Mathematical Illustrations

A MANUAL OF GEOMETRY AND POSTSCRIPT

BILL CASSELMAN
University of British Columbia

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2005

Printed in Hong Kong, China

Typefaces Photina MT 10/13.5 pt. with ITC Symbol and Lucida Sans Typewriter
System LaTeX 2ε [TB]

A catalog record for this book is available from the British Library.

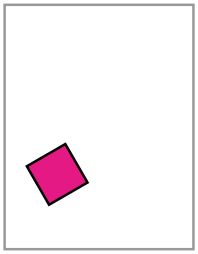
Library of Congress Cataloging in Publication Data
Casselman, Bill, 1941–
Mathematical illustrations : a manual of geometry and PostScript / Bill Casselman.
p. cm.
Includes bibliographical references and index.
ISBN 0-521-83921-1 (hardback) – ISBN 0-521-54788-1 (pbk.)
1. PostScript (Computer program language) I. Title.
QA76.73.P67C37 2004
005.13′3 – dc22 2004045886

ISBN 0 521 83921 1 hardback
ISBN 0 521 54788 1 paperback

In this book, the phrase, "PostScript interpreter" mean an interpreter of the PostScript language.

Contents

CHAPTER 1



Getting started in PostScript

In this book we use a program called **Ghostscript**, as well as one of several programs that in turn rely on Ghostscript running behind the scenes, to serve as our PostScript® interpreter and interface. All the programs we use are available without cost through the Internet. Be careful – the language we are writing our programs in is PostScript, and the program used to interpret them is Ghostscript. See Appendix 2 for how to acquire Ghostscript and set up your programming environment.

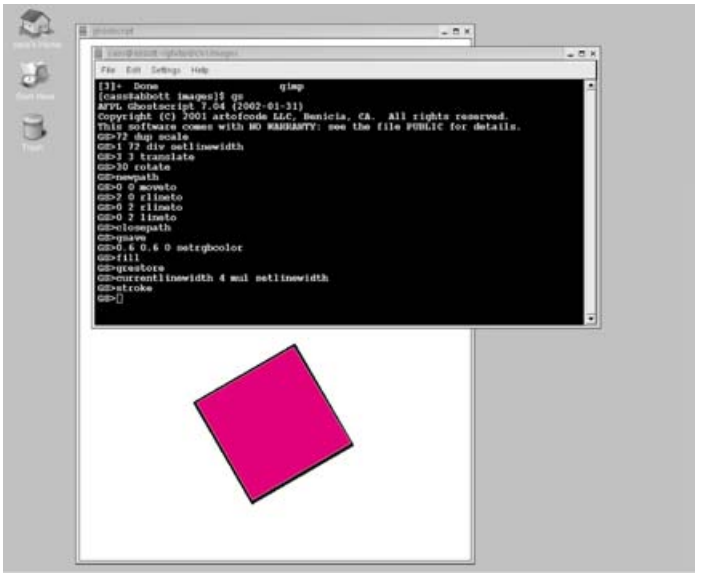
The interpreter Ghostscript has by itself a relatively primitive user interface that will turn out to be too awkward to use for very long, but learning this interface will give you a valuable feel for the way PostScript works. Furthermore, Ghostscript will continue to serve a useful although limited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

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

1.1 SIMPLE DRAWING

Start up Ghostscript. On Unix® networks this is usually done by typing `gs` in a terminal window, and on other systems it is usually done by clicking on the icon for Ghostscript. (You can also run Ghostscript in a terminal window – even on Windows® systems; see Appendix 2.) 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 will often call the **page**, opens up with a **default coordinate system**. The origin of this coordinate system on a page is at the lower left, and the unit of measurement, which is the same in both horizontal and vertical



The program Ghostscript running with image and terminal windows showing.

directions, is equal to a **point** exactly $1/72$ of an inch in length. (This **Adobe point** is almost, but not quite, the same as the classical **printer's point**, which measures 72.27 to an inch.) The size of the graphics window will probably be either letter size ($8.5'' \times 11''$ or 612×792 points^2) or the size of European A4 paper, depending on your locality. As we will see in a moment, the coordinate system can easily be 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:

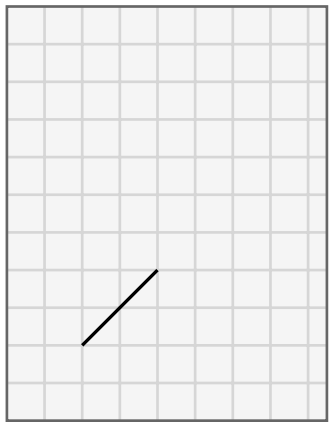
```
AFPL Ghostscript 7.04 (2002-01-31)
Copyright (C) 2001 artofcode LLC, Benicia, CA. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>
```

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

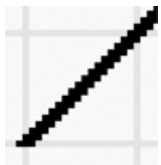
looks like:

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

GS>
```



(The grid is just there to help you orient yourself and is 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 is not one that this book will consider. So, in effect, in this book all lines will be assumed to be . . . well, lines – not things that look jagged and ugly – dare I say *pixellated*? – 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 will 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, and so you write down *first* the coordinates of the point to move to and *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 and then the command.
- So far you have just built your path. You draw it – that is, make it visible – with the command `stroke`. You have some choice over what color you can draw with, but the color that is used by default is black.

From now on I will usually leave the prompts `GS>` *out.* Let me repeat what I hope to be clear from this example:

- *PostScript always digests things backwards. The arguments to an operator always go before the operator itself.*

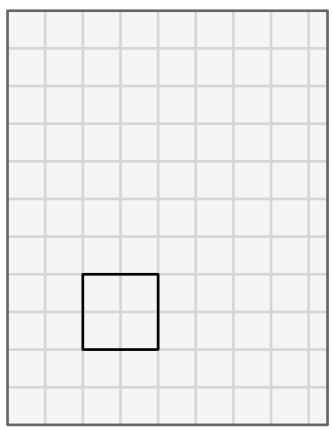
This convention is called **Reverse Polish Notation** (RPN). It will seem somewhat bizarre at first, but you'll get used to it. It is arguable that manual calculations, at least when carried out by those trained in European languages, should have followed RPN conventions instead of the ones used commonly in mathematics. It makes a great deal of sense to apply operations as you write from left to right.

RPN was devised by logicians for purely theoretical reasons, but PostScript is like this for practical reasons of efficiency. There is one great advantage from a user's standpoint: it allows a simple "cut and paste" style of programming.

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

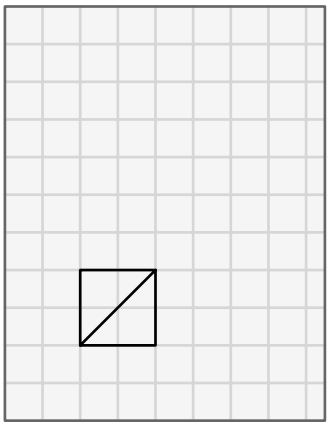
```
newpath
144 144 moveto
288 144 lineto
288 288 lineto
144 288 lineto
144 144 lineto

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'll tell you in Section 1.6 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 what has already been drawn.

There are often many 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 colors 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 gray square, and

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

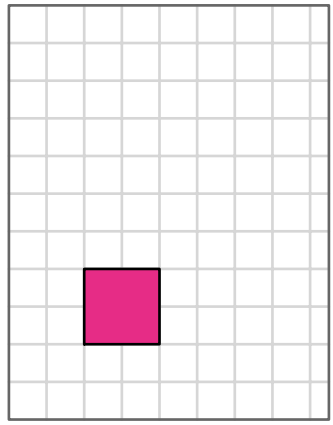
will make a red one. The `rgb` here stands for red, green, blue, and for each color you choose a set of three parameters between 0 and 1. Whenever you set a new color, it will generally persist until you change it again. Note that `0` is black and is `1` white. The command `x setgray` is the same as `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 1.1. *How would you set the current color to green? Pink? Violet? Orange?*

Filling or stroking a path normally deletes it from the record. So if you want to fill and stroke the same path, you have to be careful. One way of dealing with this is straightforward if tedious – just copy code. If you want to draw a red square with a black outline, you then 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
```



In Section 1.8 we will see a way to produce this figure without redundant typing.

EXERCISE 1.2. *Run Ghostscript. Draw an equilateral triangle near the center 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 will see in Section 1.4, you can get PostScript to do the calculations.)*

1.2 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, and so scaling by 72 is the same as scaling first by 8 and 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 by different factors, 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. Thus, scaling x by 2 and y by 1 has this effect on a square with a thick border:

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 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 that do not actually vanish. You should realize, however, that on a device of high resolution, such as a 1200 DPI printer, such lines will be nearly invisible. Setting the line width to 0 contradicts the general principle of **device independence** – *you should always aim in PostScript to produce figures that do not in any way depend directly on the particular device on which it will be reproduced.*

EXERCISE 1.3. *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

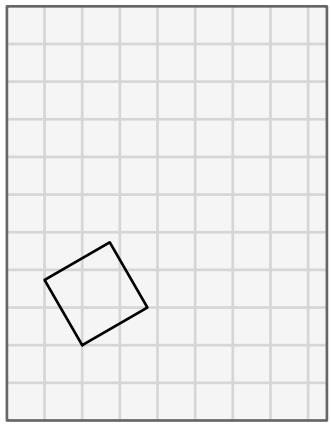
```
72 72 scale
4.25 5.5 translate
```

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

There is one more simple coordinate change: `rotate`.

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

stroke
```



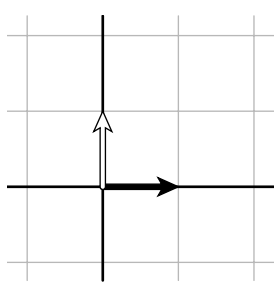
The translation is done first because rotation always takes place around the current origin. Note that *PostScript works with angles in degrees*. This will cause us some trouble later on, but for now it is probably A Good Thing.

