CS405 Introduction to Empirical Modelling
Lab 4: Line drawing and animation using DoNaLD

Empirical Modelling Research Group
http://www.dcs.warwick.ac.uk/modelling/

Introduction

In laboratory session 3 you were introduced to the SCOUT notation and you should have developed an interface for a simple jugs model. This laboratory session will introduce the DoNaLD notation for line drawing and complete the four week introduction to the `tkeden` tool and the standard notations. In particular, by the end of the session you should know how to write DoNaLD definitions, and use DoNaLD together with the EDEN and SCOUT notations.

During laboratory 3 you modelled two jugs and the actions of filling, emptying and pouring. You can use the jugs model you developed as the starting point of this laboratory. You should have a model consisting of two jugs with liquid and five buttons to perform the actions. Changing the values of `contentA` and `capA` should alter the amount of liquid and the jug capacity respectively.

1 Adding a DoNaLD window in SCOUT

Before you write any DoNaLD definitions, you must have an area to display your DoNaLD drawing. This can be achieved by creating a SCOUT window of type `DONALD` and adding this to the SCOUT `screen`.

```
%scout
window mydrawing = {
  type: DONALD
  box: [{0, 0}, {300, 300}]      ## size and position (different to frame)
  pict: "DRAWING1"               ## DoNaLD viewport
  bgcolour: "white"
};
```

> **Task 1**
>
> Load your model from the last laboratory. Create a new window in SCOUT of type `DONALD`. Add this to the screen and position it such that it does not obstruct your other jugs.

Before entering DoNaLD definitions, you must indicate which SCOUT window should be used for line drawing. The SCOUT window above is defined with the attribute `pict: "DRAWING1"` and therefore any DoNaLD definitions in the `DRAWING1` viewport will be displayed in that window. To select a viewport in DoNaLD, use the viewport statement:

```
%donald
viewport DRAWING1
```

From now on, as long as you do not change the viewport, all your DoNaLD drawings will be displayed in the SCOUT window you created.

2 Types, points and lines

In DoNaLD, as with SCOUT, observables must be declared before use. DoNaLD supports the following basic datatypes: `int`, `real`, `char`, and `boolean`. Unlike EDEN and SCOUT, there is no semicolon at the end of a statement – instead, the line break is used to separate statements. For example, to define an integer observable:

```
int a        ## first declare the observable
a = 1        ## then assign it a value
```

DoNaLD supports a range of 2D drawing datatypes (see the quick guide), the most basic of which are points and lines. Points can be added together and other geometric functions exist to find, for example, the midpoint of a line:

```
point p, q                 ## declare the points p and q
line l                     ## declare a line
p = { 100, 200 }           ## set point p to (100,200) from the origin
l = [ p, p + { 50, 0 } ]   ## draw a horizontal line of length 50 from p
q = midpoint(l)            ## set point q to be the midpoint of l
```

Note: in SCOUT, the origin (0,0) lies in the top-left corner of the window, but in DoNaLD the origin lies in the bottom-left corner.

> **Task 2**
>
> Draw a single jug using DoNaLD. Create an observable that represents the height of the jug. Check that you can change the height of your new jug.

3 Using dependency and EDEN

DoNaLD and SCOUT are similar in that they both are purely dependency-based notations, and therefore they both use the equals (`=`) for dependency. However, they have different methods for accessing EDEN observables. You do not need to declare EDEN observables in DoNaLD before using them. To use an EDEN observable, you simply place an exclamation mark after the observable name. For example, if you have the observable `capA` in EDEN, then it can be referenced as `capA!` in DoNaLD.

> **Task 3**
>
> Draw a line for the height of the liquid. Using the observables contentA and capA, make the jug height and liquid height consistent with the underlying model. Use your buttons to check that this new jug A shows the same representation as your SCOUT jug A.

