application

e.g. combinations of rectangular boxes for tex t layout

assumes special-purpose notations for any graphics

Role of SCOUT

- screen layout, scaling and operations at pixel level
- coordinates the use of different definitive notations
- gives basic text display facilities

[For serious text-processing require another definitive notation

e.g. can't control font size or get sophisticated layout]

SCOUT data types

Essential data types in SCOUT:

display, window, **point, integer**

Overall concept

definitive script in SCOUT defines state of the screen

screen is a special variable of type *display*

the display is made up out of windows

Simplest definition of screen has the form

```
screen = < win1 / win2 / win3 / win4 / win5 / .... >
```

where ordering of windows determines how they overlay

Alternatively can realise screen as union of displays

```
screen = disp1 & disp2 & disp3 & disp4 & ....
```

T2.1. The Window Data Type

Layout design in Scout addresses the following three questions

What are the things to be displayed?

Where are they to be displayed?

How are they to be displayed?

=> Scout *window* data type

The type *window* is a union of subtypes:

one subtype for each definitive notation

each subtype has a number of fields

the number and types of the fields vary with the subtype

In general, a *window* should have fields that define

1) which definitive notation is concerned
2) what information is to be displayed

   e.g. which DoNaLD picture or which text string
3) the region of the screen

   onto which the required information is mapped
4) the supplementary info that that definitive notation needs.

In the current design and implementation, have 3 window types:

the text window, the DoNaLD window and the ARCA window.

Form of text window definition

Text Window		
field name	***type***	***description***
type	content[1]	Must be the value `TEXT`
string	string	The string to be displayed
frame	frame	The region in which the string is shown
border	integer	Width of the border of the boxes of the frame
alignment	just[2]	`NOADJ`, `LEFT`, `RIGHT`, `EXPAND` and `CENTRE` are the possible values to denote no alignment, left justification, right justification, left and right justification and centre of the text inside each box in the frame
bgcolour	string	Colour name for the background colour of the text
fgcolour	string	Colour name for the (foreground) colour of the text

where point = integer · integer

box = point · point

frame = *list of* box

[1] The type *content* is currently a set { TEXT, DONALD, ARCA }.

[2] The type *just* is { NOADJ, LEFT, RIGHT, EXPAND, CENTRE }.

Form of DoNaLD window definition

DoNaLD Window		
field name	***type***	***description***
z	content	Must be the value `DONALD`
box	box	The region in which the DoNaLD picture is shown
border	integer	Set the border width of the bounding box
pict	string	The name[3] of the DoNaLD picture
xmin	point	Show the portion of the DoNaLD picture bounded by the points (xmin, ymin) and (xmax, ymax)
ymin	point	
xmax	point	
ymax	point	

[3] There is no picture name specified in the original DoNaLD notation, but there is now a statement

`viewport` *`name`*

required before the DoNaLD definitions to identify which picture these definitions are defining.

Comments on the window data types:

1) Easy to add other attributes to windows

2) There is no formal restriction on how to define a region.

   By convention

   - for a DoNaLD or ARCA window a region is defined by a box
   - for a text window, a region is defined by a list of boxes

3) Graphics & text window subtypes almost no fields in common

   type *window* may be better understood by the abstract formula

   $$\text{window} = \text{region} \cdot \text{content} \cdot \text{attributes}$$

   rather than by a concrete set of fields.

T2.2. The Display Data Type

A *display* is a collection of windows. Because the windows may overlap, there is a partial ordering among the windows. For simplicity, a *display* is defined to be a list (total ordering) of *window*s.

In general, a *display* variable represents a conceptual screen; only the distinguished *display* variable *screen* denotes the physical screen. There are simple rules for mapping the variable *screen* onto the physical screen:

1) the origin is defined at the top-left corner of the physical screen;

2) the x-coordinate counts from the origin to the right, one unit per pixel;

3) the y-coordinate counts from the origin to the bottom, one unit per pixel.

It is obvious from the mapping rules that the interpretation of the Scout notation, unlike other definitive notations, is hardware dependent. The same script of Scout definitions may have a slightly different look on a monitor with different resolution and aspect ratio.

T2.3. Other Data Types and Operators

Because there is a great 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 table shows the basic Scout operators and functions for the four essential data types.

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 2nd (y-) coordinate resp.

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* / ... / >

Meaning: Construct a display; if *W1* and *W2* overlap, *W1* overlays *W2*

Example: < don1 / don2 >

List fn: insert(*L*, *pos*, *exp*)

Meaning: Return *L* with the expression *exp* inserted in position *pos*

Example: insert(<w1, w2, w3>, 2, new) gives <w1, new, w2, w3>

List fn: delete(*L*, *pos*)

Meaning: Return *L* with the *pos*th element deleted

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*, here,*exp1* and *exp2* must have the same type.

Example: delete(<w1, w2, w3>, 2) gives <w1, w3>

As mentioned, text layout should ideally be described by another definitive notation. Since that notation does not exist, part of the Scout notation is designed for 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 prescribed 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 – which are specially designed. 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 width 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 example, {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. More examples of this kind can be found in the Jugs example in ~wmb/public/SCOUT/tank.s.

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. Our solution is to divide an arbitrary shape into subregions, each of which is a box. The definition of a region will then be a ordered list of boxes. For example, the region depicted in Figure 1 below is interpreted in such a way that a string should be filled in the first box first, then the second, then the third.

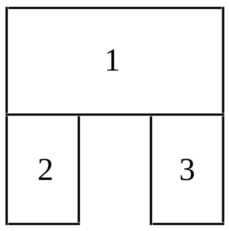

Figure 1: A Way of Partitioning a Non-rectangular Region

This "frame = list of boxes" definition of region is not perfect. For instance, if two boxes overlap (which may depict the overlapping of two sheets of the same document), which box should be put on top is still ambiguous. However, except for serious desktop publishing, this definition of region should be adequate for most applications.