Drawing diagrams

Arjen Markus[1]
WL|Delft Hydraulics

Introduction

Much software today is geared to interactive use and graphical user-interfaces abound, especially on the MS Windows platform. The primary paradigm is WYSIWYG: What You See Is What You Get. This can lead to awkward situations that we all know too well:

- Word processors stuffed with options you never use but see anyway. They expect you to be a half-decent markup specialist so that you at least understand all the terms they use.
- Internet browsers that print an exact copy of what you saw on the screen, only too happy to forget that the paper is too narrow and that you are not interested in the navigational frame on the left.

Perhaps the most dramatic failure of this paradigm is that of drawing programs. Try drawing a simple diagram, for instance one illustrating the interaction between a PC, a network and the printers. Typically you will spend a lot of time positioning the boxes, the text inside or near those boxes and the arrows between them.

Sure, there are lots of computer programs dedicated to the job: look at software design programs that allow you to draw all different types of diagrams defined by the *unified modelling language* - UML. Look at such well-known software packages as Visio or IgrafX - these are magnificent tools in the hands of an expert, with libraries of symbols from any field of industry imaginable. But for those users who only need to make a little sketch every now and again they are very heavy tools indeed.

An alternative is to use a simple decriptive language: define what items to draw, specify their positions relative to each other and have the software decide where and how to draw them exactly.

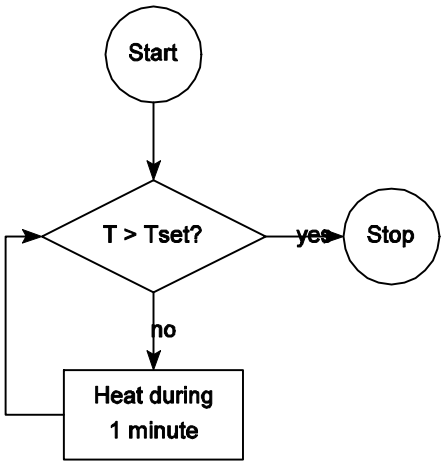
An example: a flow diagram

Suppose you have a heating device that tries to keep the the living-room at (more or less) the same temperature. All it does is:

- Measure the current temperature
- Check if it is lower than the target temperature
- If so, turn on the heater for a while
- Otherwise do nothing
- Check again from time to time

Below is a simple drawing that illustrates the process:

[1] Email: arjen.markus@wldelft.nl



The drawing was creating with the following Tcl code:

```
source draw_diagram.tcl
namespace import ::Diagrams::*

pack [canvas .c -width 300 -height 300 -bg white]
drawin .c

currentpos [position 100 20]
circle "Start"
direction south
arrow "" 40

#
# Store the objects for later reference
#
set d [diamond "T > Tset?"]
arrow "no" 40
set b [box "Heat during\n1 minute"]
#
# Note: the order of direction/currentpos is important
# (The [direction] command sets the current position too)
#
direction east
currentpos [getpos E $d]

arrow "yes" 40
circle "Stop"

bracket west 30 [getpos W $b] [getpos W $d]
```

The Diagrams package that is used (http://wiki.tcl.tk/13434) simplifies the positioning and sizing of the each separate canvas item:

- It keeps information about a current position and a current direction
- It "knows" about the hooks or anchor points of each "object"
- It computes the dimensions of the boxes and other polygons from the given text, rather than that you need to do it yourself

Because the positioning and sizing is done automatically you can easily add new objects, change the distance between them and so on. Relations like alignment of the centres of the objects are strictly kept - one of the most tedious aspects of general drawing programs.

The inner workings

Before we show some other possible uses of the general approach in the Diagrams package, let us look at the underlying strategy. The first versions, mere experiments, were created independently of the once famous PIC program by B. Kernighan (http://www.troff.org/papers.html), but they were extended later on with that program in mind.

