Mathematics 308 — Geometry

Chapter 3. Evolution of a program

One rule for happy PostScript programming is to first get a simple picture up on the screen that comes somewhere close to what you want, and then refine it and add to it until it is exactly what you want. The main thing to avoid is to have to debug large segments of PostScript all at once, since debugging is so painful.

One thing to keep in mind as you do this development is the goal of flexibility—can you reuse in another drawing what you are doing in this one?

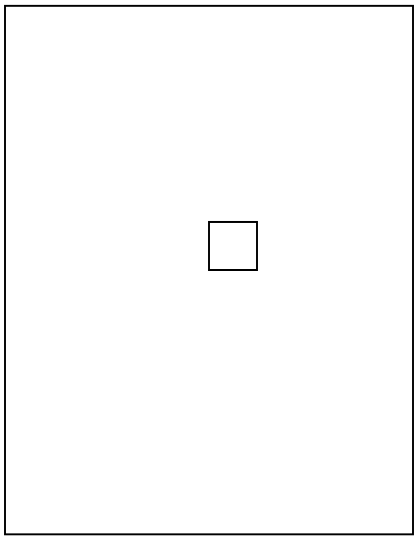
In this Chapter I shall show how one PostScript program evolves according to this process. Technically, the main ingredients we are going to add to your tool kit are **variables** and **procedures**.

1. Drawing a square on a page

The following program draws a square one inch on a side roughly in the middle of a page.

```
%!

72 72 scale
4.25 5.5 translate
0.012 setlinewidth

newpath
0 0 moveto
1 0 rlineto
0 1 rlineto
-1 0 rlineto
closepath
stroke

showpage
```

The page looks like this:



This program is extremely simple, and not very interesting. Among other things, it is not very flexible.

Suppose you wanted to change the size of the square? You would have to replace each occurrence of "1" with the new size. This is awkward—you might miss an occurrence, at least if your program were more complicated. It would be better to introduce a **variable** s to control the length of the side of the square.

Variables in PostScript can be just about any sequence of letters and symbols. They are defined in statements like this

```
/s 1 def
```

which sets the variable s to be 1. After a variable is defined in your program, any occurrence of that variable will be replaced by what it was defined to be.

Using a variable for the side of the square, the new program would look like this (I include only the interesting parts from now on):

```
/s 1 def

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
```

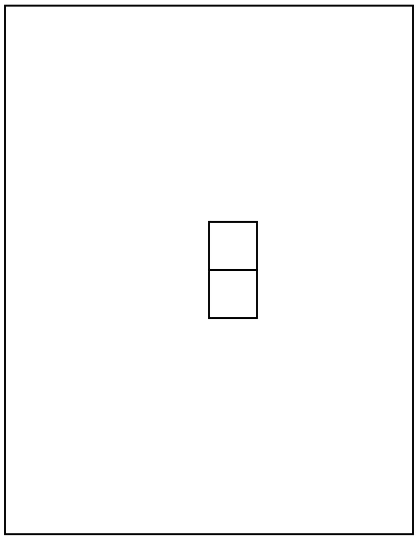
One new thing to note here is the command **neg**, which replaces anything on the top of the stack by its negative. This code is indeed a bit more flexible than the original, because if you want to draw a square of different size you would have to change only one line.

Suppose you wanted to draw two squares, one of them say at $(0, -1)$ (that is to say, just below the first)? Most straightforward:

```
/s 1 def

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke

0 -1 translate

newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke

showpage
```

which just repeats the part of the program which actually draws the square, of course. Recall that **translate** shifts the origin of the user's coordinate system in the current units.

Now the page looks like this:



2. Procedures in PostScript

Repeating the code to draw the square is somewhat inefficient—this technique will lead to a lot of text pasting and turns out to be very prone to error. It is both more efficient and safer to use a PostScript **procedure** to repeat the code for you. A procedure in PostScript is an extremely simple thing—it is just any sequence of commands, enclosed in brackets {...}. You can assign variables to procedures just like any other kind of data. When you insert this variable in your program, it is replaced by the sequence of commands inside the brackets. In this way, you do things in two steps:

(1) You define a procedure called **draw-square** in the following way:

```
/draw-square {
newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
} def
```

I repeat: the effect of this is that after you have made this definition, whenever you have the expression **draw-square** in your program, PostScript will simply substitute the lines in between the curly brackets **{** and **}**. The effect of calling a procedure in PostScript is always this sort of text substitution.

