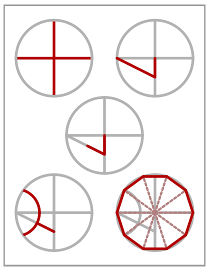
CHAPTER 5

Drawing polygons: loops and arrays



We begin by learning how to draw regular polygons, and then look at arbitrary polygons. Both will use loops, and the second will require learning about arrays.

There are several kinds of loop constructs in PostScript. Three are frequently used.

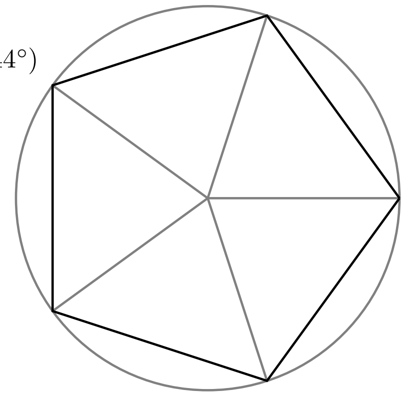
5.1. The repeat loop

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 that has N sides all of the same length, and which also possesses central symmetry around a single 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 neighbors. 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 that 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 5.1. *Modify this procedure to have two arguments, the first equal to the radius of the polygon. Why is not worthwhile to add the center and the location of the initial point as arguments?*

5.2. The for loop

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 `for` loop has one slightly tricky feature which requires the line `/i exch def`. 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 last loop where this variable is equal to or less than N. This hidden (or 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, but just want to get rid of it, use the command pop, which just removes the top item from the stack.

Incidentally, it is safer to use only integer variables in the initial part of a `for` loop, because otherwise rounding errors may cause a last loop to be missed, or an extra one to be done.

Exercise 5.2. *Make up a procedure* `polygon` *just like the one in the first section, but using a* `for` *loop instead of a* `repeat` *loop.*

Exercise 5.3. *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 center of the axes should be at the center of the page. Fill as much of the page as you can with the grid.*

5.3. The loop 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 *must* test a condition in the loop, and explicitly force an exit if it is not satisfied. Otherwise you will loop forever. 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` should almost always be used together. Exits can be put into any loop in order to break out of it under exceptional conditions.

5.4. Graphing functions

Function graphs or parametrized curves can de done easily with simple loops, although we shall see in the next chapter a more sophisticated way to do them. Here would be sample code to draw a graph of $y = x^2$ from -1 to 1:

```
/N 100 def
/x -1 def
/dx 2 N div def

/f {
  dup mul
} def

newpath
x dup f moveto
N {
  /x x dx add def
  x dup f lineto
} repeat
stroke
```

5.5. General polygons

Polygons don't have to be regular. In general a polygon is essentially a sequence of points P_0, P_1, $\ldots$, P_{n-1} called its **vertices**. The edges of the polygon are the line segments connecting the successive vertices. We shall impose a convention here: a point will be an array of two numbers $[x\ y]$ and a polygon will be an array of points $[P_0\ P_2\ \ldots\ P_{n-1}]$. 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;
(4) you create an array on the stack by entering `[`, a few items, then `]`;

```
% argument:  array of points
 % builds the corresponding polygon
/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 $n-1$ line segments. 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.

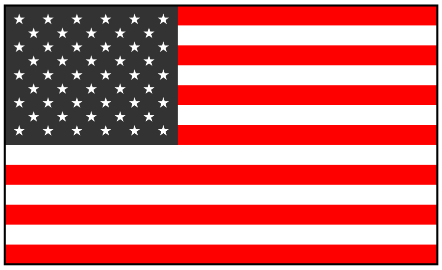
There is another way to unload the items in an array onto the stack: the sequence `P aload` puts all the entries of `P` onto the stack, together with the array `P` itself at the top. The sequence `P aload pop` thus puts all the entries on the stack, in order. This is simpler and more efficient than getting the items one by one.

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 more important thing to know about arrays. Normally, 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 5.4. *Use loops and* `make-polygon` *to draw the American flag in color, say $3''$ high and $5''$ inches wide. (The stars—there are 50 of them—are the interesting part.)*



Exercise 5.5. *Another useful pair of commands involving arrays are* `array` *and* `put`*. The sequence* `n array` *puts on the stack an array of length n. What is in it? A sequence of* `null` *objects, that is to say essentially faceless entities. Of course the array will be of no use until null objects are replaced by proper data. The way to do this is with the* `put` *command. The sequence* `A n x put` *sets $A[n] = x$.*

(1) Construct a procedure `reversed` *that replaces an array by the array that lists the same objects in the opposite order, without changing the original array. (2) Then use* `put` *and careful stack manipulations to do this without using any variables in your procedure. Thus*

