GETTING STARTED WITH POSTSCRIPT

Martin Erickson

January 26, 2002

The purpose of this introductory guide is to help you begin using PostScript (PS), a page description language with which you can create beautiful, precise graphics. Although Postscript can produce entire pages, interweaving text and pictures, we focus on Encapsulated PostScript (EPS), which is used especially for producing figures for inclusion in other documents. This tutorial contains simple examples for you to try. It covers the basics, certainly enough to get you started. To learn more about PostScript, you may want to consult the sources listed in Section 8 and in the references that follow.

Contents

1 What is PostScript?

PostScript (PS), created in 1985 by Adobe Systems, Inc., is a printing and imaging system in which you can produce high quality, beautiful graphics. PostScript is the default graphics language for UNIX/LINUX operating systems. PostScript is an interpreted, vector-based graphics language, with many small commands. PostScript is also a programming language,

containing the usual programming constructs such as procedures, arrays, and loops. PostScript files can be produced with any text editor, e.g., NotePad. Output can be viewed with viewers such as the ones in PCTeX and Ghostscript.

The name PostScript is, in part, a play on words (postscript of a letter) and an indication of the postfix order of operations in PostScript, which is in turn intimately related to the main data structure in the PostScript language. This data structure is a stack. To understand PostScript, you must understand the stack.

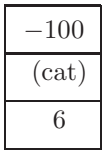
2 How to use the stack

The basic data structure in PostScript is a stack, which we can think of as being like of a stack of books. The last item placed on the stack is the first item removed. The stack contains operands which operators work on. The stack concept ties in nicely with the convention of postfix notation.

Suppose that we list the following objects: 6, `(cat)`, and −100.

```
6 (cat) -100
```

Then the interpreter stores the objects on the stack as pictured here.



(Notice that the objects read from left to right appear on the stack from bottom to top.)

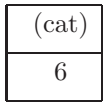
Three useful operators

The `pop` operator removes the top element from the stack. The `exch` operator interchanges the top two elements of the stack. The `dup` operator duplicates the top element of the stack.

For example, starting with the stack from before, we remove the top element.

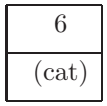
```
pop
```

Now the stack looks like this.



Next, we interchange the top two elements of the stack.

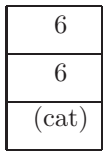
```
exch
```



Finally, we duplicate the top element of the stack.

```
dup
```

And now the stack looks like this.

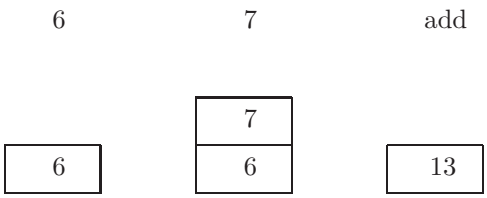


Arithmetic and other basic operations

The arithmetic operations addition, subtraction, multiplication, and division are performed via the operators `add`, `sub`, `mul`, and `div`. For example, let us add two numbers, 6 and 7.

```
6 7 add
```

The 6 is placed on the stack, followed by the 7. The operator `add` takes the top two numbers off the stack (6 and 7) and places their sum (13) on the stack.



Other mathematical functions are calculated in a similar manner. For example,

```
45 sin
```

yields the sine of 45 degrees.

3 How to make simple pictures

In this section and the following sections, we show several examples of PostScript figures. Not every command in these examples is explained. But it is hoped that the reader will gain an understanding of what is possible to achieve with PostScript as well as some concrete ideas to try out.

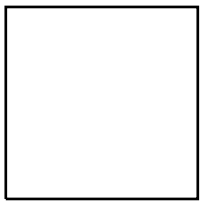


Figure 1: A square.

Coordinates

PostScript pictures are based on coordinate geometry. The origin is at the lower left corner; x increases as you move to the right and y increases as you move up. One unit equals 1 "big point"; that is, 72 units equal 1 inch.

We begin by describing how to draw lines. To draw a line, begin a new drawing path with `newpath`. Then move the drawing instrument to a point with `moveto`. Then create a line with `lineto`. Finally, fill the path with ink with `stroke`.

