Chapter 1. Getting started in PostScript

PostScript is a rather low level computer language developed by Adobe Systems in its startup days. Its primary purpose is to produce high quality graphics from computers, and especially to output computer graphics on printers. Professional graphics designers, for example, often work with high-end programs that in turn produce their output through PostScript. To them, the PostScript itself is usually invisible. One would not therefore expect PostScript to be comprehensible to humans. It is nonetheless a convenient computer language for producing by hand, so to speak, designs with a large mathematical component. One great virtue of PostScript is that there is no effective limit on the *quality* of the final product, precisely because it is designed for use by professionals. Some other virtues are that it is amenable to 'programming by cut and paste', and that the effects of commands are very local in nature—local, that is to say, within the text of the program. Both virtues and drawbacks will become better known as we proceed.

In these notes we shall use a program called `ghostscript`, as well as one of several programs which in turn rely on `ghostscript` running behind the scenes, to serve as our PostScript interpreter and interface. All the programs we shall use are available without cost through the Internet.

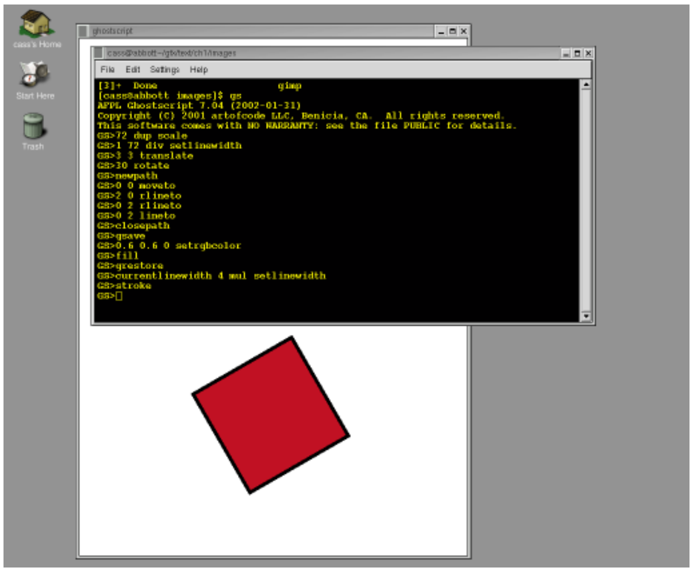
Be careful—the language we are writing our programs in is **PostScript**, and the program we'll use to interpret them is `ghostscript`.

The interpreter `ghostscript` has by itself a relatively primitive user interface which will turn out to be too awkward to use for very long, but learning this interface will give you a valuable feel for the way PostScript works. Furthermore, it will continue to serve a useful purpose, in debugging as well as animations.

We shall begin in this chapter by showing how `ghostscript` works, and then later on explain a more convenient way to produce pictures with PostScript.

1. Simple drawing

Start up `ghostscript`. On Unix networks this is usually done by typing gs, and on other systems it is usually done by clicking on the icon for `ghostscript`. What you get while gs is running are two windows, one a kind of terminal window into which you type commands and from which you read plain text output, and the other a graphics window in which things are drawn.



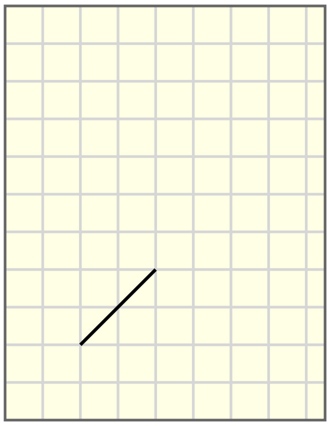
The graphics window—which I shall often call the **page**—opens up with a default coordinate system. The origin of the default coordinate system on a page is at the lower left, and the unit of measurement—which is the same in both horizontal and vertical directions—is equal to a **point**, exactly equal to $1/72$ of an inch. (This **Adobe point** is almost, but not quite, the same as the classical **printer's point**, which measures 72.27 to an inch.) Usually the size of the graphics window is $8.5'' \times 11''$, or 612×792 points^2. As we shall see in a moment, the coordinate system can be easily changed so as to arrange x and y units to be anything you want, with the origin anywhere in the plane of the page.

