PostScript Tutorial

Written by Paul Bourke
Original November 1990. Last updated December 1998

Translation into Bulgarian by Albert Ward.

Introduction

Postscript is a programming language that was designed to specify the layout of the printed page. Postscript printers and postscript display software use an interpreter to convert the page description into the displayed graphics.

The following information is designed as a first tutorial to the postscript language. It will concentrate on how to use postscript to generate graphics rather than explore it as a programming language. By the end you should feel confident about writing simple postscript programs for drawing graphics and text. Further information and a complete specification of the language can be obtained from **The Postscript Language Reference Manual** from Adobe Systems Inc, published by Addison-Wesley, Reading, Massachuchusetts, 1985.

Why learn postscript, after all, many programs can generate it for you and postscript print drivers can print to a file? Some reasons might be:

- Having direct postscript output can often result in much more efficient postscript, postscipt that prints faster than the more generic output from printer drivers.
- There are many cases where generating postscript directly can result in much better quality. For example when drawing many types of fractals where high resolution is necessary, being able to draw at the native high resolution of a postscript printer is desirable.
- It isn't uncommon for commercial packages to make errors with their postscript output. Being able to look at the postscript and make some sense of what is going on can sometimes give insight on how to fix the problem.

The Basics

Postscript files are (generally) plain text files and as such they can easily be generated by hand

or as the output of user written programs. As with most programming languages, postscript files (programs) are intended to be, at least partially, human-readable. As such, they are generally free format, that is, the text can be split across lines and indented to hilight the logical structure.
Comments can be inserted anywhere within a postscript file with the percent (%) symbol, the comment applies from the % until the end of the line.
While not part of the postscript specification the first line of a postscript file often starts as %!. This is so that spoolers and other printing software detect that the file is to interpreted as postscript instead of a plain text file. The inline example below will not include this but the postscript files linked from this page will include it since they are design for direct printing.
The first postscript command to learn is **showpage**, it forces the printer to print a page with whatever is currently drawn on it. The examples given below print on single pages and therefore there is a showpage at the end of the file in each example, see the comments later regarding EPS.

A Path

A path is a collection of, possibly disconnected, lines and areas describing the image. A path is itself not drawn, after it is specified it can be stroked (lines) or filled (areas) making the appropriate marks on the page. There is a special type of path called the clipping path, this is a path within which future drawing is constrained. By default the clipping path is a rectangle that matches the border of the paper, it will not be changed during this tutorial.

The Stack

Postscript uses a stack, otherwise known as a LIFO (Last In First Out) stack to store programs and data. A postscript interpreter places the postscript program on the stack and executes it, instructions that require data will read that data from the stack. For example, there is an operator in postscript for multiplying two numbers, **mul**. it requires two arguments, namely the two numbers that are to be multiplied together. In postscript this might be specified as

```
10 20 mul
```

The interpreter would place 10 and then 20 onto the stack. The operator **mul** would remove 20 and then 10 from the stack, multiply them together and leave the result, 200, on the stack.

Coordinate system

Postscript uses a coordinate system that is device independent, that is, it doesn't rely on the

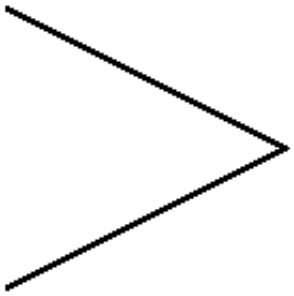
resolution, paper size, etc of the final output device. The initial coordinate system has the x axis to the right and y axis upwards, the origin is located at the bottom left hand corner of the page. The units are of "points" which are 1/72 of an inch long. In other words, if we draw a line from postscript coordinate (72,72) to (144,72) we will have a line starting one inch in from the left and right of the page, the line will be horizontal and be one inch long.

The coordinate system can be changed, that is, scaled, rotated, and translated. This is often done to form a more convenient system for the particular drawing being created.

Basic Drawing Commands

Time to draw something. The following consists of a number of operators and data, some operators like **newpath** don't need arguments, others like **lineto** take two arguments from the stack. All the examples in this text are shown as postscript on the left with the resulting image on the right. The text on the left also acts as a link to a printable form of the postscript file.

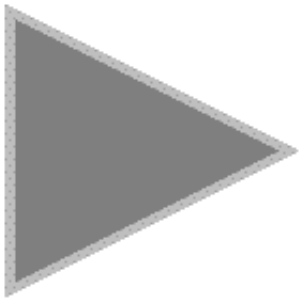
```
newpath
100 200 moveto
200 250 lineto
100 300 lineto
2 setlinewidth
stroke
```



There are also a relative moveto and lineto commands, namely, **rmoveto** and **rlineto**.