4 Shapes

An important type in DoNaLD is the `shape` (or `openshape`) which is a collection of objects (points, lines, shapes, etc). There are two types of shape, called `shape` and `openshape`. The `openshape` is used to define the components of a shape (e.g. points, lines, and other shapes), whereas the type `shape` is used for the transformation of shapes. You cannot change the components of a `shape`, you can only transform it. An example of an `openshape`:

```
openshape cross
within cross {
   line l1, l2
   l1 = [{10, -10}, {-10, 10}]
   l2 = [{10, 10}, {-10, -10}]
}
```

If you need to refer to an observable inside the openshape then you need to reference it by its location and name. In the above example, you would refer to `l1` outside the shape as `cross/l1` (see the quick guide for more information on referencing observables in DoNaLD).

Task 4

Create a new openshape called unitjug and within this create three lines, each of length 40, to represent the sides and bottom of the jug. The bottom-left corner of the jug should be at the origin.

5 Transforming shapes

If you have defined a shape (or openshape) then you can perform transformations to get another shape:

- `trans(shape, x, y)` – translate a `shape` by `(x,y)`
- `scale(shape, factor)` – scale a `shape` by a `factor` from the origin
- `scalexy(shape, factorx, factory)` – scale a `shape` by `factorx` in the x-direction and by `factory` in the y-direction
- `rot(shape, point, angle)` – rotate a `shape` round a `point` by an `angle` (in radians)

For example, to create a new shape which is the `cross` rotated by an angle of 45 degrees:

```
shape cross2                              ## declare the shape
cross2 = rot(cross, {0,0}, pi*0.25)       ## define cross2 as a rotation of cross
```

Task 5.1

Create a new shape called `jugA`. Define `jugA` as a transformation of unitjug that expands in the x-direction by a factor of `widthA!` and in the y-direction by a factor of `capA!`. Create another shape for `jugB`.

If you explore the underlying EDEN model then you will find an observable called `action` which has the value of 5 when pouring, and an observable `pourdirection` which tells you which direction the jugs are pouring (`ATOB` or `BTOA`).

Task 5.2

Create an observable called `angleA` (of type real) which represents the angle of the jug. Define `angleA` to be dependent on `action`, `pourdirection`, `contentA`, and `capA` such that when `jugA` is pouring to `jugB`, `jugA` is rotated at an angle to give the impression of pouring into `jugB`. Alter your definition for `jugA` so that the transform also incorporates a rotation of `angleA`. Perform the same operation for `jugB`.

6 Completing your jugs animation

The final part of this laboratory is an open exercise in elaborating your jugs model. You should familarise yourself with attributes in DoNaLD, so that you can change the line width, colour, and other attributes to make your model look more realistic.

Task 6

Reintroduce the line representing the amount of liquid in the jug. Add dependencies such that when one of the jugs is pouring, the water level remains horizontal and gives the impression of liquid being poured.

7 Saving scripts

It is a good idea to save your model and make sure it is documented – this will be necessary in the coursework. Your jugs model may be relatively small, and so you can save all the definitions in one file. Ensure that groups of DoNaLD definitions start with `%donald` and likewise for other notations.

As your models get larger, you might consider saving different parts of the model in different files. For example, in `jugsBeynon1988` there is the underlying model in jugs.e, the text interface in `jugs_disp.e`, and a SCOUT interface in `jugs.s`. The usual naming convention is to place the EDEN definitions in a file with the extension `.e`, DoNaLD with the extension `.d`, and SCOUT with the extension `.s`. Finally, a script called `Run.e` is usually created to include all the relevant scripts in your model.

Summary

In this lab session you should have learnt how to:

- create a SCOUT window to display DoNaLD drawings.
- write DoNaLD definitions consisting of points, lines and shapes.
- perform shape transformations.
- organise your model by saving definitions in scripts.

This completes the introduction to the `tkeden` environment and the standard notations. You should now be ready to start building your own models using `tkeden`.

Revision history

This worksheet was written by Antony Harfield for CS405 Introduction to EM.