When I start up running my local version of `ghostscript` in a terminal window I get a display in that window looking like this:

```
AFPL Ghostscript 7.04 (2002-01-31)
Copyright (C) 2001 artofcode LLC, Benicia, CA.  All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>
```

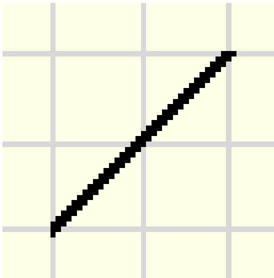
In short, I am facing the `ghostscript` **prompt** `GS>`, and I am expected to type in commands. Let's start off by drawing a line in the middle of the page. On the left is what the terminal window displays, and on the right is what the graphics window looks like:

```
GS>newpath
GS>144 144 moveto
GS>288 288 lineto
GS>stroke
GS>
```



(The grid is just there to help you orient yourself, and not displayed in the real window.) The machine produces the prompts here, and everything else is typed by you. The graphics window displays the diagonal line in the figure on the right.

If we look really closely at the line on the screen that comes up, say with a magnifying glass, we'll see a rather jagged image:



That's because the screen is made up of pixels, with about 75 pixels in an inch. But PostScript is a **scalable** graphics language, which means that if you look at output on a device with a higher resolution than your screen, the effects of pixelization will be seen only at that resolution. Exactly how the computer transforms the directions for drawing a line into a bunch of black pixels is an extremely interesting question, but not one that these notes will consider. So in effect all lines will be assumed to be . . . well, lines—not pixelly things that look odd and ugly close up.

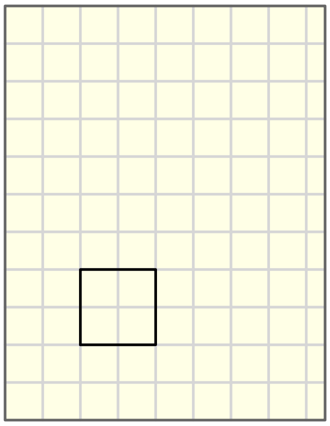
You draw things in PostScript by constructing **paths**. Any path in PostScript is a sequence of lines and curves. At the beginning, we shall work only with lines. In all cases, first you **build** a path, and then you actually **draw** it.

- You begin building a path with the command `newpath`. This is like picking up a pen to begin drawing on a piece of paper. In case you have already drawn a path, the command `newpath` also clears away the old path.
- You start the path itself with the command `moveto`. This is like placing your pen at the beginning of your path. In PostScript, things are generally what you might think to be backwards, so you write down *first* the coordinates of the point to move to, *then* the command.
- You add a line to your path with the command `lineto`. This is like moving your pen on the paper. Again you place the coordinates first, then the command.
- So far you have just built your path. You draw it—i.e. make it visible—with the command `stroke`. You have some choice over what colour you can draw with, but the colour that is used by default is black.

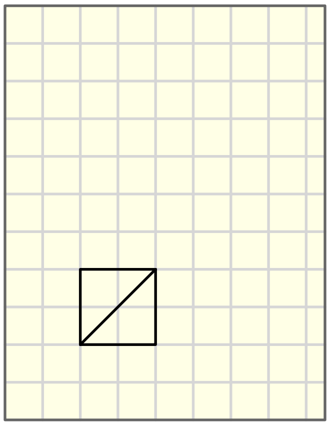
From now on I will usually leave the prompts `GS>` *out.*

You would draw a square 2 inches on a side with the command sequence

```
newpath
144 144 moveto
288 144 lineto
288 288 lineto
144 288 lineto
144 144 lineto
stroke
```



If you type this immediately after the previous command sequence, you will just put the square down on top of the line you have already drawn:



I will tell you later how to start over with a clean page. For now, it is important to remember that *PostScript paints over what you have already drawn*, just like painting on a canvas. There is no command that erases stuff already drawn.

There are often lots of different ways to do the same thing in PostScript. Here is a different way to draw the square:

```
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
stroke
```

The commands `rmoveto` and `rlineto` mean motion **relative** to where you were before. The command `closepath` closes up your path back to the last point to which you applied a `moveto`.