Essential to the implementation is that the current settings for colour, direction, position and so on are kept hidden from the user. Then there is a small set of conventions to adhere to:

- Objects have a complete set of anchor points (known by the eight main compass directions and the "centre") which can be queried.
- Objects "know" which canvas items they are composed of, so that all can be repositioned if needed.
- For lines and arrows the list of anchor points is slightly different: based on the breakpoints, rather than on more abstract positions around the canvas items.

To illustrate the implementation here is the code to draw a box with text inside:

```
# box --
#    Draw a box from the current position
# Arguments:
#    text      Text to be fitted in the box
#    width     (Optional) width in pixels or "fitting"
#    height    (Optional) height in pixels
# Result:
#    ID of the box
# Side effect:
#    Box drawn with text inside, current position set
#
proc ::Diagrams::box {text {width {}} {height {}}} {
    variable state

    #
    # Before we create the text object, we need to store the
    # current position ...
    #
    pushstate
    set textobj [plaintext $text $width $height]

    foreach {dummy x1 y1} [getpos NW $textobj] {break}
    foreach {dummy x2 y2} [getpos SE $textobj] {break}

    set x1 [expr {$x1-5}]
    set y1 [expr {$y1-5}]
    set x2 [expr {$x2+5}]
    set y2 [expr {$y2+5}]

    #
    # Construct the box
    #
    set     items [lindex $textobj 1]
    lappend items [$state(canvas) create rectangle $x1 $y1 $x2 $y2 \
                       -fill    $state(fillcolor) \
                       -outline $state(color)]
    $state(canvas) raise [lindex $items 0]

    #
    # Move the combined object to the original "current" position
    #
```

```
        popstate
        set obj [moveobject [list BOX $items [boxcoords $x1 $y1 $x2 $y2]]]
        set state(usegap)   1
        return $obj
    }
```

Each diagram object consists of a list of the following items:

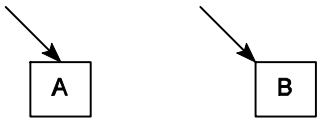
- The keyword "BOX" (to indicate it has anchor points like N and E)
- A list of canvas item IDs (to move the whole object around)
- A list of xy coordinates, one pair for each anchor point

The procedure starts by creating the text item inside the box via the `plaintext` procedure - this particular procedure computes the extents of the text and uses this to define the anchor points. The `box` procedure then relies on this information to define the coordinates for the surrounding rectangle.

Once these are known, a new canvas item can be created, with the various options stored in the `state` array.

The last few lines of the procedure are mainly administrative in nature, but they are essential:

- The object must be moved into the right spot - that is: the correct anchor point must coincide with the current position, as illustrated below with two different choices for the anchor point:



The code for these fragments is:

```
pack [canvas .c -width 200 -height 120 -bg white]
drawin .c

direction southeast
currentpos [position 20 20]
arrow "" 40
box " A "

direction southeast
currentpos [position 120 20]
arrow "" 40
attach northwest
box " B "
```

- The current position must be adjusted, so that the next user command can be run correctly. This is also the reason for the `pushstate`/`popstate` commands: they preserve and restore the user's settings.

All different diagram objects are drawn by different procedures. This allows greater flexibility at the cost of some code duplication - an alternative would have been to use a set of small procedures that delegate the work to larger ones with many parameters, but with the current approach and the use of helper procedures such as `boxcoords` to compute the default set of anchor points, it is much easier to expand the package.

Beyond simply drawing pictures …

Sofar we have concentrated on a convenient way to draw pictures with fixed symbols, but you can use this technique for much more. This section illustrates that with a few examples.

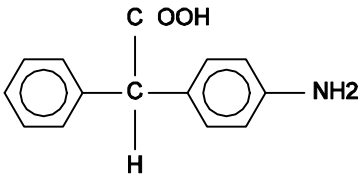
Interactive diagrams

Consider such human activities as *trouble shooting* or *filling in a form*: often you need to make a decision that leads to a different part of the underlying decision tree or a different part of the form, as certain questins do not concern you. You could use a diagram with boxes and diamonds to visualise the flow and make the various objects interactive: when you click on them a window pops up that allows you to fill in the relevant data. The diagram would be a compact navigational tool. When the underlying program properly manages what can be activated, it would be difficult for the user to accidentally skip questions.