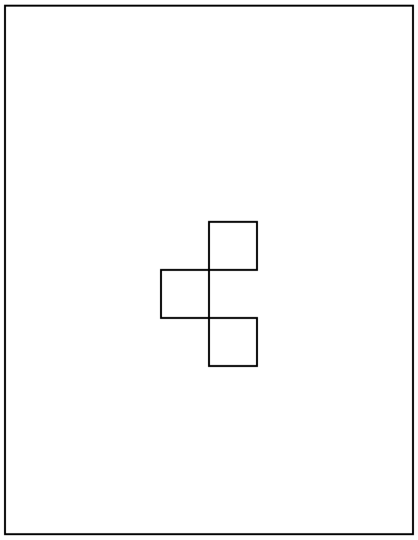
(2) Then you call the procedure when you need it. In this case, the new commands on the page will include the above definition, and also this:

```
draw-square
0 -1 translate
draw-square
```

Of course if we have done things correctly, the page looks the same as before. But we can now change it easily by mixing several translations and calls to **draw-square** like this:

```
draw-square
-1 -1 translate
draw-square
1 -1 translate
draw-square
```

to get this:



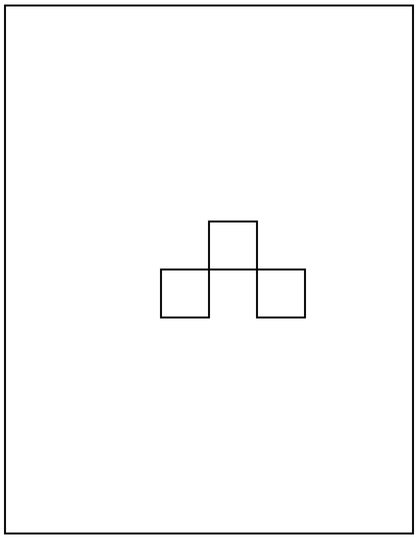
3. Keeping track of where you are

In the lines of PostScript above, you can easily forget exactly where you are with all those translations. What you might do is translate back again after each translation and drawing operation to restore the original coordinates. But this would become complicated later on in your work, when you will perform several changes of coordinates and it will be difficult to figure out how to invert them. Instead, you can get PostScript to do the work of remembering where you are. It has a pair of commands that help you do the job easily: **`gsave`** saves the current coordinate system somewhere (together with a few other things like the current line width) and **`grestore`** brings back the coordinate system you saved with your last **`gsave`**. *They have to come in pairs!*

In this scheme we could write

```
draw-square

gsave
-1 -1 translate
draw-square
grestore

1 -1 translate
draw-square
```

Now we get



To be a bit more precise, **`gsave`** saves the **current graphics state** and **`grestore`** brings it back. The graphics state holds data about coordinates, line widths, the way lines are joined together, the current color, and more—in effect everything that you can change easily to affect how things are drawn. You might recall that we saw **`gsave`** and **`grestore`** earlier, where we used them to set up successive pages correctly, enclosing each page in a pair of **`gsave`** and **`grestore`**.

Incidentally, it is usually—but not always—a bad (very bad) idea to change anything in the graphics state in the middle of drawing a path. Effects of this bad practice are often unintuitive, and therefore unexpected.

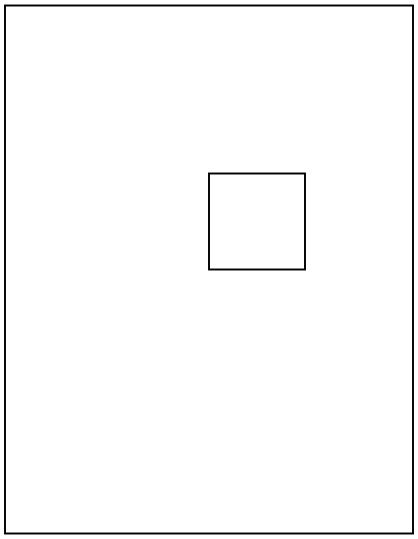
4. Local variables

The definition we have made of the procedure **`draw-square`** has a variable s in it. The variable s is not defined in the procedure itself, but must be defined before the procedure is used. This is awkward—if you want to draw squares of different sizes, you have to redefine s each time you want to use a new size.