A very different effect is obtained with:

```
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
fill
```

This just makes a big black square in the same location. *Whenever you build a path, the operations you perform to make it visible are* `stroke` *and* `fill`*.* The first draws the path, the second fills the region inside it.

You can draw in different shades and colours with two different commands, `setgray` and `setrgbcolor`. Thus:

```
0.5 setgray
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
fill
```

will make a grey square, and

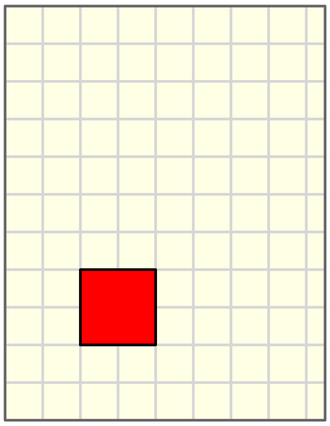
```
1 0 0 setrgbcolor
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
fill
```

will make a red one. The `rgb` here stands for Red, Green, Blue, and for each colour you choose a set of three parameters between 0 and 1. Whenever you set a new colour, it will generally persist until you change it again. Note that `0` is black, `1` white. The command `x setgray` is the same as `x x x setrgbcolor`. You can remember that `1` is white by recalling from high school physics that white is made up of all the colors put together.

Exercise 1. How would you set the current colour to green? Pink? Violet? Orange?

If you want to draw a red square with a black outline you type

```
1 0 0 setrgbcolor
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
fill
0 setgray
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
stroke
```



We shall see later a way to produce this figure without redundant typing.

Exercise 2. Run `ghostscript`. Draw an equilateral triangle near the centre of the page, instead of a square. Make it 100 points on a side, with one side horizontal. First draw it in outline, then fill it in black. Next, make it in turn red, green, and blue with a black outline. (You will have to do a few calculations first. In fact, as we shall see later, you can get PostScript to do the calculations.)

2. Simple coordinate changes

Working with points as a unit of measure is not for most purposes very convenient. For North Americans, since the default page size is $8.5'' \times 11''$, working with inches usually proves easier. We can change the basic unit of length to an inch by typing

```
72 72 scale
```

which scales up the x and y units by a factor of 72. Scaling affects the current units, so scaling by 72 is the same as scaling first by 8, then by 9. This is the way it always works. The general principle here is this:

- *Coordinate changes are always interpreted relative to the current coordinate system.*

You can scale the x and y axes separately, but it is usually a bad idea. Lines are themselves drawable objects of finite width. If scaling is not uniform, the thickness of a line will depend on its direction. To be sure to get both scale factors the same, you can also type `72 dup scale`. The command `dup` duplicates the previous entry.

When you scale, you must take into account the fact that the default choice of the width of lines is 1 unit. So if you scale to inches, you will get lines 1 inch wide unless you do something about it. It might be a good idea to add

```
0.01389 setlinewidth
```

when you scale to inches. This sets the width of lines to $1/72$ of an inch. A linewidth of 0 is also allowable—it just produces the thinnest possible lines which do not actually vanish. You should realize, however, that on a device of high resolution, such as a 1200 DPI printer, this will be nearly invisible. Setting the line width to 0 contradicts the general principle that *you should always aim in PostScript to produce figures which do not in any way depend directly on the particular device on which it will be reproduced.*

Exercise 3. How would you scale to centimeters?

You can also shift the origin.

```
1 2 translate
```

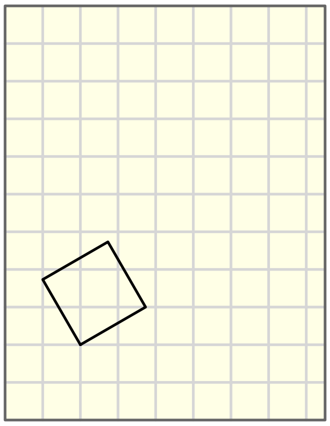
moves the coordinate origin to the right by 1 unit and up by 2 units. The combination

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

moves the origin to the centre of an $8.5'' \times 11''$ page.

