Mathematics 308—January 4, 1996

Some simple PostScript examples

Drawing lines

The basic drawing routine in `PostScript` is to draw a line segment between two points. For example, to draw a line between the points $(0, 0)$ and $(1, 1)$ on the plane you write

```
newpath
0 0 moveto
1 1 lineto
stroke
```

What does this do? (1) Every time you want to draw a path of some kind you begin with `newpath`. Next you build your path, for example by constructing line segments with operations like `moveto` and `lineto`. Finally you do the actual drawing with a command like `stroke` which goes over the path you have built and does the actual drawing. (2) The `moveto` command can be thought of as moving your pen to the point $(0, 0)$ without drawing anything. (3) The `lineto` command then adds to the path you are building a line to $(1, 1)$ from the point you moved to. In general, a `lineto` command will add a line segment to the path you are building, starting at the **current point** you are located at (which will be where you last moved to or drew to). Keep in mind here as always that `PostScript` uses reversed notation, so that you write `0 0 moveto` instead of `moveto 0 0`.

For a more complicated path, suppose you want to draw the right triangle with corners at $(0, 0)$, $(1, 0)$, $(0, 1)$. There are many ways to do this, but the natural one in `PostScript` as well as by hand is to trace out the triangle without lifting your pen. This one goes around clockwise.

```
newpath
0 0 moveto
1 0 lineto
0 1 lineto
0 0 lineto
stroke
```

Variables in `PostScript`

`PostScript` allows you to use variables. Assignments are made like this:

```
/x 3 def
```

means that the variable `x` is assigned the value 3. Variable names can be just about anything—for example `silly-variable-name-without-any-significance` is as far as I know perfectly OK. But silly.

Here is an example of how variables can be used in a drawing routine:

```
/a 2 def
/b 3 def
newpath
0 0 moveto
a 0 lineto
0 b lineto
0 0 lineto
stroke
```

will draw a triangle with corners at $(0, 0)$, $(2, 0)$, $(0, 3)$. The point is that it is very simple to draw a different triangle by changing the values of a or b, or to draw several pictures with the same values of a and b.

Transforming a drawing

You can transform a given picture in many different ways: you can rotate it, shift it, and scale it.

If you want to draw the same triangle rotated counter-clockwise by 90° you would insert a single commnad just before the path is drawn:

```
/a 2 def
/b 3 def
90 rotate
newpath
0 0 moveto
a 0 lineto
0 b lineto
0 0 lineto
stroke
```

Incidentally, this has the effect of a permanent rotation by 90°, so it would usually be a good idea to follow it by `-90 rotate` which restores the original drawing scheme.

Along the same lines, you can translate your origin, or scale your drawings in x and y directions.

Scaling and translation are usually necessary in order to see your drawing on the page. This is because

- The origin of the default coordinate system in `PostScript` is at the bottom left of the page
- The default scale is in **points**—there are 72 points to an inch.

The things we have been drawing so far, for example, are extremely small and indeed probably almost invisible unless we scale things up and shift the origin of the coordinate system towards the middle of a page. By the way, the default page in `PostScript` is $8.5'' \times 11''$.

```
72 72 scale
4.25 5.5 translate
```

will set the drawing unit to one inch, and also move the centre of the coordinate system to the centre of the page. Notice that it makes a difference in which order you do the scaling and translation, since the translation takes place in the new units.

A complete example

Here is a complete working program to draw a triangle with corner at the centre of such a page, with sides $2''$ and $3''$.

```
%!

72 72 scale
4.25 5.5 translate

0.025 setlinewidth

newpath
0 0 moveto
3 0 lineto
0 2 lineto
0 0 lineto
stroke

showpage
```

There are a number of minor but new things to notice here.

One odd thing to keep in mind is that on some computers, and nearly all printers, you will get no picture at all without the `%!` starting off your program. This is just a convention you have to take into account.

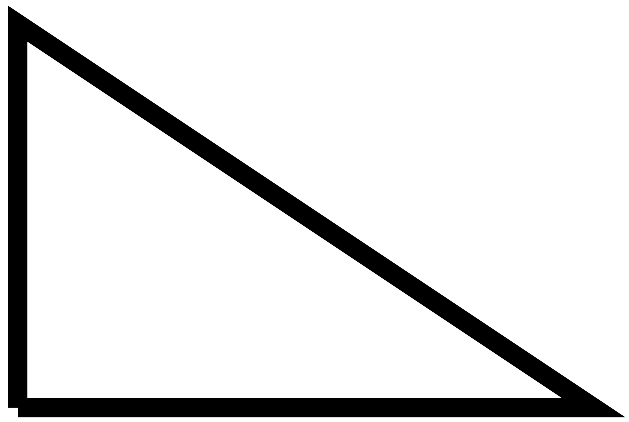
The `showpage` at the end is not usually required to get a picture displayed on the monitor, but it may be required to get printed output. It is good practice always to include it. You can draw on a sequence of pages by drawing a sequence of figures separated by `showpage` commands. We will see later that we can even get some convincing animations in this way. One thing that may cause you trouble is that *the coordinate system must be set up all over again on each page.*

The width of lines is a quantity you can determine inside your program. It seems that normally the line width you will get unless you specify otherwise is 1 unit in whatever units you are working with—one point, one inch, etc. Try drawing the picture above without the `linewidth` specification. A line width of 0 will give you the thinnest line the computer can display.

