G5BDOC — PostScript® Examples 2001

Steven Bagley

School of Computer Science and IT
Jubilee Campus
University of Nottingham
NOTTINGHAM
NG8 1BB

1. Introduction

This handout contains a few examples, which show the use of PostScript® programming in the real world. It may even be of use in your G5BDOC coursework…

Before we continue, a small hint — when programming PostScript®, the stack can get very large, so it can be helpful to write down what you expect to see on the stack as a comment (in PostScript® these are proceeded by a % character) in your source code.

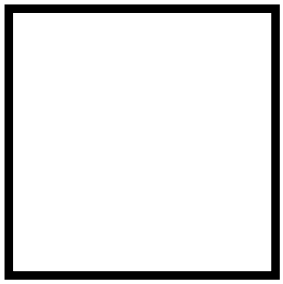
2. The Boxes Example

This example reads draws a series of boxes stacked side by side of a variable width. One use for this might be to draw stacked bar charts.

Our starting point is a routine that can draw a simple box, such as the following code:

```
%!PS-Adobe-2.0
%%BoundingBox: 300 300 400 400

0 setgray

newpath
300 300 moveto
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath
stroke

showpage
```

The above example draws a box like this:



Exciting isn't it… Breaking down the code line by line, we can see how it works. Starting with the line:

```
0 setgray
```

simple sets the current drawing colour (or g**r**ey level) to black. The next line tells the PostScript® interpreter that it is about to start a new path. The program then moves to the bottom-left corner of the box and then we start drawing the lines using the `rlineto` command which draws a line relative to the current position.

First drawing the left-hand side of the box, then the line across the top and then the line that forms the right-hand side. Note that we don't have to draw in the the bottom line as PostScript® automatically completes paths when they are closed. So we use

```
closepath
stroke
```

to close the path and draw it.

2.1. Procedures again...

While the code above works, it will only draw the box at a specific point. What would be more useful is a routine (or procedure) that would draw a box anywhere on the page.

Before setting off to write the procedure it makes sense to step back and work out the order of the parameters on the stack as the procedure is called. By careful planning of parameter order, it is possible to minimise the use of dictionaries within a PostScript® program, and also to cut down on the number of `dup`s and `roll`s needed.

Our original programs first operation was to move to the bottom-left corner of the box. Also, these values were never used in the program again, so if we pass these on the the stack first we can forget about them after out first `moveto`

The next thing it does is to move up, so it'd make sense to place the height next on the stack. The final parameter required is the width of the box, so this is placed last.

Our final stack call would then look something like this:

```
width height x y box
```

where `box` is the procedure name, the rest of the arguments are obvious.

So now we have to code the procedure, which will look something like this:

```
/box
{
newpath
moveto
dup
0 exch rlineto
exch
0 rlineto
0 exch -1 mul rlineto
closepath
stroke
} def
```

The `newpath` command again starts a new path, and we use `moveto` to move to the bottom-left point of the box. A copy of the height (now at the head of the stack) is made using `dup`, as this is needed later when we draw the other side of the box. We then draw a vertical line up using `rlineto`, note the use of `exch` to flip the horizontal and vertical values around.

The stack is then manipulated again to get the width parameter into the right place (as the top of the stack is still the height — thanks to the `dup` earlier). A simple `exch` suffices, in this case — however in more complex situations, it becomes necessary to rotate the stack using `roll`)

The right-hand line is now drawn, using the saved value of `height`. As the line is being drawn downwards, `height` is inverted by multiplying it by -1.

To draw the same box as before, the code now looks like:

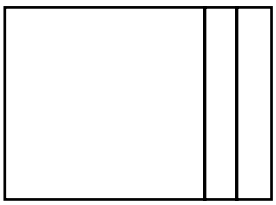
```
%!PS-Adobe-2.0
%%BoundingBox: 300 300 400 400

/box
{
newpath
moveto
dup
0 exch rlineto
exch
0 rlineto
0 exch -1 mul rlineto
closepath
stroke
} def

0 setgray

100 100 300 300 box

showpage
```

2.2. Multiple Boxes

Our `box` procedure draws a box for us at any point and at any width and height. So a bit of PostScript® like this:

```
75 72 300 300 box 12 72 375 300 box 13 72 387 300 box
```

can be used to draw a stacked bar chart like this



The problem with this method is that we have to manually do all the maths to work out where to draw each box. As computers are good at maths, it would make sense to let it work it all out.