One more simple coordinate change command rotates things.

```
144 144 translate
30 rotate
newpath
0 0 moveto
144 0 lineto
144 144 lineto
0 144 lineto
0 0 lineto
stroke
```



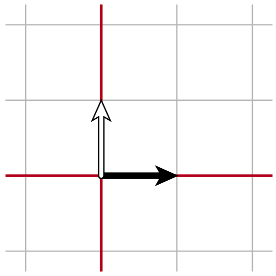
The translation is done first because rotation always takes place around the current origin. Note that *PostScript works with angles in* **degrees**. This will cause us some trouble later on, but for now it is probably A Good Thing.

Exercise 4. Europeans use A4 paper. Find out its dimensions, and show how to draw a square one centimetre on a side with centre in the middle of an A4 page.

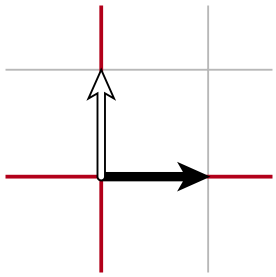
3. Coordinate frames

The way to visualize how the coordinate changes `scale`, `translate`, and `rotate` affect drawing is by thinking of **coordinate frames**.

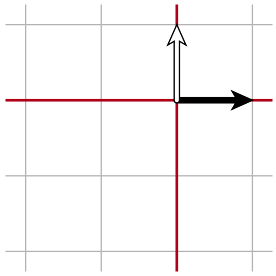
A simple frame, with units in centimetres



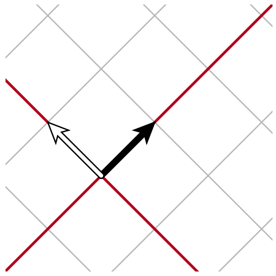
Scaled by $\sqrt{2}$ in both directions



Translated by $[1, 1]$



Rotated by $45°$



There are two fundamental things to keep in mind when wondering how coordinate changes affect drawing.

- *Coordinate changes affect the current frame in the natural way. That is to say,* `2 2 scale` *scales the current frame vectors by a factor of* 2*, etc.*
- *Drawing commands take effect relative to the current frame.*

4. Doing arithmetic in PostScript

PostScript is a complete programming language. But with the goal of handling data rapidly, it has only limited built-in arithmetical capabilities. In most programming languages, both integers and real numbers are of severely limited precision. In some implementations of PostScript, integers must lie in the range $[-32784, 32783]$, and real numbers are only accurate to about 7 significant places. This is where the the roots of the language in graphics work show up, since normally drawing a picture on a page of reasonable size does not have to be extremely accurate. This limited accuracy is not usually a problem, but it does mean you have to be careful.

At any rate, with arithmetical operations as with drawing operations, the sequence of commands is backwards from what you might expect. To add two numbers, first enter the numbers, followed by the command `add`. The result of adding numbers is also not quite what you might expect. Here is a sample run in the `ghostscript` interpreter:

```
GS>3 4 add
GS<1>
```

What's going on here? What does the `<1>` mean? Where is the answer?

PostScript uses a **stack** to do its operations. This is an array of arbitrary length which grows and shrinks as programs move along. The very first item entered is said to be at the **bottom** of the stack, and the last item entered is said to be at its **top**. This is rather like manipulating dishes at a cafeteria. Generally, operations affect only the things towards the top of the stack, and compute them without displaying results. For example, the sequence `3 4 add` does this:

Entry	*What happens*	*What the stack looks like*
`3`	The number 3 goes onto the stack	`3`
`4`	The number 4 goes above the 3 on the stack	`3 4`
`add`	The operation `add` goes above 4 . . .	`3 4 add`
	. . . then collapses the stack to just a single number	`7`

You might be able to guess now that the `<1>` in our run tells us the size of the stack. To display the top of the stack we type `=`. If we do this we get

```
GS>3 4 add
GS<1>=
7
GS>
```

Note that `=` removes the result when it displays it (as does the similar command `==`). An alternative is to type `stack` or `pstack`, which displays the entire stack, and does not destroy anything on it.

```
GS>3 4 add
GS<1>stack
7
GS<1>
```

The difference between `=` and `==` is too technical to explain here, but in practice you should usually use `==`. Similarly, you should usually use `pstack`, which is a bit more capable than `stack`. There is a third and slightly more sophisticated display operator called `print`. It differs from the others in that it does not automatically put in a carriage return, and can be used to format output. The `print` command applies basically only to strings, which are put within parentheses. (Refer to Appendix 1 for instructions on how to use `print`.) One feature of `print` is that its output will normally appear in the graphics window rather than in the terminal window.

