G5BDOC — POSTSCRIPT EXAMPLES

1. Background

This handout gives an example of a piece of PostScript which adds a 'drop shadow' to a text string in a given font. We then go on to make this into a PostScript routine, called `shadow`, complete with arguments for the position of the string, granularity of the shading, pointsize, font etc.

2. The *CPU* example

The following piece of code will print the word CPU in drop-shadowed Avant-Garde

```
%!PS-Adobe-2.0
%%BoundingBox: 312 370 384 442

/AvantGarde-Book findfont 20 scalefont setfont

/printCPU
{0 0 moveto (CPU) show} def

320 400 translate

.95 -.05 0  %start incr end
{setgray printCPU -1 .5 translate} for

1 setgray printCPU

stroke
showpage
```

It begins by establishing 20 point AvantGarde as the current font. The procedure `printCPU` is then defined and the origin of the coordinate system is moved to the desired starting point. The **`for`** loop then begins and the semantics of its execution are as follows. The numbers `.95`, `-.05` and `0` are pushed onto the stack followed by the executable array

```
{setgray printCPU -1 .5 translate}
```

which constitutes the procedure body. The **`for`** operator repeats these operations for each value of the loop counter from `.95` down to `0`. When the loop has terminated the gray value (note that US spelling is essential — PostScript will not recognise 'setgrey' as being a valid operator) is set to white with

```
1 setgray printCPU
```

and the word CPU is printed one last time.

The effect of this code is to produce:

3. Making this into a procedure

The example below shows how the drop shadowing routine can be made into a procedure

```
%!PS-Adobe-2.0
%%BoundingBox: 312 370 484 442
% arguments for shadow are
% xc yc - the placement coordinates for start of the string to be drop shadowed
% startgr - starting gray shade of the shadow
% endgr - ending gray shade of the shadow
% incgr - incrementation of the gray level on every journey round the loop
% p - the pointsize of the text string
% str - the string to be drop shadowed
% ft - font dictionary for the chosen font to be used
% calling sequence is thus
% xc yc startgr endgr incgr p str ft

/shaddict 50 dict def   %dictionary for local variables of shadow
/shadow                 %start of proc defn. for shadow
{shaddict begin         %using pre-allocated shadow dict store the args.
 /ft exch def
 /str exch def
 /p exch def
 /incgr exch def
 /endgr exch def
 /startgr exch def
 /yc exch def
 /xc exch def

/printstr {0 0 moveto str show} def

gsave
        ft findfont p scalefont setfont
        xc yc translate


        startgr incgr endgr {setgray printstr -1 .5 translate} for

        1 setgray printstr
grestore

end
} def

320 400 .95 0 -.05 20 (CPU) /AvantGarde-Book shadow
400 400 .95 0 -.05 40 (Gorilla) /Helvetica shadow

showpage
```

Figure 1: PostScript *shadow* procedure for drop-shadowed text

4. Explanation of shadow routine

The comments included within the above PostScript programme describe the incoming arguments to the `shadow` routine. Remember that these arguments will be present, on top of the stack, when the routine is entered, in an order that is the *reverse* of that in which they are listed in the main program. Using a pre-declared dictionary, `shaddict`, of adequate size, the first task of the procedure is to extract the arguments from the stack, one at a time, and to place them in named locations in `shaddict`. When allocating a value to a dictionary name the syntax is normally:

```
/fred 20 def
```

where the name is the first value on the stack and the value comes last. However, for the incoming actual parameters to a procedure, it is the *values* that are already present on the stack. Therefore, in every case, we need to push the desired formal parameter name onto the stack and then do an `exch` operation to get name and value the right way round for the subsequent `def` operation.

The procedure `printstr` is declared locally to `shadow`. Since it is a local declaration and *not* a formal parameter it follows that name and value will be pushed on the stack in the correct order and an `exch` is not needed. Note carefully the need for declarations to precede usage: the body of `printstr` uses the incoming string `str`, which must have been popped off the stack and defined in a dictionary *before* the `printstr` definition.

The main body of the procedure behaves in essentially the same way as the initial CPU example in section 1, except that the pointsize, positioning, string value and **`for`** loop parameters are now arguments to the procedure. Note carefully that one of the values passed over to `shadow` is the name of the font to be used. The fact that this begins with a forward slash, as in `/AvantGarde-Book` means that this parameter is passed over in the form of a stack address which points to the beginning of the information for that font in the font dictionary.

Finally, notice that the entire procedure body is bracketed between a `gsave` and a `grestore` which save and restore the graphic state. This is because each text string translates the co-ordinate origin and successive calls of `shadow` would incrementally move the origin *relative to where it was left by the previous call* in the most alarming way. The placement of `gsave` and `grestore` saves the initial (0,0) co-ordinate origin and restores it after every call.

We can now show the effect of the two calls of `shadow` that are embedded in the above example by including the PostScript into this present *troff* document using *psfig*. For this to work the top two lines:

```
%!PS-Adobe-2.0
%%BoundingBox: 312 370 484 442
```

must be present. The arguments for the bounding box of a diagram are the *x* and *y* coordinates (in points) of its lower left and upper right corners. The final printed effect of the above example is: