Mathematics 308—Fall 1996

Happier PostScript programming hints—free advice at half-time

This year people aren't having as much trouble as in previous years. I would like to think this is because I am getting more experienced at heading off trouble, but it is more likely a random occurrence. Neverheless the following remarks may be useful.

• So far repeats and procedures have caused some trouble. They are a bit tricky. So the best thing to do first in an assignment is get a picture—any picture even distantly related to the assignment—up on the screen. Forget about repeats and procedures in the first attempts. Then you have something to gauge your progress against. You might also save separately those versions which do give you pictures so you don't go 'round in circles.

• Even at this stage, however, it will help you to have variable names for certain constants like the pagewidth or size of margin. Give them readable names like **`pagewidth`** or **`margin`**. The purpose of using variables rather than raw numbers like **`11.15`** etc. is to reduce your mental effort, and readable names help a lot. define these **global variables** right at the beginning of your program, and don't change them again. Another good example of a global variable would be $\pi = 3.1415926536$.

• Do not at first put separate problems together. This slows everything up, and confuse things badly. Put each picture in a separate file at first. Then when each one is working on its own, you can put them all together. The best way to do this is with the UNIX command **`cat`**. Thus

```
cat prob1.ps prob2.ps > hw.ps
```

typed within an **`xterm`** window will copy **`prob1.ps`** and **`prob2.ps`** in that order into the file **`hw.ps`**. If that file already exists, use **`>!`** instead of **`>`**, so it will be overwritten. Then you will have to do a little 'light editing' in **`hw.ps`** (removing initial setup, etc; adding showpages perhaps) to make it ready to hand in. Above all, try out **`gs`** on the final file before sending it along to me to be sure it all fits together. Remember that two lines

```
72 72 scale
72 72 scale
```

are cumulative and will scale to 72×72! **`gsave`** and **`grestore`** are ways around this.

• Using **`xterm`** and **`gs`** is messy unless you remember my advice to begin by launching two copies of **`xterm`** by typing

```
xterm -sb -sl 500 &
```

in your login window when you first start up. I remind you that you can adjust font sizes in your terminal windows with **`ctl-shift`**, I believe.

Use one terminal window to run **`gs`** in and the other to do Other Useful Stuff like **`cat`** or **`ls`** or **`mail`**. Perhaps I haven't mentioned the **`-sl 500`** before—it remembers the last **`500`** lines so you can scroll back to them. The **`&`** is very important!

• If you feel confident, just jump in and start drawing. But when things go wrong, keep in mind that the process is hypnotic. Take time out for a cup of coffee or whatever and sketch your PostScript in a notebook before you go back to programming. Take advantage of the fact that YUM-YUM's is right next door! (Too bad it closes so early.)

• Use variables only when it seems really useful. And as I mentioned above, give them good names.

• To find out what **`x`** is equal to at some point in your program insert code

`(x =) == x ==`

This looks cryptic, but what it says is to display the string "x =" then the current value of **`x`**. For serious stuff insert **`pstack`**.

• Do not confuse the **name** of a variable **`/b`** with its **value** **`b`**. Early in the course, the only time so far in this course that you should use the name **`/b`** is when you assign a value to it. But now with **`mkpath`** you are going to use the names of functions to draw.

• Make lots of comments. *More than I do!* In PostScript you should do this to keep track of what's on the stack.

• Keep your stack clean. A number of the homeworks finish up with tons of unused garbage sitting there, indicated by prompts like **`GS<397>`**. Be sure that at the end of procedures the stack has on it just what you want it to and no more extra junk. Especially true of **`for`**, where you have to **`pop`** the loop variable if you don't use it.

• I repeat again that drawing in PostScript is like drawing by hand. **`newpath`** amounts to picking up a pencil. **`moveto`** amounts to moving it. **`lineto`** amounts to drawing a straight line from the current location of the pencil to the next one. But the thing you have drawn is not visible until the end, when you apply

```
stroke
fill
clip
```

Keep in mind that the last two are different from the first in that the path you draw must be *continuous* and *closed*, or stuff will 'leak out'.