Other arithmetic operations are `sub`, `mul`, `div`. Some of the mathematical functions we can use are `sqrt`, `cos`, `sin`, `atan`. For example, here is a command sequence computing and displaying $\sqrt{3*3+4*4}$.

```
GS>3 3 mul
GS<1>4 4 mul
GS<2>add
GS<1>sqrt
GS<1>=
5.0
GS>
```

Exercise 5. Explain what the stack holds as the calculation proceeds.

Exercise 6. Use `ghostscript` to calculate and display $\sqrt{9^2+7^2}$.

Here is a list of nearly all the mathematical operations and functions.

`x y add`	puts $x+y$ on the stack
`x y sub`	puts $x-y$ on the stack
`x y mul`	puts xy on the stack
`x y div`	puts x/y on the stack
`x neg`	puts $-x$ on the stack
`y x atan`	puts the polar angle of (x, y) on the stack (in degrees)
`x sqrt`	puts $\sqrt{x}$ on the stack
`x sin`	puts $\sin x$ on the stack (x in degrees)
`x cos`	puts $\cos x$ on the stack (x in degrees)
`y x exp`	puts y^x on the stack
`x ln`	puts $\ln x$ on the stack
`x abs`	puts $\|x\|$ on the stack
`x floor`	puts on the stack the integer just below x

Exercise 7. Recall that $\mathrm{acos}(x)$ is the unique angle between 0° and 180° whose cosine is x. Use `ghostscript` to find $\mathrm{acos}(0.4)$. (This will require thinking a bit about the geometry of angles.) (Note: some versions of PostScript will have `acos` built in. Do not use it, but restrict yourself to getting by with `atan`.)

In addition to these mathematical functions you will probably find useful a few elementary commands that move things around on the stack.

`x y exch`	exchanges the top two items on the stack to make them `y x`
`x pop`	removes the top item on the stack

5. Errors

You will make mistakes from time to time. The default method for handling errors in `ghostscript` (and indeed in all PostScript interpreters I am familiar with) is pretty poor. This is not an easy problem to correct, unfortunately. Here is a typical session with a mistake signaled. If you enter

```
GS>5 0 div
```

this is what you will get (more or less) spilled out on the screen:

```
Error:  /undefinedresult in --div--
Operand stack:
   5   0
Execution stack:
   %interp_exit   --nostringval--
     --nostringval--   --nostringval--
     %loop_continue   --nostringval--
     --nostringval--   false   --nostringval--
     --nostringval--   --nostringval--
Dictionary stack:
   --dict:592/631--   --dict:0/20--   --dict:34/200--
Current allocation mode is local
GS<2>
```

Holy cow! Your adrenaline level goes way up and your palms break out in sweat. What the `$%#?!` are you supposed to do now? *Calm down.* The important thing here and in just about all error messages from `ghostscript` is that you can ignore all but these first lines:

```
Error:  /undefinedresult in --div--
Operand stack:
   5   0
```

which shows you the general category of error and what the stack was like when the error occurred. Here it is division by 0. *It never pays to try too hard to interpret* `ghostscript` *error messages.* The only way to deal with them is to try to figure out where the error occurred, and examine your input carefully. There is one trick you can use to find out where the error occurred: put lines like `(1) =` or `(location #1) =` at various points in your program and try to trace how things go from the way output is displayed. Simple, but it often helps. The way this works is that `(1)` denotes the string "1", and = will display it on the terminal. This technique is clumsy, but not much more is possible.

If you are running `ghostscript`, then to recover from an error you probably want to clear the stack completely and start over, with the single command `clear`.