Some fine points

Here are some other points which you can ignore for the moment if you want, but which will you help you to produce pictures of generally higher quality.

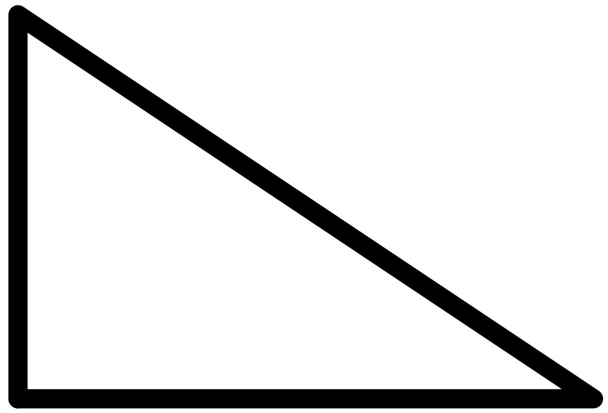
If you draw the same triangle with a bigger line width you will get



This may or may not be what you want. You can change the way the computer draws corners and the ends of lines with commands `setlinecap` and `setlinejoin`. Adding

```
1 setlinejoin
1 setlinecap
```

before the path is drawn will give you



You can draw in any shade of gray with a command like

```
0.8 setgray
```

to get

Note that 0 is black, 1 white. If you are working with a colour monitor or a colour printer you could try something like

```
0 0 1 setrgbcolor
```

Colours are specified by `RGB` components, so this makes the current colour equal to `blue`.

Making solid figures

You can draw paths by applying the `stroke` command, but you can fill closed paths with various colours (including grey) by applying `fill` instead of `stroke`.

```
0.8 setgray

newpath
0 0 moveto
3 0 lineto
0 2 lineto
closepath
```

```
fill
```

will give you

Without applying `closepath` you might have the fill colour leak out onto the whole page. Note that `closepath` draws a line from the current point to the place your path started, so it is not necessary to draw that line explicitly.

You can do an even nicer job by two drawings.

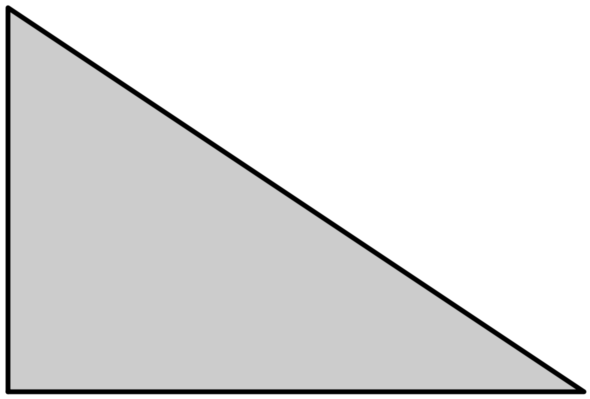
```
0.8 setgray

newpath
0 0 moveto
3 0 lineto
0 2 lineto
closepath
fill

0 setgray

newpath
0 0 moveto
3 0 lineto
0 2 lineto
0 0 lineto
stroke
```

leads to



You have to be careful about the order in which you stroke and fill.

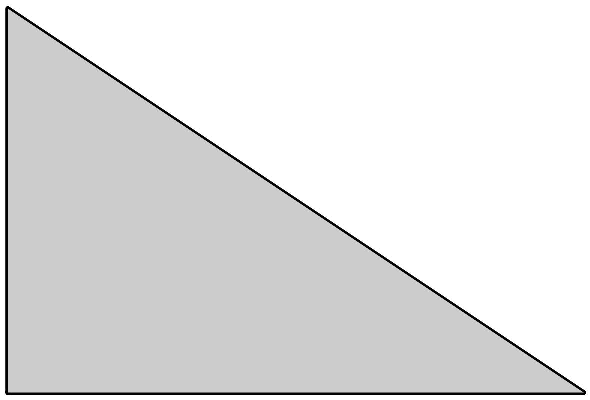
```
0 setgray

newpath
0 0 moveto
3 0 lineto
0 2 lineto
0 0 lineto
stroke

0.8 setgray

newpath
0 0 moveto
3 0 lineto
0 2 lineto
closepath
fill
```

gives you the slightly peculiar



In other words, you should think of drawing in `PostScript` as painting over whatever drawings you have already made on the same page.