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

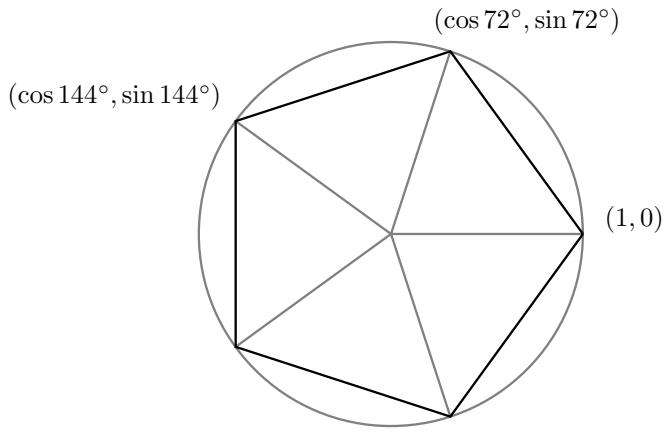
1. The repeat loop

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 which 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 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. 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. 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 loop where the variable exceeds 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 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 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 centre of the axes should be at the centre of the page. Put in a margin of 1 cm. all around the page.

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. 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 always be used together.

4. General polygons

Polygons don't have to be regular. In general a polygon is essentially a sequence of points P_0, P_1, ..., 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.

```
% array of points

/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.

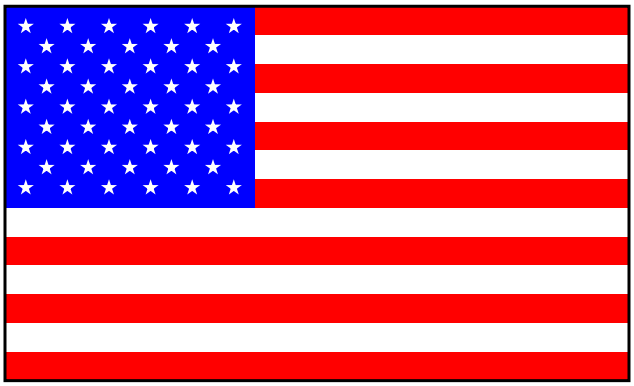
There is another way to unload the items in an array onto the stack: `P aload` puts all the entries of `P` onto the stack, together with the array `P` itself at the top. `P aload pop` thus puts all the entries on the stack, in order. This is simpler and more efficient than getting the items one by one.

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 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. 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.)



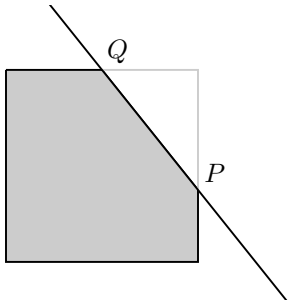
5. Clipping

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:

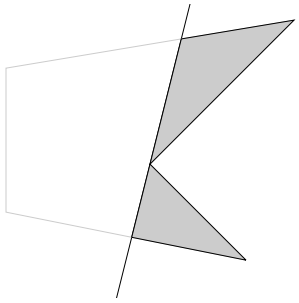
- *We are given a planar polygonal path γ, together with a line $Ax + By + C = 0$. We want to replace γ by its intersection with the half plane $Ax + By + C \leq 0$.*

In other words, if the path 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 are going to see here a simple but elegant way of solving these problems called the **Hodgman-Sutherland algorithm** after its inventors.

The tricky part is to have a robust algorithm which handles possible floating point errors correctly in the face of a situation like this



where a small modification changes the type of intersection quite a lot.

How is it going to work? First of all, let's think of the polygon γ as an array of points $[P_0,\ P_1, \ldots, P_{n-1}]$, where each point $P = (x, y)$ is itself a two-dimensional array of its coordinates. The line is represented by an array $\ell = [A,\ B,\ C]$. In these circumstances I write for convenience

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

We want to design a procedure, which I'll call `hodgman-sutherland`, that has γ and ℓ 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 procedure looks in turn at each edge of the polygon, in the order determined by the array. 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:

(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 $\langle \ell, P\rangle \le 0$ but $\langle \ell, Q\rangle > 0$ (P inside, Q outside) we add the intersection $PQ \cap \ell$ of the segment PQ with ℓ to the new polygon;
(3) 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;
(4) if both P and Q are outside we do nothing.

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 by putting an appropriate array on the stack.

Exercise 5. Replace