Incidentally, the way errors are handled by your PostScript interpreter can be modified by suitable embedded PostScript code. In particular, there is a convenient error handler called `ehandler.ps` available from Adobe via the Internet, at `www.adobe.com`. If you have a copy of it in your current directory, you can use it by putting

```
(ehandler.ps) run
```

at the beginning of your program. You can also arrange for `ghostscript` to use it instead of its default error handling, but exactly how depends on which computer you are using. *If you do import* `ehandler.ps` *you must remove all reference to it before sending your work to a printer.*

You should keep in mind that, even for experts, tracking a PostScript program explicitly can be very difficult. One way to write better code in the first place is to include lots of comments, so that someone (usually the programmer!) can tell what the program is doing without following the code itself. In PostScript these are begun with a percent sign `%`. *All text in a line after a* `%` *is ignored.* Thus the effect of these two lines is the same:

```
(ehandler.ps) run
(ehandler.ps) run     % imports the error handler
```

Another trick for more convenient debugging is to run your program without visual output. On a **Unix** system thsi is done with the command gsnd (for 'ghostscript no display') plus the name of the input file. The messages you get are the same, but this seems to help you concentrate on what `ghostscript` is trying to tell you.

6. Working with files and `ghostview` or `GSView`

Using the `ghostscript` interpreter directly shows interesting things, and you should be ready and willing to do it occasionally, but it is an extremely inefficient way to produce pictures, mostly because data entered cannot be changed easily, and errors will usually force you to start all over again. Much better is the following sequence of operations (in which `gview` is a `ghostscript` viewer such as `ghostview` or `GSView`, depending on your system):

- *Start up* `gview`*.*
- *Start up a text editor.*
- *Create or open up in your text editor the file you want to hold your PostScript program. Be sure your file is to be saved as plain text, as opposed to one of the special formats word processors seem to prefer. On Windows machines, this is the default with the simple editing program* `Notepad`*.* `Notepad` *is capable of handling only short files, however, and sooner or later you will find it inadequate.*
- *Open up that file from* `gview`*.*
- *As you make up your program inside the editor, save it from time to time and reopen it in* `gview`*, where your picture and possibly other messages will be displayed.*

There are some new features of using files for PostScript programs that you'll have to take into account, but otherwise this works well—indeed, almost painlessly.

- *At the very beginning of your file you must have the two characters* `%!`*. This tells your computer that the file is a PostScript file.*
- *At the end of your file you should have a line with* `showpage` *on it.*

Neither of these is usually absolutely necessary, but there will be times when both are required. *They will definitely be required if you want to print out your picture on a printer, or if you want to import your PostScript file into an image manipulation program in order to turn it into a graphics file of some other format.*

The command `showpage` displays the current page, at least in some situations, and then starts a new page. Later on you will want to make up files with several pages in them, and each page must have a `showpage` at the end. There is one tricky feature of `showpage`, however.

- *Scaling must be done over again on each page.*

There are better and worse ways to deal with this. The best is to put the commands `gsave` at the beginning and `grestore` at the end of each page. We will see later exactly what these commands do, but the brief description is that they save and restore the graphics state. Using them as I suggest here just means that every page starts all over in the original graphics environment.

Here, for example, is a complete two-page program:

```
%!

gsave

72 72 scale
0.01389 setlinewidth
4 5 translate

newpath
0 0 moveto
1 0 lineto
0.5 1 lineto
closepath
stroke

grestore
showpage

gsave

72 72 scale
0.01389 setlinewidth
4 5 translate

newpath
0 0 moveto
1 0 lineto
1 1 lineto
closepath
stroke

grestore
showpage
```

Exercise 8. What does this program do?

By the way, I want to emphasise that spaces, tabs, and line breaks are all the same to PostScript. Thus in the above program I could have written either

```
72 72 scale
0.01389 setlinewidth
4 5 translate
```

or

```
72 72 scale 0.01389 setlinewidth 4 5 translate
```

The only reason to be careful about spaces or tabs or line breaks in a PostScript program is to make the program readable by humans. This is extremely important to keep in mind, even when the only person who reads the program is the one who writes it. Because the person who reads a program is **never** the one who writes it. The programmer's brain inevitably changes state in between writing and reading, and it is often *very* difficult for the reader of tomorrow to recall exactly what the writer of today had in mind.