EXERCISE 1.4. *Europeans use A4 paper. Find out its dimensions and show how to draw a square one centimeter on a side with its center in the middle of an A4 page. (Incidentally, what is the special mathematical property of A4 paper?)*

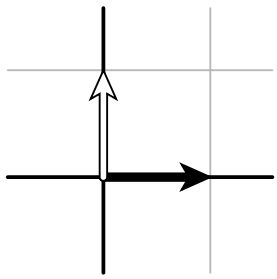
1.3 COORDINATE FRAMES

It is sometimes not quite so easy to predict the effect of coordinate changes. The secret to doing so is to think in terms of **coordinate frames**. Frames are associated to linear coordinate systems and vice versa. The way to visualize how the coordinate changes `scale`, `translate`, and `rotate` affect drawing is by realizing their effect on the frame of the coordinate system.

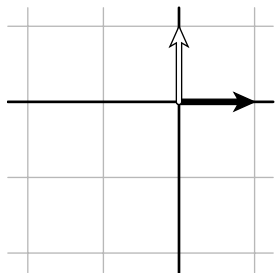
A simple frame, with units in centimeters



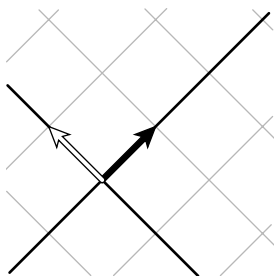
Scaled by $\sqrt{2}$ *in both directions*



Translated by [1, 1]



Rotated by 45°

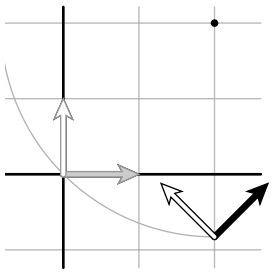


There are two fundamental things to keep in mind when wondering how coordinate changes affect drawing.

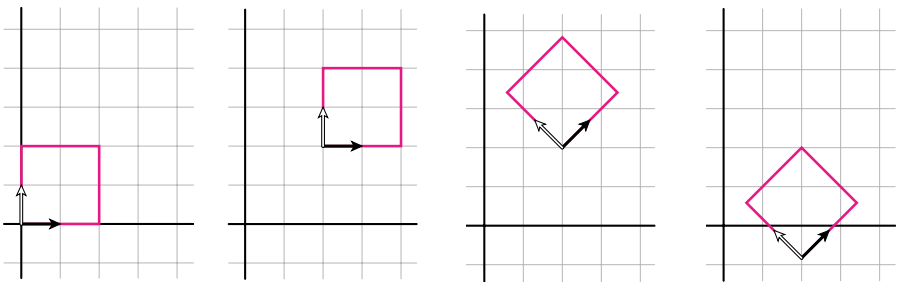
- *Coordinate changes affect the current frame in the natural and direct way. That is to say,* `2 2 scale` *scales the current frame vectors by a factor of* 2*, and so on.*
- *Drawing commands take effect relative to the current frame.*

For example, `rotate` always rotates the coordinate system around the current origin, which means that it rotates the current coordinate frame. The commands `translate`, `scale`, and `rotate`, when combined in the right fashion, can make any reasonable coordinate change you want (as well as a few you will probably never want). The restriction of "reasonability" here means those that in effect lay down

a grid of parallel lines on the plane. As an example, suppose you want to rotate your coordinate system around some point other than the origin. More explicitly, suppose you want to rotate by 45° around the point whose coordinates in the current system are $(2, 2)$. In other words, we want to move the current coordinate frame as at the right.



The way to get this is



In other words, to rotate the coordinate system by θ around the point P, you perform in sequence (1) translation by the vector $(P - O)$ from the origin O to P; (2) rotation by θ; (3) translation by $-(P - O)$.

The effect of "zooming in" is rather similar and is analyzed in Appendix 5.

1.4 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. As in many 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 seven significant places. This is where the roots of the language in graphics work show up, for normally drawing a picture on a page of reasonable size does not have to be extremely

accurate. This limited accuracy is not usually a serious 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 that grows and shrinks as a program moves 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`.)

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>
```

One thing to note here is that the number 5 is written as 5.0, which means that it is a real number, not an integer. PostScript generally treats integers differently from real numbers; only integers can be used as counters, for example. But it can't really tell that the square root of 25 is an integer.

EXERCISE 1.5. *Explain what the stack holds as the calculation proceeds.*

EXERCISE 1.6. *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
`m n idiv`	puts the integer quotient of m divided by n on the stack
`m n mod`	puts remainder after division of m by n 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)