G53DOC Coursework: Flow Layout

Steven R. Bagley

Introduction

For this coursework, you are going to write a Java program that can layout a series of document components on a page. We have not covered this in lectures directly, although we have covered the machinery you need to implement it, but the technique is very simple and the easiest way to understand it is to implement it. In this coursework, you will implement one of the simplest layout mechanisms, the *flow*.

The flow takes a series of document components and lays them out one after another, either vertically down the page, or horizontally across the page. When laying the components out, a flow considers each component, regardless of its shape, to be a rectangle with a fixed width and height. The width or height can then be used iteratively calculate the position of the other shapes in the flow. The actual content of each component can then be drawn with its *bottom-left* corner at the position calculated. Figure 1 shows a series of three shapes in a horizontal flow.

Your coursework will be written in Java, and will support two types of content: the first are Encapsulated PostScript (EPS) files, the second is raw text, which will need to be typeset by using the Line Breaking algorithm shown in lectures (you will need to modify this code to fit in with the model used by the coursework). Your program will use Java objects to define the flow and generate PostScript on the standard output that when viewed displays the finished page. The marks for each question are given in the right margin, the total will be divided by four to give a mark out of 25 used for your coursework component.

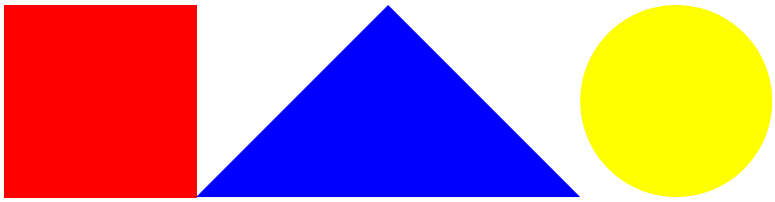

Figure 1 — Three shapes (`square.eps`, `bigtriangle.eps` and `circle.eps`) in a horizontal flow.

Flowable objects

The model we will use in the coursework is that all content is represented by a Java object that implements the `Flowable` interface:

```
public interface Flowable
{
        public void drawAt(double x, double y);
        public double height();
        public double width();
}
```

The methods for this interface are simple: `width()` and `height()` return the width and height of the object respectively. `drawAt()` simple emits the PostScript to cause the object to be drawn at the specified x,y-co-ordinate. So if we implemented an object that drew rectangles, `drawAt()` would emit instructions to draw a rectangle from x,y that is `width()` wide and `height()` high.

However, the `drawAt()` routine needs to ensure that both the graphics state and the overall state of the PostScript VM are left in a known state, otherwise further drawing will happen in random locations. So for our rectangle example above, we could ensure this by surrounding the rectangle drawing with a `gsave` and a `grestore`. It gets slightly more complicated when handling external PostScript fragments, because it is necessary to stop them from resetting the graphics state or issueing a `showpage`.

Interestingly, we'll also make our flows implement `Flowable`, which means we'll be able to nest flows inside each other to create more complex layouts. You will need to bear this in mind when outputting the PostScript to generate the flow (i.e. enusre the graphics state is preserved).

Question 1 — Getting Started: Encapsulation

The easiest place to get started with this coursework is by implementing the document component that can display an EPS file at any point on screen. This should be implemented as a Java object that implements `Flowable`, which takes the path to a PostScript file when the object is created.

At its heart, all this object needs to do is copy the contents of the specified PostScript file to the standard output when `drawAt()` is called. However, in practice, to make all the other sections of the code work a little more implementation is required. This can be broken down into working out how big the PostScript is, and ensuring that the PostScript within the file does not effect any PostScript that comes after it.

Finding out how big an arbitary piece of PostScript is a complex task but fortunately the specfication for Encapsulated PostScript makes it very easy. An EPS file must have a comment near the beginning of the PostScript file that defines the bounding box, in a defined format. The following extract shows the beginning of the `tiger.eps` file, with the releveant line enbolden.

```
%!PS-Adobe-3.0
%%Title: flattened PostScript generated from file: tiger-original.eps
%%Creator: pstoedit
%%BoundingBox: 17 166 565 752
%%Pages: (atend)
%%EndComments
%%BeginProlog
```

The `%%BoundingBox` is always of this form:

```
%%BoundingBox: llx lly urx ury
```

Where *llx*, and *lly* represent the bottom-left of the visible material, and *urx*, and *ury* represent the top-right. Given these, it is possible to calculate the width and height of the piece, and also note where it starts on the page (*llx* and *lly* may not be zero).

Using the co-ordinates of the lower-left corner and the position provided to `drawAt()` you should be able to implement `drawAt()` to emit the necessary PostScript to cause the EPS file to be drawn at the specified location before copying the contents of the EPS file to the standard output. However it is necessary to ensure that the EPS file leaves the PostScript interpreter in a known state. Previously, we've use `gsave`/`grestore` pairs to preserve the graphics state but we need to ensure that the EPS file does not alter any variables, etc. we might be using. To do this, we can use the `save` operator which packages up the state of the interpreter into an object which is saved on the stack. This can then be stored in a variable (with a name that is unlikely to be used by the EPS file itself) and after the EPS file has executed give the saved state to the `restore` operator as a parameter. Finally, there are one or two operators (`showpage` and `initgraphics`) which can cause problems so you probably want to re`def`ine these to have no effect.

Once you've implemented this object, you should be able to test it by calling `drawAt()` with some of the sample EPS files to draw them at various points on the page, and check everything is working. (40)

Question 2 — Flows

Once you have the encapsulation mechanism defined in question one implemented, it is relatively easy to write classes that implement both a horizontal flows and vertical flows. The description below defines how a horizontal flow is implemented, the vertical flow is identical but goes down the page instead of across it.

You will need to implement a class that extends `Flowable` to represent a flow. The constructor for this flow takes a single integer as a parameter which defines how the components are aligned vertically within the flow. Values of 0, 1 and 2 mean the pieces are aligned along the bottom, middle or top respectively (for vertical flows, left-aligned, centred or right-aligned).

You will also need to implement a method that can be used to add other `Flowable` objects into the flow — these are added in order and there's no need to support removing objects from the flows.

To implement `drawAt()`, you just need to iterate over the objects that have been added to the flow, one by one and call their `drawAt()` method. The position each component should be drawn at can be calculated relatively simply. The x-co-ordinate for the first component is just the x-co-ordinate passed into the flow's `drawAt()`. The x-co-ordinate for the second component then is just the x-co-ordinate parameter passed in *plus* the width of the first co-ordinate. For the third component, you also need to add on the width of the second component to the x-co-ordinate and so on. From this definition, it should be possible to see that the width of a horizontal flow is just the sum of the widths of its components.

The y-co-ordinate can also be calculated easily, for a bottom-aligned flow it is always the value of the y-parameter. For other alignments, you will need to add on the correct amount to align the values.

Once implemented, you should be able to test relatively easily by creating a flow and adding `Flowable` versions of the EPS files using the class you created in Question one. With the horizontal flow, working you should be able to adapt your code to create another class that implements a vertical flow (although watch out for the vertical axis direction!). Putting a vertical flow inside a horizontal flow (or vice versa) should give you a good idea of whether your code is stable or not. (40)

Question 3 — Text blocks

Finally, it should be relatively easy for you to adapt the LineBreaking algorithm code to implement the `Flowable` interface, which would allow you to mix blocks of text in with EPS files. You will need to specify how wide the text is when creating the object as well as giving it a text file containing the text to typeset. (20)