```
[0 1 2 3] reversed
```

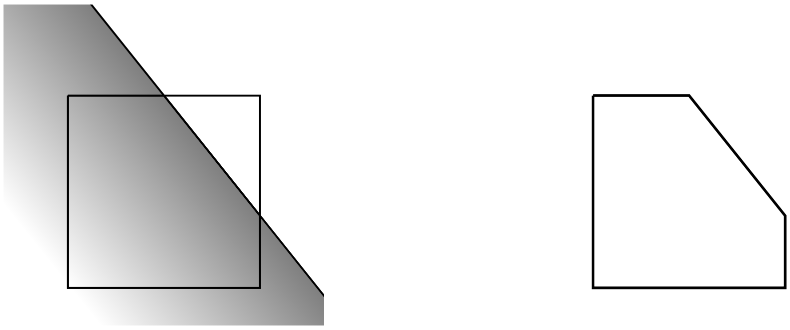
should return `[3 2 1 0]`.

5.6. Clipping polygons

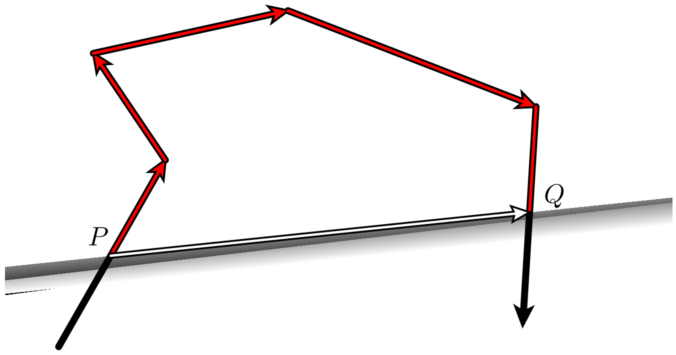
In this section, now that we are equipped with loops and arrays, we shall take up a slight generalization of the problem we began with in the last chapter. We shall also see a new kind of loop.

Here is the new problem:

- *We are given a closed planar polygonal path γ, together with a line $Ax + By + C = 0$. We want to replace γ by its intersection with the half plane $f(x, y) = Ax + By + C \leq 0$.*



We are going to see here a simple but elegant way of solving these problems called the **Hodgman-Sutherland algorithm** after its inventors. We may as well assume the path to be oriented. The solution to the problem reduces to one basic idea: if the path γ exits the half plane at a point P and next crosses back at Q, we want to replace that part of γ between P and Q by the straight line PQ.



We want to design a procedure, which I'll call `hodgman-sutherland`, that has the closed polygon γ and the line $Ax + By + C = 0$ as arguments and returns on the stack a new polygon obtained from γ by cutting off the part in the region $Ax + By + C > 0$. The polygon γ will be represented by the array of its vertices and the line by the array $\ell = (A, B, C)$. Let

$$\langle \ell, P \rangle = Ax + By + C$$

if $P = (x, y)$.

The procedure looks in turn at each edge of the polygon, in the order determined by the array. It starts with the edge P_{n-1}, P_0, a convenient trick in such situations. As we proceed, we are going to build up the replacement polygon by adding points to it. Suppose we are looking at an edge PQ. What we do will depend on circumstances. Roughly put:

(1) If $\langle \ell, P \rangle \le 0$ and $\langle \ell, Q \rangle \le 0$ (both P and Q inside the half plane determined by ℓ) we add Q to the new polygon;
(2) if $\langle \ell, P \rangle < 0$ but $\langle \ell, Q \rangle > 0$ (P inside, Q outside) we add the intersection $PQ \cap \ell$ of the segment PQ with ℓ, which is

$$\frac{\langle \ell, Q \rangle P - \langle \ell, P \rangle Q}{\langle \ell, Q \rangle - \langle \ell, P \rangle} ,$$

to the new polygon;
(3) if $\langle \ell, P \rangle = 0$ but $\langle \ell, Q \rangle > 0$ we do nothing;
(4) if $\langle \ell, P \rangle > 0$ but $\langle \ell, Q \rangle \le 0$ (P outside, Q inside) then we add both $PQ \cap \ell$ and Q, unless they are the same, in which case we just add one;
(5) if both P and Q are outside we do nothing.

This process is very much like that performed in Chapter 4 to find the intersection of a line and a rectangle, and one convention is certainly the same—an edge does not really contain its starting point.

In certain singular cases, one of the vertices lies on ℓ and no new point is calculated.

One peculiar aspect of the process is that if the line crosses and recrosses several times, it still returns a single polygon.

In writing the procedure to do this, we are going to use the `forall` loop. It is used like this:

```
a {
  ...
} forall
```

where a is an array. The procedure `{ ... }` is called once for each element of the array a, with that element on top of the stack. It acts much like the `for` loop, and in fact some `for` loops can be simulated with an equivalent `forall` loop by putting an appropriate array on the stack.

In the program excerpt below there are a few things to notice in addition to the use of `forall`. One is that for efficiency's sake the way in which a local dictionary is used is a bit different from previously. I have defined a procedure `evaluate` which calculates $Ax + By + C$ given `[A B C]` and `[x y]`. If I were following the pattern recommended earlier, this procedure would set up its own local dictionary on each call to it. But setting up a dictionary is inefficient, and `evaluate` is called several times in the main procedure here. So I use no dictionary, but rely entirely on stack operations. The new stack operation used here is `roll`. It has two arguments n and i, shifting the top n elements on the stack cyclically up by i—i.e. it rolls the top n elements of the stack. If the stack is currently

$$x_4\ x_3\ x_2\ x_1\ x_0 \quad \text{(bottom to top)}$$

then the sequence `5 2 roll` changes it to

$$x_1\ x_0\ x_4\ x_3\ x_2\ .$$

```
  % x y [A B C] => Ax + By + C
 /evaluate {  % x y [A B C]
   aload pop  % x y A B C
   5 1 roll   % C x y A B
   3 2 roll   % C x A B y
   mul        % C x A By
   3 1 roll   % C By x A
   mul        % C By Ax
   add add    % Ax+By+C
} def
```

Another thing to notice is that the data we are given are the vertices of the polygon, but what we really want to do is look at its edges, or pairs of successive vertices. So we use two variables P and Q, and start with $P = P_{n-1}$. In looping through P_0, P_1, ... we are therefore looping through edges $P_{n-1}P_0$, P_0P_1, ...

```
  % arguments:  polygon [A B C]
  % returns:  closure of polygon truncated to Ax+By+C <= 0
 /hodgman-sutherland { 4 dict begin
 /f exch def
 /p exch def
 /n p length def
  % P = p[n-1] to start
 /P p n 1 sub get def
 /d P length 1 sub def
 /fP P aload pop f evaluate def
 [
   p {
     /Q exch def
     /fQ Q aload pop f evaluate def
     fP 0 le {
       fQ 0 le {
         % P <= 0, Q <= 0:  add Q
         Q
       }{
         % P <= 0, Q > 0
         fP 0 lt {
           % if P < 0, add intersection
           /QP fQ fP sub def
           [
             fQ P 0 get mul fP Q 0 get mul sub QP div
```

```
              fQ P 1 get mul fP Q 1 get mul sub QP div
            ]
          } if
        } ifelse
      }{
        % P > 0
        fQ 0 le {
          % P > 0, Q <= 0:  if fQ < 0, add intersection;
          % add Q in any case
          fQ 0 lt {
            /QP fQ fP sub def
            [
              fQ P 0 get mul fP Q 0 get mul sub QP div
              fQ P 1 get mul fP Q 1 get mul sub QP div
            ]
          } if
          Q
        } if
        % else P > 0, Q > 0:  do nothing
      } ifelse
      /P Q def
      /fP fQ def
    } forall
  ]
end } def
```

Exercise 5.6. *If a path goes exactly to a line and then retreats, the code above will include in the new path just the single point of contact. Redesign the procedure so as to put two copies of that point in the new path. It is often useful to have a path cross a line in an even number of points.*

5.7. Code

There is a sample function graph in `function-graph.ps`, and code for polygon clipping in dimensions three as well as two in `hodgman-sutherland.inc`.