In this next example a filled object will be drawn in a particular shade, both for the outline and the interior. Shades range from 0 (black) to 1 (white). Note the **closepath** that joins the first vertex of the path with the last.

```
newpath
100 200 moveto
200 250 lineto
100 300 lineto
closepath
gsave
0.5 setgray
fill
grestore
4 setlinewidth
0.75 setgray
stroke
```



The drawing commands such as **stroke** and **fill** destroy the current path, the way around this is to use **gsave** that saves the current path so that it can be reinstated with **grestore**.

Text

Text is perhaps the most sophisticated and powerful aspect of postscript, as such only a fraction of its capabilities will be discussed here. One of the nice things is that the way characters are placed on the page is no different to any other graphic. The interpreter creates a path for the character and it is then either stroked or filled as per usual.

```
/Times-Roman findfont
12 scalefont
setfont
newpath
100 200 moveto
(Example 3) show
```

Example 3

As might be expected the position (100,200) above specifies the position of the bottom left corner of the text string. The first three lines in the above example are housekeeping that needs to be done the first time a font is used. By default the font size is 1 point, scalefont then sets the font size in units of points (1/72 inch). The brackets around the words "Example 3" indicate that it is a string.

A slightly modified version of the above uses **charpath** to treat the characters in the string as a path which can be stroked or filled.

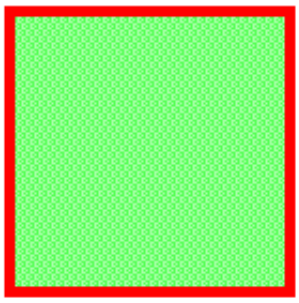
```
/Times-Roman findfont
32 scalefont
setfont
100 200 translate
45 rotate
2 1 scale
newpath
0 0 moveto
(Example 4) true charpath
0.5 setlinewidth
0.4 setgray
stroke
```

You should make sure you understand the order of the operators above and the resulting orientation and scale of the text, procedurally it draws the text, **scale** the y axis by a factor of 2, **rotate** counter clockwise about the origin, finally **translate** the coordinate system to (100,200).

Colour

For those with colour LaserWriters the main instruction of interest that replaces the **setgray** is previous examples is **setrgbcolor**. It requires 3 arguments, the red-green-blue components of the colour each varying from 0 to 1.

```
newpath
100 100 moveto
0 100 rlineto
100 0 rlineto
0 -100 rlineto
-100 0 rlineto
closepath
gsave
0.5 1 0.5 setrgbcolor
fill
grestore
1 0 0 setrgbcolor
4 setlinewidth
stroke
```

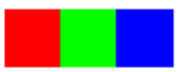


Programming

As mentioned in the introduction postscript is a programming language. The extend of this language will not be covered here except to show some examples of procedures that can be useful to simplify postscript generation and make postscript files smaller.
Lets assume one needed to draw lots of squares with no border but filled with a particular colour. One could create the path repeatedly for each one, alternatively one could define something like the following.

```
/csquare {
   newpath
   0 0 moveto
   0 1 rlineto
   1 0 rlineto
   0 -1 rlineto
   closepath
   setrgbcolor
   fill
} def

20 20 scale
```



```
5 5 translate
1 0 0 csquare

1 0 translate
0 1 0 csquare

1 0 translate
0 0 1 csquare
```

This procedure draws three coloured squares next to each other, each 20/72 inches square, note the **scale** of 20 on the coordinate system. The procedure draws a unit square and it expects the RGB colour to be on the stack. This could be used as a method (albeit inefficient) of drawing a bitmap image.

Even if one is simply drawing lots of lines on the page, in order to reduce the file size it is common to define a procedure as shown below. It just defines a single character "l" to draw a line segment, one can then use commands like 100 200 200 200 l" to draw a line segment from (100,200) to (200,200).

```
/l { newpath moveto lineto stroke } def
```

Some other useful Commands

The following are some other commonly used commands along with a brief description, again you should consult a reference manual for the entire set of commands.

arc Draws an arc (including a circle). The arguments are xcenter, ycenter, radius, start angle, stop angle. The arc is drawn counterclockwise, the angles are in units of degrees.

currentpoint This is an example of an instruction that takes no arguments but leaves numbers on the stack, namely the coordinates of the current point.

setdash This sets the dash attribute of a line in terms of a mark-space array. Just as strings are denoted by round braces (), arrays are denoted by square braces []. For example the following command "[3 3] 0 setdash" would make any following lines have a 3 unit dash followed by a 3 unit space. The argument after the dash array is the offset for the start of the first dash.