Also with a suitable choice of symbols, the diagram can visualise the state of affairs in, say, a power plant – nothing new: such graphical overviews have been in use for many years.

Chemical structure formulae

While we have seen symbols that come from the classical flow diagrams that used to be used for depicting the structure of compter programs, we can also look at chemical structure formulae:



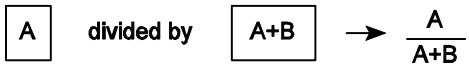
This was created with the following code:

```
direction east
currentpos [position 40 60]
benzene; bond;

set Catom [plaintext C]

bond 90 $Catom
direction north
plaintext C
direction east
plaintext OOH

bond -90 $Catom
direction south
plaintext H

bond 0 $Catom
direction east
benzene
bond; direction east
plaintext NH2  ;# UNICODE \u2082 is subscript 2, but that is not
                # supported in PostScript, it seems
```

This particular example was directly inspired by the CHEM program (*cf.* http://www.troff.org/papers.html). Supporting the full range of possibilities is a considerable challenge, but somebody with a keen interest in the matter can certainly achieve it.

Mathematical formulae

Typesetting mathematical formulae is a kind of black art, unless you realise, again following the footsteps of Kernighan, that things can be done recursively. For instance, to typeset a fraction, you first typeset the numerator and the denominator, each characterised by some bounding box. Then you can position them around the horizontal bar.

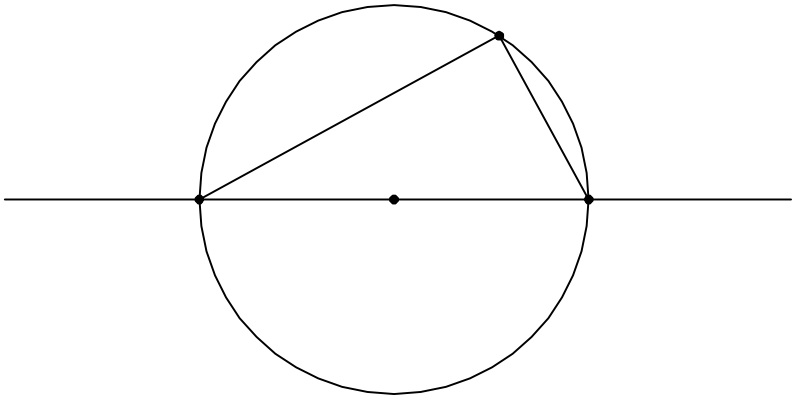


Unfortunately, I had to cheat a bit, as the code to layout fractions automatically is not finished yet. But here is the code to achieve the above "formula":

```
pack [canvas .c -width 300 -height 80 -bg white]
drawin .c

direction east
currentpos [position 20 40]
box "A"
plaintext "divided by"
box "A+B"
plaintext " "
arrow "" 20
plaintext "      "
set pos [currentpos]
lset pos 2 15

direction south
currentpos $pos
plaintext "A"

usegap 0
set xc [lindex [currentpos] 1]
set xb [expr {$xc-15}]
set xe [expr {$xc+15}]
.c create line $xb [lindex [currentpos] 2] $xe [lindex [currentpos] 2]

plaintext "A+B"
```

Geometrical constructions

As a final example I want to show a simple geometrical diagram (http://wiki.tcl.tk/13314). A well-known theorem about circles is that the triangle formed by the two point of intersection of a line through the centre and a point on the perimeter has an angle of 90°:



And the code:

```
pack [canvas .c -width 400 -height 300 -bg white]

circle [point  0 0] 1
line   [point -3 0] [point 3 0]
set p  [point [expr {cos(1)}] [expr {sin(1)}]]
line   [point  1 0] $p
line   [point -1 0] $p

#
# Shift the drawing to centre it
#
.c move all 0 -50
```

So, with a little effort one can create a useful descriptive language to easily draw diagrams.