```
0 1 10 {
  ...
} for
```

with an equivalent `forall` loop.

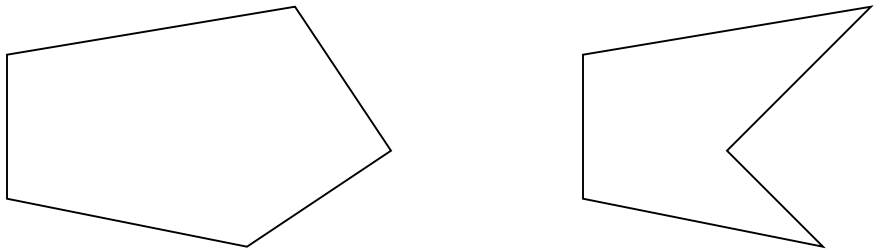
In the program excerpt below there are a few things to notice in addition to the use of `forall`. One is that for efficiency's sake the way in which a local dictionary is used is a bit different from previously. I have defined a procedure `evaluate` which calculates $Ax + By + C$ given `[A B C]` and `[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. Instead, I set up a single dictionary to be used on all calls to `evaluate`, and bring it up on each call to the main procedure. In any event, you don't have to understand this technique in order to use it.

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, ...

```
% [A B C] [x y] => Ax + By + C

/hsdict 10 dict def

hsdict begin

/evaluate {
  /hsP exch def
  /hsell exch def
  hsell 0 get hsP 0 get mul
  hsell 1 get hsP 1 get mul
  add
  hsell 2 get
  add
} def

end

% polygon line => truncated polygon
```

```
/hodgman-sutherland { hsdict begin
  /ell exch def
  /p   exch def
  /n p length def
  % P = p[n-1] to start
  /P p n 1 sub get def
  /CP ell P evaluate def
  % (a\n) print
  [
  p {
    /Q exch def
    /CQ ell Q evaluate def
    % (b\n) print
    CP 0 le {
      CQ 0 le {
        % P in, Q in
        Q
      }{
        % P in, Q out
        % R
        [
          CQ P 0 get mul CP Q 0 get mul sub CQ CP sub div
          CQ P 1 get mul CP Q 1 get mul sub CQ CP sub div
        ]
      } ifelse
    } {
      CQ 0 le {
       % P out, Q in
       % R
        [
          CP Q 0 get mul CQ P 0 get mul sub CP CQ sub div
          CP Q 1 get mul CQ P 1 get mul sub CP CQ sub div
        ]
       Q
      } if
    } ifelse
    % else both out
    /P Q def
    /CP CQ def
  } forall
  ]
end } def
```

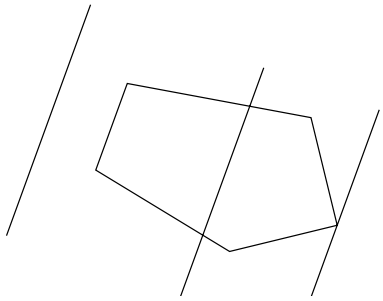
There is a related problem we shall also look at. A closed polygon is called **convex** if it bends outwards everywhere. Thus in the following figure the polygon on the left is convex but that on the right is not.



A convex polygon can be described in either or two ways: (1) the array of its successive vertices; (2) the array of the 'lines' `[A B C]` characterizing its successive edges, with the polygon on the inside of $Ax + By + C \leq 0$.

Exercise 6. Design a PostScript procedure which converts the vertex form to the line form.

If we intersect a convex polygon by a straight line we can get any of these types of intersection: (1) nothing; (2) a single line segment, either part of the boundary of the polygon or part of its interior; (3) a single point.



The technique we use for the first problem will be equally capable of finding such an intersection. Drawing the visible part of a line is a special case of this where the polygon is the boundary of a page.

Exercise 7. Design a PostScript procedure which has two arguments, a line `[A B C]` and a convex polygon characterized by inequalities $A_i x + B_i + C_i \leq 0$, and returns 0, 1, or 2 points where the line intersects the polygon.