7. Some fine points

There are a number of commands that control fine points of the way PostScript draws.

Every line in PostScript has a finite width. Usually this is not apparent, but occasionally it will be. How should the end of a line look (i.e. be capped)? How should two lines join at a corner?

PostScript stores internally a variable `linecap` which controls how lines are capped. This table shows the effect:

`linecap`	Appearance of a thick line segment
0	
1	
2	

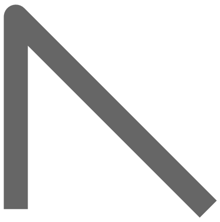
This variable is set with a command sequence like `1 setlinecap`. If it is not explicitly set, it is 0.

How lines join is controlled by an internal variable `linejoin`.

```
9 setlinewidth
% linejoin = 0 by default
newpath
0 0 moveto
0 72 lineto
72 0 lineto
stroke
```

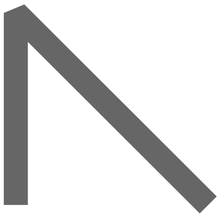
If I add a single line like the following, near the beginning, I get something very slightly different.

```
1 setlinejoin
```



Another possibility:

```
2 setlinejoin
```



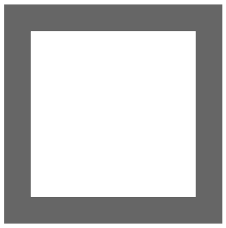
There is nothing wrong with any of these pictures, but in some circumstances you will want to use something other than the default, which is the first one. This is particularly true, for example, in drawing three dimensional figures.

Also, the effect of `closepath` may not be what you expect. Compare:

```
newpath
0 0 moveto
72 0 rlineto
0 72 rlineto
0 72 lineto
0 0 lineto
stroke
```

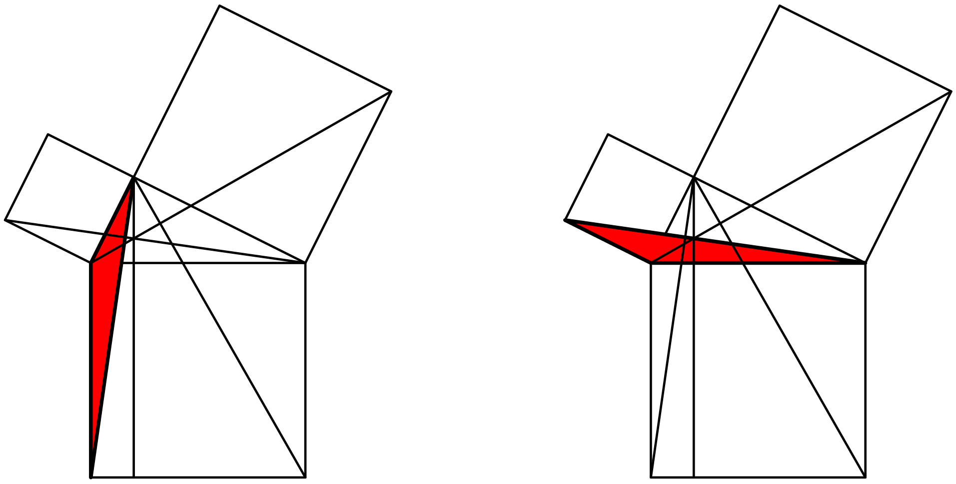
and

```
newpath
0 0 moveto
72 0 rlineto
0 72 rlineto
0 72 lineto
closepath
stroke
```



The moral of this is that if you mean to draw a closed path, then use `closepath`. It closes up the path into a seamless path—without break—so that all corners become essentially equivalent. As I shall likely repeat often, *programs should reflect concepts*—i.e. not depend on accidents to look OK.

Exercise 9. Draw in PostScript the following pictures, taken, with modifications, from the proof of Proposition I.47 in the standard edition of Euclid's *Elements*. Of course you might want to look up Euclid's argument first. One thing to keep in mind is that in drawing a complex figure, each conceptual component in the program should be handled as a unit. A triangle is not just a bunch of line segments, but line segments assembled in a particular order and style. You should also think here about about using colours in an intelligent way to help explain Euclid's proof.