As before, the first thing we do is work out the parameters for the procedure. It's going to need the *xy* coordinates of the bottom-left, and the height of the boxes. The only other

thing we need is the widths of each section.

There are two ways, this can be done. The simplest is to load all the values on the stack, along with the number of values, and just decrement the counter each time we draw a box. The procedure would then be called thus::

```
(number)* counter height x y stackedbar
```

where `stackedbar` is the name of the procedure, and `(number)*` means any number of numbers can be placed here

PostScript® provides us with conditional execution statements such as `if` and the ability to create loops as found in more traditional programming languages, such as C.

Storing away the values of `x`, `y`, and the `counter` in a dictionary for later use. Our stack then contains the list of widths.

The PostScript® command `loop` will execute a block of code indefinately, until `exit` is called to break out of the loop.

The format of this command is:

```
{ … } loop
```

The code that we wish to be looped, needs to start with some form of conditional, so we can break out when we've drawn all the boxes. One bit of code, we could use is the following:

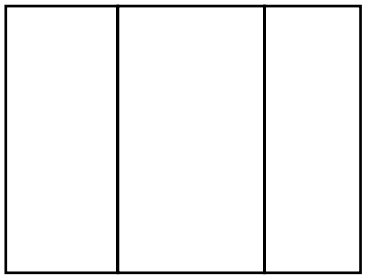
```
counter 1 sub dup /counter exch def -1 eq { exit } if
```

The `if` command in PostScript® is interesting in that it expects a boolean and an executable array on the stack (in that order). If the boolean is true, it executes the code in the array (in this case, the `exit` command to escape the loop). PostScript® also provides us with several functions which will compare two values against each other and return a boolean value (such as `eq` for equality, `ne` for inequality, `lt` for less than, `gt` for greater than, etc) based on the outcome of the test. By combining these commands (as above), it's possible to produce useful conditional statements. It is left to an exercise for the reader to work out why we use -1 and 0 to test against…

The final `stackedBar` procedure looks like this:

```
%!PS-Adobe-2.0
%%BoundingBox: 300 300 433 400

/stackedBar
{
 50 dict begin

       /y exch def
       /x exch def
  /height exch def
 /counter exch def

% STACK: null (int)*

  {
    counter 1 sub dup /counter exch def -1 eq { exit } if

    newpath
    x y moveto
    0 height rlineto
    dup 0 rlineto
    0 height -1 mul rlineto
    closepath stroke

    x add /x exch def
  } loop

 end
} def

0 setgray

36 55 42 3 100 300 300 stackedBar

showpage
```

and produces the following output:



2.3. The alternative

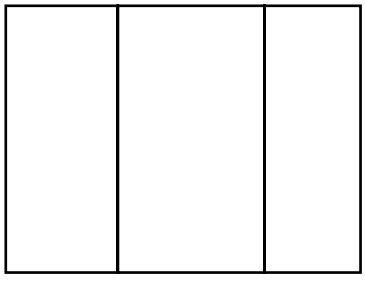
Instead of putting a counter and all the values on the stack, we could push them all into an array, making our function call look like this:

```
[(number *)] height x y stackedBar
```

It turns out that this actually makes our procedure easier to code, thanks to PostScript®'s `forall` command. This takes an array and an executable array, and calls the executable array for every value in the array, this makes our procedure much simpler, boiling down to the following:

```
%!PS-Adobe-2.0
%%BoundingBox: 300 300 433 400

/stackedBar
{
 50 dict begin

 /y exch def
 /x exch def
 /height exch def

% STACK: null (int)*

  {
    newpath
    x y moveto
    0 height rlineto
    dup 0 rlineto
    0 height -1 mul rlineto
    closepath stroke

    x add /x exch def

  } forall

 end
} def

0 setgray

null [42 55 36] 100 300 300 stackedBar

showpage
```

producing the following output (note how the order of the widths has to be reversed to get the same effect.



3. Useful commands...

One PostScript® command you may find useful is `stringwidth` which returns the width and height of a string in the current font and pointsize. Its details are:

stringwidth

string `stringwidth` W_x W_y

calculates the change in the current point that would occur if string were given as the operand to `show` with the current font. W_x and W_y are x and y dimensions describing the width of the glyphs for the entire string in user space.

To obtain the glyph widths, `stringwidth` may execute the descriptions of one or more of the glyphs in the current font and may cause the results to be placed in the font cache. However, `stringwidth` prevents the graphics operators that are executed from painting anything onto the current page. Note that the width returned by stringwidth is defined as movement of the current point. It has nothing to do with the dimensions of the glyph outlines.