Mathematics 308 — Geometry

Chapter 5. Drawing polygons: loops and arrays in PostScript

We begin by learning how to draw regular polygons, and then consider the problem of drawing arbitrary polygons. Both will use loops, and the second will require learning about arrays.

There are three kinds of loop constructs in PostScript.

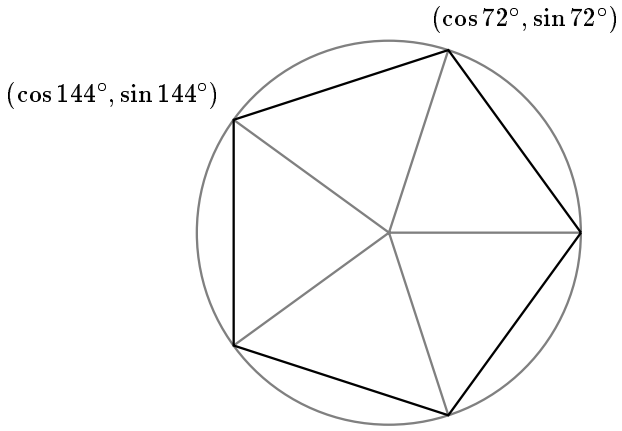
1. 'Repeat'

The simplest loop is the **`repeat`** loop. It works very directly. The basic pattern is:

```
N {
...
} repeat
```

Here N is an integer. Next comes a procedure, followed by the command **`repeat`**. The effect is very simple: the lines of the procedure are repeated N times. Of course these lines can have side effects, so the overall complexity of the loop might not be negligible.

One natural place to use a loop in PostScript is to draw a regular N-sided polygon. This is a polygon with N sides, all of them of the same length, and all with a central symmetry around a single central point. If you are drawing a regular polygon by hand, the simplest thing to do is draw first a circle of the required size, mark N points evenly around this circle, and then connect neighbours. Here we shall assume that the radius is to be 1, and that the location of the N points is fixed by assuming one of them to be $(1, 0)$.



If we set $\theta = 360/N$, then the other points on the circle will be $(\cos\theta, \sin\theta)$, $(\cos 2\theta, \sin 2\theta)$, etc. To draw the regular polygon, we first move to $(1, 0)$ and then add the lines from one vertex to the next. At the n-th stage we must add a line from $(\cos(n-1)\theta, \sin(n-1)\theta)$ to $(\cos n\theta, \sin n\theta)$. How can we make this into a repetitive action? By using a variable to store the current angle, and incrementing it by $360/N$ in each repeat. Here is a procedure which will do the job:

```
% At entrance the number of sides is on the stack
% The effect is to build a regular polygon of N sides

/make-regular-polygon {
4 dict begin
/N exch def
/A 360 N div def

1 0 moveto
N {
  A cos A sin lineto
  /A A 360 N div add def
} repeat
closepath

end
} def
```

In the first iteration, $A = 360/N$, in the second $A = 720/N$, etc.

Exercise 1.1. *Modify this procedure to have two arguments, the first equal to the radius of the polygon. Why is not worthwhile to add the centre and the location of the initial point as arguments?*

2. 'For'

Repeat loops are the simplest in PostScript. Slightly more complicated is the **for** loop. To show how it works, here is an example of drawing a regular pentagon:

```
1 0 moveto
1 1 5 {
  /i exch def
  i 72 mul cos i 72 mul sin lineto
} for
closepath
```

The slightly tricky part answers the question 'What is the line **/i exch def** doing there?' The structure of the **for** loop is this:

```
s h N {
...
} for
```

This loop involves a 'hidden' and nameless variable which starts with a value of s, increments itself by h each time the procedure is performed, and stops after doing the loop where the variable is equal to N. This implicit variable is put on the stack just before each repetition of the procedure. The line **/i exch def** behaves just like the similar lines in procedures—it takes that hidden variable off the stack and assigns it to the named variable i. It is not necessary to do this, but you must do something with that number on the stack, because otherwise it will just accumulate there, causing eventual if not immediate trouble. If you don't need to use the loop variable just want to get rid of it, use the command **pop** which just removes the top item from the stack.

