APPENDIX 2

Setting up your PostScript environment

In order to run PostScript programs, you will need to have a PostScript viewer installed on your machine. The most convenient way to do this is to install the basic PostScript interpreter Ghostscript, and then on top of that one of several possible interactive viewers that call on Ghostscript for basic graphics rendering. The program Ghostscript is available without cost for download from `http://www.cs.wisc.edu/~ghost/`. The viewers `GhostView`, `GSView`, `MacGSView`, and `GV` (for various platforms) can also be found there.

On UNIX and Macs the command line interface for the interpreter Ghostscript (as opposed to a file viewer) should be straightforward to figure out, but for Windows machines it is a little more difficult. First run (i.e. **Run**) the program `cmd.exe`, and then in the terminal window that pops up type `gswin32c.exe` togther with various options to get Ghostscript on its own. One variant that you can use fruitfully for debugging is

```
gswin32c.exe -dNODISPLAY <filename>
```

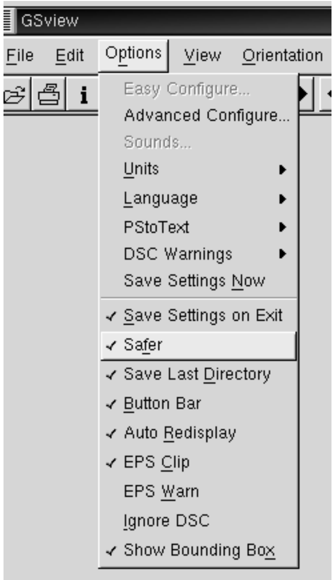
You should be able to set shortcuts up so that not so much typing is involved.

1. Editing PostScript files

It is important to use the right text editor in writing PostScript programs, or at least to know how to use correctly the one that you do use. First of all, a PostScript file must be just an ordinary text file, without formatting adornments such as those produced by Microsoft **Word** in its default configuration. So you must be careful, if necessary, to save your file as a plain text file. In some text editors, text files will be automatically given an extension `.txt`. This is not necessarily a problem, but for your own long-term sanity it is probably best to store all your PostScript files with an extension `.ps` (or a variation like `.eps`). This may require that you explicitly rename it.

2. Running external files

Once you have installed Ghostscript and a viewer, you will have to do a small amount of work to configure your environment for easy PostScript program development. I have described in this book a number of packages of PostScript procedures that you will want to incorporate in your own programs with the PostScript `run` command. This command simply loads a file that it interprets as any other sequence of PostScript code. But for security reasons, the way most PS viewers are configured by default is to disallow this. In order to allow it you must toggle one of the viewer options, usually associated with the keyword **Safer**. In `GSView`, for example, you can do this by opening the **Options** menu (here showing **Safer** toggled on):



If you do this, and you use your viewer to look at PostScript files on the Internet from within your browser, you should be sure to make the **Safer** option still in force inside the browser. How to do this depends on which browser and which operating system you are using. With my viewer `gv` inside Netscape, I set the application that reads PostScript documents to be `gv -safer %s`.

Another remark about the `run` command: recent versions of Ghostscript (8.0 and after) handle some files (those with an `EPSF` in the first line) to be run differently from others. In effect, alas, the default Ghostscript is no longer a strict PostScript interpreter, since it reads comments. This dubious 'feature' should probably be normally disabled by specifying the option `gs -dNOEPS` in your viewer.

3. Making images

At some point you will probably want to make image files from PostScript programs, for example `.jpg` or `.gif` images that you can include directly inside a Web page. There are several ways to do this, one being to use Ghostscript itself to do the job, or your Ghostscript viewer. With `GSView`, for example, you can see what your options are if you look at the menu item `File/Convert`. But you will have a bit more control over the output if you use a full-featured image manipulation program. The most comprehensive of these is *PhotoShop*, but it is expensive and complicated. Another possibility is the program **GIMP** (**G**nu **I**mage **M**anipulation **P**rogram), available without cost for many platforms.