setlinecap This specifies what the ends of a stroked line look like. It takes one argument which may be 0 (butt caps), 1 (round caps), or 2 (extended butt caps). The radius of round caps and the extension of the butt caps is determined by the line thickness.

```
/LINE {
   newpath
   0 0 moveto
   100 0 lineto
   stroke
} def

100 200 translate
10 setlinewidth 0 setlinecap 0 setgray LINE
1 setlinewidth 1 setgray LINE

0 20 translate
10 setlinewidth 1 setlinecap 0 setgray LINE
1 setlinewidth 1 setgray LINE

0 20 translate
10 setlinewidth 2 setlinecap 0 setgray LINE
1 setlinewidth 1 setgray LINE
```

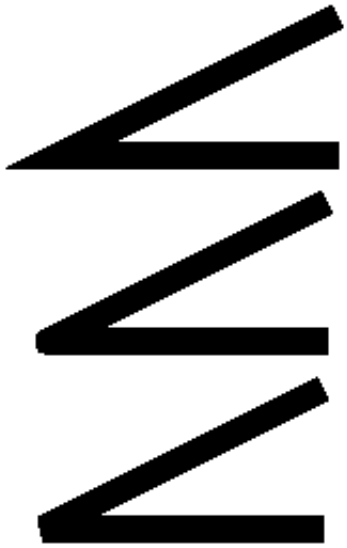
setlinejoin This determines the appearance of joining lines. It takes one argument which may be 0 (miter join), 1 (round join), or 2 (bevel join).

```
/ANGLE {
   newpath
   100 0 moveto
   0 0 lineto
   100 50 lineto
   stroke
} def

10 setlinewidth
0 setlinejoin
100 200 translate
ANGLE

1 setlinejoin
0 70 translate
ANGLE

2 setlinejoin
0 70 translate
ANGLE
```



curveto This draws a bezier curve through the three points given as arguments. The curve starts at the first point, end at the last point, and the tangents are given by the line between the first-second and second-third pair.

save and **restore** Instead of having to "undo" changes to the graphics state it is possible using **save** to push the entire graphics state onto the stack and then reinstate it later with a **restore**.

Drawing "small" Images

Printing images using postscript is somewhat more involved than the graphics discussed so far. Examples of two image types will be presented, grey scale images (which could also be used for black and white images) and 24 bit RGB images.

8 Bit Grey Scale

The simplest method that is applicable to small images is the postscript command **image**. This take 5 arguments: the width and height of the image, the bits per pixel, a transformation matrix (6 numbers) and finally a procedure for acquiring the image data. In the following example, the grey scale (8 bit) image is 24 pixels wide by 34 pixels tall. Since in postscript an image is defined from the lower left corner, and the image is defined from its top left corner, the transformation matrix used here does the appropriate vertical flipping. The procedure used here is esentially a null procedure, it is a hexadecimal string as indicated by the angle brackets <>. Postscript renders images as one point square hence the scaling by 24 horizontally and 34 vertically. There are many details left out of this discussion, the reader needs to consult a postscript reference manual for more information.

```
100 200 translate
26 34 scale
26 34 8 [26 0 0 -34 0 34]
{<
ffffffffffffffffffffffffffffffffffffffffffffffff
ff000000000000000000000000000000000fffffffffffff
ff00efefefefefefefefefefefefefefef0000fffffffffff
ff00efefefefefefefefefefefefefefef00ce00fffffffff
ff00efefefefefefefefefefefefefefef00cece00ffffffff
ff00efefefefefefefefefefefefefefef00cecece00ffffff
ff00efefefefefefefefefefefefefefef00cecececе00ffff
ff00efefefefefefefefefefefefefefef00000000000000ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efef000000ef000000ef000000ef0000ef0000efefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efef000000ef00000000ef00000000ef000000efefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efef0000ef00000000000000ef000000ef0000efefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefefefefef00ff
```

```
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff00efefefefefefefefefefefefefefefefefef00ff
ff000000000000000000000000000000000000000000000000ff
ffffffffffffffffffffffffffffffffffffffffffffffffff
>}
image
```

24 Bit RGB Colour

RGB images with 8 bits per pixel can be represented in postscript using the command **colorimage** which is very similar to the **image** command. In the following example the image is 32 pixels wide by 38 pixels tall.

```
100 200 translate
32 38 scale
32 38 8 [32 0 0 -38 0 38]
{<
1dfb0023fb002afb0031fb0037fb003ffb00
66fb006cfb0073fb0079fb0081fb0086fb00
adfb00b5fb00bbfb00c3fb00c8fb00cffb00
23f5002af50031f50037f5003ff50044f500

          ...cut...

3807003f08004508004c0800520800590800
8108008608008d07009308009a0700a20800
c90800d00800d60800dd0800e40700ea0700
>}
false 3 colorimage
```

What is EPS?