The slightly dangerous thing about **for** loops is that if the sequence s, $s + h$, $s + 2h$, never contains the exact number N, the loop will keep going forever. Therefore: because of the problems caused by rounding errors in computers, you must always use integers in a **for** loop. The following, for example, are not good. The first is guaranteed not to work, while the second might work on some machines and not on others.

```
1 2 4 {
...
} for

0 1 3 div 1 {
...
} for
```

Exercise 2.1. *Make up a procedure* **polygon** *just like the one in the first section, but using a* **for** *loop instead of a* **repeat** *loop.*

Exercise 2.2. *Write a complete PostScript program which makes your own graph paper. There should be light gray lines* 1 *mm. apart, heavier gray ones* 1 *cm apart, and the axes done in black. The centre of the axes should be at the centre of the page. Put in a margin of* 1 *cm. all around the page.*

3. 'Loop'

The third kind of loop is the most complicated, but also the most versatile. It operates somewhat like a **while** loop in other languages, but with a slight extra complication.

```
1 0 moveto
/A 72 def
{ A cos A sin lineto
  /A A 72 add def
  A 360 gt { exit } if
} loop
closepath
```

The complication is that you put in the condition yourself, and explicitly force an exit if it is not satisfied. Thus if you put in your condition at the beginning of the loop, you have the equivalent of a **while** loop, while if at the end a **do ... while** loop. Thus, the commands **loop** and **exit** must be used together.

4. General polygons

Polygons don't have to be regular. In general a polygon is essentially a sequence of points P_1, $P_1, \ldots, P_n$ called its vertices. The polygon is made up of lines connecting the successive neighbours. We shall impose a convention here: a point will be an array of two numbers $[x\ y]$ and a polygon will be an array $[P_1\ P_2\ \ldots\ P_n]$. We now want to define a procedure which has an array like this as a single argument, and builds the polygon from that array by making line segments along its edges.

There are a few things you have to know about arrays in PostScript in order to make this work (and they are just about all you have to know): (1) the numbering of items in an array starts at 0; (2) if a is an array then **a length** returns the number of items in the array; (3) if a is an array then **a i get** puts the i-th item on the stack.

```
/make-polygon {
3 dict begin
/a exch def
/n a length def
n 1 gt {

  a 0 get 0 get
  a 0 get 1 get
  moveto
  1 1 n 1 sub {
    /i exch def
    a i get 0 get
```

```
    a i get 1 get
    lineto
  } for

} if

end
} def
```

This procedure starts out by defining the local variable a to be the array on the stack which is its argument. Then it defines n to be the number of items in a. If $n \leq 1$ there is nothing to be done at all. If $n > 1$, we move to the first point in the array, and then draw $n - 1$ lines. Note that since there are n points in the array, we draw $n - 1$ segments, and the last point is P_{n-1}. Note also that since the i-th item in the array is a point P_i, which is itself an array of two items, we must 'get' its elements to make a line. If $P = [x\ y]$ then **`P 0 get P 1 get`** puts **`x y`** on the stack.

Note also that if we want a closed polygon, we must add **`closepath`** outside the procedure. There is no requirement that the first and last points of the polygon be the same.

There is one final thing to know about arrays. You build one by entering any sequence of items in between square brackets [and], separated by space, possibly on separate lines. An array can be any sequence of items, not necessarily all of the same kind. The following is a legitimate use of **`make-polygon`** to draw a pentagon:

```
newpath
[
[1 0]
[72 cos 72 sin]
[144 cos 144 sin]
[216 cos 216 sin]
[288 cos 288 sin]
]
make-polygon
closepath
stroke
```

Exercise 4.1. *Use loops and* **`make-polygon`** *to draw the American flag in colour, say* $3''$ *high and* $5''$ *inches wide. (The stars—there are* 50 *of them—are the interesting part.)*