- *In order to use GIMP on Windows machines to interpret PostScript programs, it must be able to locate the Ghostscript executable file. The simplest way to do this is to copy the file* `gswin32c.exe`*, which is put somewhere on your computer as part of the installation of gs, into the directory* `C:\Windows`*.*

One thing to be careful about when importing a PostScript file into an image manipulation program is that many of them require that the file be terminated with `showpage`. Another is that all files which are imported into your file with a `run` command must normally be included in the file explicitly. I'll discuss this problem in the next section.

When you import a PostScript file into an image manipulation program, you will likely have some choices to make about the size and quality of the import. The most subtle choice to make is about **anti-aliasing**. A PostScript file is normally scalable, which means it has no intrinsic granularity. But an image manipulation program will turn this into a bit map, an image with a particular resolution. In other words, it works with discrete units of colour called pixels. In rendering a scalable picture into a bit map, different algorithms are possible. The highest quality will be obtained if transitions of pixel colours are as smooth as possible, and this is what anti-aliasing accomplishes. The term 'aliasing' here comes from the theory of the Fourier transform, but to explain how it applies here would take up far too much space.

4. Printing files

You can print PostScript files if you have a PostScript driver for your printer. This is no problem on some operating systems, but on Windows machines it is not present in the usual installation. It will be possible once you have Ghostscript installed, however. You can also use your PostScript viewer to print PostScript files with a few mouse clicks.

In any event, there are a few precautions to observe. First of all, most printers will not print a page unless it is terminated by `showpage`. Second, the printer will not be able to load files referred to with a `run` command, so you must actually include such files inside your program before printing them. (This is usually true of files imported into an image manipulation program, also.) Your text editor almost certainly allows file inclusion, but this is a relatively slow process, and awkward. Awkward because you will usually be much happier doing your own editing in a file without the package included. Once you include the file explicitly, then, your own program development will be somewhat restricted. I prefer to make the process somewhat automatic, as well as flexible. For this purpose I do the following: (1) I write my original program in a file called, say, `x.px`. It may contain several `run` commands. (2) I apply a `PERL` script I call `psinc` which includes explicitly all the files referred to by `run` commands, and then creates a new file `x.ps` which is self-contained. Furthermore, it includes files recursively, which means that even the files that are included may themselves run other files. I work with a Linux system, but this sort of thing should be possible on almost any system. Here is my `PERL` script:

```
#!/usr/bin/perl

# This reads in ps files, printing out all lines except
# ^( ...  )  run
# where it performs an inclusion --- recursively.
# The current directory is updated.
use File::Basename;
include("", STDIN, 0, "stdin");

# insert a file into output
sub include {
    local($curdir, $input, $depth, $cf) = }@_;
    $i = $depth;
    while ($i > 0) {
        print STDERR "  ";
        $i--;
    }
    print STDERR "Opening $cf\n";
    $fh++;
    while ($_ = <$input>) {
        if (/^\((.*)\)[ ]*run/) {
            ($name, $dir, $suffix) = fileparse($1, '');
            if ($dir eq './') {
                $dir = "";
            }
            $file = "$curdir$dir$name$suffix";
            # print STDERR "file to open = {$file}\n";
            if (open($fh, $file)) {
                print "\n";
                print "% - Inserting $name ----------------------\n\n";
```

```
                include($curdir.$dir, $fh, $depth+1, $file);
            } else {
                print("Current directory $curdir.$dir\n");
                print STDERR "Unable to open $file\n";
                exit 1;
            }
        } else {
            print $_;
        }
    }
    if ($input ne STDIN) {
        $i = $depth;
        while ($i > 0) {
            print STDERR "  ";
            $i--;
        }
        print "\n";
        print "% - closing $name ------------------------\n";
        print STDERR "Closing $cf\n";
        close($input);
    }
}
```

On my Linux system, I have available to me the `make` utility, and turning a `.px` file into a `.ps` file is an option built into my `make` configuration:

```
.SUFFIXES: .px .ps

.px.ps:
    rm -f $*.ps; EPS2eps < $*.px | psinc > $*.ps
```

The whole process therefore becomes quite painless—I just type `make x.ps` when `x.ps` is changed.