• How the path is actually drawn depends on the **graphics state** only at the moment you finish it. The graphics state takes into account (•) the current transformation; (•) the current colour; (•) current dash pattern; (•) current linewidth; and a few things you won't have to know about. *If you want to draw paths which are different in any of these aspects then you must finish one path before taking up another.* Like putting down your current pencil and picking up another one. In particular it does not make much sense to change any of these things in the middle of drawing a path, between the first **`newpath`** and the last **`stroke`**. It only confuses things. Make all changes of this kind *before* you put down a newpath.

• Putting **`gsave`** and **`grestore`** around a path construction will wipe out the path you build. So if you want to draw it, do not put these commands between **`newpath`** and the path, or between the path and **`stroke`**.

• Keep the lines which actually draw your paths as uncluttered as you can. It is hard enough to draw stuff, trying to keep track of where your pencil point is, so to speak, without worrying about how variables are changing; and unnecessary if those variables do not *have* to chnage while you are drawing. Path drawing is the cock-pit of PostScript programming. Take your shoes off when you enter the drawing room.

• You have seen essentially all of the PostScript you will need.

• Programming is always potential chaos, since computers are so fast and yet so stupid. Anything you can do to make your programs simpler to understand and easier to read is always a good idea. Especially in this course where, if you are careful, like any good artist you will be able to use old sketches in new work.

• You can move stuff from your home machines to the system in the lab by using the ancient IBM clone in the corner. To transfer **`myfile`** from your disk to your home directory, put the disk in the piece of junk and then:

```
cd \tmp
copy b:myfile .
ftp gamba
```

log into **`gamba`** with your id and password, and then

```
put myfile
```

To retrieve the stuff from the lab, do things in the opposite order, and use **get** instead of **put**.

```
cd tmp
ftp gamba
(login)
get myfile
copy myfile b:
```

• David Maxwell (bless his heart) has put up a large help system for PostScript on the undergraduate lab network. A link to it can be found on the undergraduate lab home page

http://gamba.math.ubc.ca/localdoc

This is in addition to online copies of my notes available on the course home page

http://www.math.ubc.ca/~cass/courses

This help system is in **html** format, and is available on a single floppy disk for use on a PC at home, to be read with the help of any net browser.

• I repeat again more strongly one thing I have said a number of times. In this course and *when solving any kind of difficult problem, it is always a good idea to break the problem up into small chunks and solve them individually*. In this course this means essentially specific thing: *Use separate, isolated procedures as often as possible, procedures which should do something relatively simple, and which can be debugged simply*. You should do this just about whenever you have to do a job which is simple and clearly defined. In a later assignment involving drawing on spheres, for example, it will be a good idea to have procedures to calculate arc $\cos(x)$ for a number between -1 and 1, to calculate the angle between two vectors, and replace an array of spherical coordinates (ρ, φ, θ) by an array of equivalent rectangular coordinates (x, y, z).

• What I am saying is good advice for all problem solving: **Isolate difficulties!** Break a large problem into bite-sized pieces instead of trying to swallow it whole. Assemble smaller items to buid a large object . . .

• The idea of isolation applies in other ways. *Different parts of your programs should do different things.* For example, I saw yesterday where a student with otherwise almost impeccable style had included inside a procedure a **showpage** operation. This is almost certainly a mistake—suppose that this procedure were to be required several times on one page? Separate function as much as possible to your programs **flexible**.

• Summary of how to use `mkpath`

The array listed first contains a list of things necessary to specify your path from a whole family of paths. Next comes the name of the fundction which parametrizes your path. This function has a well defined interface: it assumes two things on the stack when it is called, the array **[...]** *passed to* **mkpath** *and the parameter* t *which varies along the path.* (Keep in mind: there are two uses of the word 'parameter' here: one for the things which determine which path in a family you are drawing, and one for the thing which varies to specify the path.)