EPS (Encapsulated PostScript) is normal postscript with a few restrictions and a few comments in a specified format that provides more information about the postscript that follows. It was design to make it easier for applications to include postscript generated elsewhere within their own pages. The full specification can be obtained from Adobe but in order to make a postscript file DSC (Adobe's Document Structuring Convention) compliant the following must be true:

- There shouldn't be a **showpage**, since EPS is designed to be included inside other documents a **showpage** would obviously ruin the intended effect. In reality most programs that import EPS redefine **showpage** so that if it does exist it doesn't cause problems, a common definition is "/showpage { } def"

- The file should consist of just one page.
- The first line of the file should be "%!PS-Adobe EPSF-3.0"
- There must be a correctly formed BoundingBox comment, this looks like
  %%BoundingBox: xmin ymin xmax ymax
  and tells application that plans to include the postscript how large the image is.
- The file should not use any operators that change the global drawing state. In particular the following command may not be used:

```
banddevice      exitserver    initmatrix     setshared
clear           framedevice   quit           startjob
cleardictstack  grestoreall   renderbands    copypage
initclip        setglobal     initgraphics   setpagedevice
erasepage       nulldevice    sethalftone    setscreen
setgstate       setmatrix     settransfer    undefinefont
```

- The stack must be left EXACTLY in the same state at the end of the EPS file as it was at the start of the EPS file.
- The lines in EPS files cannot exceed 255 characters in length.

Perhaps most importantly, since usually an application that supports postscript file insertion doesn't have the full postscript interpreter, an EPS file generally has a preview image associated with it. The application dealing with the EPS file can display the preview in the user interface giving a better idea what will print. It should be noted that EPS previews are one of the more machine/OS dependent aspects of EPS.

Frequently Used Comments

Comments can of course be be added anywhere and they will be ignored by the interpreter. There are some standard comments the most common of which are be listed below. The text within the square brackets should be replaced with the appropriate text for the file in which they appear (without the []).

```
%!PS-Adobe-3.0 EPSF-3.0
%%Creator: [generally the program that generated the postscript]
%%Title: [descriptive name or just the file name]
%%CreationDate: [date the file was created]
%%DocumentData: Clean7Bit
%%Origin: [eg: 0 0]
%%BoundingBox: xmin ymin xmax ymax
%%LanguageLevel: 2 [could be 1 2 or 3]
%%Pages: 1
%%Page: 1 1
%%EOF
```

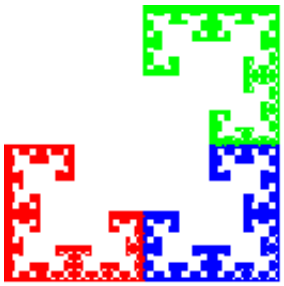
Drawing "large" images

Due to line length and other restrictions, turning 'large" bitmaps into postscript requires a modification to the methods discussed earlier. The following will describe the most general case of representing a 24 bit RGB colour image as an EPS file. While inefficient this can also be used for greyscale and even black and white images. In the following code "width" and "height" should be replaced with the numbers appropriate to the image.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Creator: someone or something
%%BoundingBox: 0 0 width height
%%LanguageLevel: 2
%%Pages: 1
%%DocumentData: Clean7Bit
width height scale
width height 8 [width 0 0 -height 0 height
{currentfile 3 width mul string readhexstring pop} bind
false 3 colorimage

...hexadecimal information cut...

%%EOF
```



The modifications for greyscale images are quite simple, change the line

```
{currentfile 3 width mul string readhexstring pop} bind
```

to

```
{currentfile width string readhexstring pop} bind
```

and of course only write one hexadecimal number (representing the grey level of the pixel) for each pixel of the image. This technique should work for images of any size.

Paper sizes

Paper Size	Dimension (in points)
Comm #10 Envelope	297 x 684
C5 Envelope	461 x 648
DL Envelope	312 x 624
Folio	595 x 935
Executive	522 x 756
Letter	612 x 792
Legal	612 x 1008
Ledger	1224 x 792
Tabloid	792 x 1224
A0	2384 x 3370
A1	1684 x 2384
A2	1191 x 1684

A3	842 x 1191
A4	595 x 842
A5	420 x 595
A6	297 x 420
A7	210 x 297
A8	148 x 210
A9	105 x 148
B0	2920 x 4127
B1	2064 x 2920
B2	1460 x 2064
B3	1032 x 1460
B4	729 x 1032
B5	516 x 729
B6	363 x 516
B7	258 x 363
B8	181 x 258
B9	127 x 181
B10	91 x 127