For example, if we write

```
/s 2 def

draw-square
```

we get



There are a few tricky things to keep in mind here. (1) When I put s in the program it replaces that occurrence of s by the **current value** of s, which is 2. This again illustrates how **`def`** works by substitution. (2) The line **`s neg`** just puts $-s$ on the stack, and does not change the value of the variable s. The only way to change the value of s is to redefine it. (3) The variable s is used in the procedure, but it does not have to be defined until the procedure is actually used. (4) The phrase **`/s 2 def`** assigns the value of 2 to s. This use of **`def`** might seem quite a bit different from its use to define procedures, but in fact it is not: in both cases it effects a straightforward substitution when the variable defined is put into the program—in the first case the variable **`draw-square`** is defined to be a procedure, in the second s is defined to be the number 2. (5) The variable s is a **global variable**. It can be used anywhere in the program after it has been defined. If you called **`draw-square`** before you defined s you would get an error message. (6) The **`def`** command requires that exactly one item separates the word **`def`** from the name of the variable being defined. Thus **`/s 2 3 def`** is not acceptable. If for some reason you want s to be the sequence of numbers **`2 3`** you would write **`/s {2 3} def`**. This is OK because **`{2 3}`** is a procedure which just places 2 and 3 on the stack in succession.

Let me repeat: *If you want to assign a new value to a variable you have to define it over again.*

It is awkward to have to assign a value to s every time we want to draw a square. It would be much better if we could just type something like **`2 draw-square`** to do the job. We can in fact do this, by doing a bit of stack manipulation. The command **`exch`** exchanges the top two items on the stack. Therefore **`2 /s exch def`** has exactly the same effect as **`/s 2 def`**, since in the first version the effect of **`exch`** is to change **`2 /s`** to **`/s 2`** and then add **`def`**. Thus the lines

```
/draw-square {
/s exch def
newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
} def

2 draw-square
```

do exactly what we want. The important point is that the procedure itself now handles the assignment of a value to s, and all we do is pass the value of s to the procedure as an argument to it by putting it on the stack before the procedure is called. If you know how programming language compilers work, you will recognize this as what programming languages do to pass arguments. The difference is that PostScript does it in the open, and effectively forces you to do a bit more work yourself. If you wanted to draw rectangles with different width and height, you would pass two arguments in a similar way:

```
/draw-rectangle {
/h exch def
/w exch def
newpath
0 0 moveto
w 0 rlineto
0 h rlineto
w neg 0 rlineto
closepath
stroke
} def

2 3 draw-rectangle
```

draws a rectangle of width 2 and height 3. Notice that the stuff on the stack is removed in the order *opposite* to that in which you placed it there.

Another problem is possible name conflicts. If you have a large program with lots of different figures being drawn in various orders, you might very well have several places where you use w and h with different meanings. This can cause a lot of trouble. The way around this is a technique in PostScript that I suggest you use without trying to understand too much about it in detail. We want the variables we use in a procedure to be **local** to that procedure, so that assignments we make to them inside that procedure don't affect other variables with the same name outside the procedure. To do this we add some lines to the procedure:

```
/draw-rectangle {
2 dict begin
/h exch def
/w exch def
newpath
0 0 moveto
w 0 rlineto
0 h rlineto
w neg 0 rlineto
0 h neg rlineto
closepath
stroke
end
} def

2 3 draw-rectangle
```

The line **`2 dict begin`** sets up a local variable mechanism, and **end** restores the original environment. The **`2`** is in the statement because we are defining 2 local variables. *You should begin and end all procedures in which you make variable definitions in this way.* The only tricky thing to be aware of is that *all variables defined within this pair will be local variables, so that is impossible to change the value of global variables within them.* It is not usually a good idea to redefine global variables within a procedure, anyway. I don't think it is too strong to say that you should **never** assign a value to a global variable inside a PostScript procedure. Again: **`begin`** *and* **end** *have to come in pairs.* If they don't, the effect will be that a certain block of space in the computer will fill up. You might get away with it for a while, but unless you are careful sooner or later some awful error is bound occur because of this problem.

At any rate, don't worry too much about exactly what is going on here. Just copy the pattern without thinking about it. The **2** could have been **3** or **4** or **20**, but it does have to be at least as large as the number of variables you are about to define.

5. A final improvement

I have already mentioned in class that it is usually a good idea to use procedures to build paths without drawing them. Furthermore, it is *always* a good idea to tell in a comment what you have to do to use a procedure, and what its effect is. Thus

```
% Builds a rectangular path with
% first corner at origin.
% On stack at entry:
% width height

/rectangle {
2 dict begin
/h exch def
/w exch def
0 0 moveto
w 0 rlineto
0 h rlineto
w neg 0 rlineto
0 h neg rlineto
closepath
end
} def

newpath
2 3 rectangle
stroke
```

is the preferable way to use a procedure to draw rectangles. This way you can **fill** them or **clip** them as well as **stroke** them. (We shall meet clipping later.) You can also link paths together to make more complicated paths.