Here and elsewhere, when you are asked to reproduce a picture, you are almost always expected to reproduce its dimensions as closely as you can.

One of the main purposes of this book is to get a reader used to the idea that pictures can and should play a larger role in the exploration and exposition of mathematics. Nothing like diving in at the deep end—you should take it as dogma here that

- *The ideal proof has no words and no labels on the figures.*

A good picture—well,a really good one—tells its story without these crutches. The point is that the reader should be able to tell what's going on without having to go back and forth between text and figure, and see in a glance what no number of words could express as clearly.

Of course the word 'proof' in these circumstances does not have the conventional meaning. Now a proof in mathematics is what it always has been—a rigourous arrangement of statements, each one following logically from previous ones. But in much mathematics, such an arrangement is often flat and without appropriate emphasis. It does not lay out clearly the important points to be dealt with, whereas pictures often do this superbly. The magic of geometry in mathematics, even at the most sophisticated level, is that geometrical concepts are somehow more . . . well, more *visible* than others.

Exercise 10. Draw a picture of the French flag (blue, white, and red vertical stripes). Do two versions, one with and one without thin black lines separating the different colours and outlining the flag.

I do not know the official aspect ratio, but you must try to find it out before you do this exercise. There are many sites on the 'Net that display country flags. One unlikely find

`http://www.immigration-usa.com/flags/`

Exercise 11. Try a Canadian flag, too. This is somewhat more difficult, but you ought to be able to produce a schematic maple leaf.

8. A trick for eliminating redundancy

I mentioned before that if we want to fill a path with one colour and then stroke it with another, it is not necessary to build the path a second time. Instead, we can use `gsave` and `grestore` in a tricky way. These two operators save and restore the **graphics state**, but the graphics state includes any current paths unfinished, as well as the current colour. Therefore we can do this:

```
newpath
144 144 moveto
144 0 rlineto
0 144 rlineto
-144 0 rlineto
closepath
gsave
1 0 0 setrgbcolor
fill
grestore
0 setgray
stroke
```

to get a red square outlined in black, and without any redundant code.

9. Summary

The basic commands necessary to draw in Postscript are pretty simple. The commands

```
newpath
moveto
lineto
rmoveto
rlineto
closepath
stroke
```

```
fill
```

are used to build and draw paths. The commands

```
translate
scale
rotate
```

allow us to make simple coordinate changes. The commands

```
setlinewidth
setrgbcolor
setgray
setlinejoin
setlinecap
```

allow us to change the attributes of paths to be drawn. Finally

```
gsave
grestore
showpage
```

lets us put together several pages of drawings.

These are almost all the basic drawing commands. They are pretty simple, and it might be difficult at this stage to understand how one can use them to draw anything complicated. Constructing interesting things to draw can take a lot of work. To help us here we have so far seen only a small set of mathematics functions like `cos`, etc. and the stack manipulation commands `dup`, `exch`, and `pop`. Next we need to learn how to get PostScript to do more sophisticated things for us.

Finally, remember that in PostScript you put data on the stack before you say what to do with those data.

References

1. **PostScript Language—Tutorial and Cookbook**, Adobe Systems, 1985. Available at

   `http://www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF`

This is easy and pleasant reading, with lots of intriguing examples.

2. **PostScript Language—Program Design**, Adobe Systems, 1985. Available at

   `http://www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF`

Not quite so useful as the tutorial, but still with useful ideas.

3. **PostScript Language Reference**, Adobe Systems, Third Edition, 1999. Available at

   `http://partners.adobe.com/asn/developer/technotes/postscript.html`

Invaluable. Huge, comprehensive, but nonetheless very readable.

4. Three good 'Net sources for PostScript are

   `http://www.prepressure.com/ps/whatis/PSlanguage.htm` (PostScript humour)

   `http://www.vergenet.net/ conrad/fractals/legobrot/` (PostScript LEGO)

   `http://cgm.cs.mcgill.ca/ luc/PSgeneral.html` (Luc Devroye's PostScript pages)