Example 3.1. We use basic commands to produce a square of side length 1 inch.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Square
%%BoundingBox: 0 0 144 144

% draw square
newpath
36 36 moveto
108 36 lineto
108 108 lineto
36 108 lineto
36 36 lineto
stroke
```

The picture is shown in Figure 1.

Note. The `%` (percentage) sign is used for comments. Certain comments give structural information about a file. In our code above, the first three lines are an identification, a title, and a bounding box. (Your printer may be able to accept merely `%!` as the identification line.) The bounding box gives coordinates for the lower left corner and upper right corner of the picture.

Note. There is some flexibility in writing expressions. For example, the commands

```
36 36 moveto 108 36 lineto
```

and

```
108 36 36 36 moveto lineto
```

and even

```
108 36 36 moveto 36 lineto
```

all produce the same action.

"Freehand drawings"

It is at times convenient to make "freehand drawings," i.e., ones in which correct coordinates are guessed. If, upon viewing, things are a little bit off, they can be rectified in a revision by "smudging," i.e., making small changes. (Many viewers help by showing coordinates of the currrent mouse location.)

Example 3.2. We illustrate with a drawing involving three circles and seven lines. (The diagram is shown in Figure 2.)

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Monge's theorem
%%BoundingBox: 0 0 234 234

0.5 setlinewidth

% draw the circles
newpath 65 65 50 0 360 arc stroke
newpath 125 145 25 0 360 arc stroke
newpath 165 90 10 0 360 arc stroke

% draw the lines
newpath 3 80 moveto 189 229 lineto stroke
newpath 93 1 moveto 188 233 lineto stroke
newpath 15 124 moveto 195 95 lineto stroke
newpath 75 10 moveto 195 100 lineto stroke
newpath 60 167 moveto 197 49 lineto stroke
newpath 125 210 moveto 195 45 lineto stroke
newpath 185 234 moveto 192 43 lineto stroke
```

Note. The operator `arc` takes five arguments: coordinates of the center of the arc, radius of the arc, and beginning and ending reference angles of the arc.

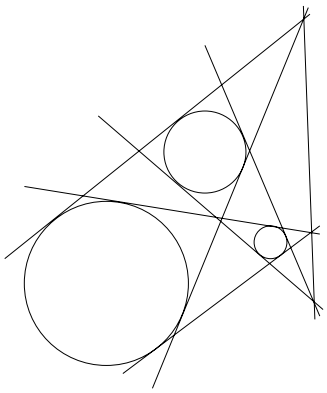


Figure 2: An illustration of Monge's theorem.

Procedures and variables

A procedure is defined by a command such as

```
/p {operations} def
```

(this command defines a procedure called `p` that performs `operations`). Procedures accept their arguments on the stack.

A variable is similar to a procedure but without the `{}` symbols.

Example 3.3. We make a series of shaded squares using a procedure. The squares are produced with relative `moveto` and `lineto` commands called, respectively, `rmoveto` and `rlineto`. The squares are filled in with the `fill` command. In the black-and-white world, `setgray` defines a gray level from 0 to 1, where 0=*black* and 1=*white*. Notice that each level of gray is opaque, completely covering any graphical elements beneath it.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Shaded squares
%%BoundingBox: 0 0 198 144

/inch {72 mul} def

/sidelength 1 inch def

% square procedure
/square {
  sidelength 0 rlineto
  0 sidelength rlineto
  sidelength neg 0 rlineto
  0 sidelength neg rlineto
} def
```

Figure 3: Shaded squares.

```
% make first square
newpath
0.5 inch 0.5 inch moveto square
0 setgray
fill

% make second square
newpath
0.75 inch 0.5 inch moveto square
0.25 setgray
fill

% make third square
newpath
1 inch 0.5 inch moveto square
0.5 setgray
fill

% make fourth square
newpath
1.25 inch 0.5 inch moveto square
0.75 setgray
fill
```

The result is shown in Figure 3.

Clipping

Clipping allows us to create a closed path that serves as a "universe" for all further graphics insertions.

Example 3.4. We create a Venn diagram using a clipping path.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Venn diagram
%%BoundingBox: 0 0 144 144

0.9 setgray

% draw left shaded circle
newpath 72 90 36 0 360 arc fill

% draw right shaded circle
newpath 108 90 36 0 360 arc fill

% draw bottom shaded circle
newpath 90 54 36 0 360 arc fill

0.75 setgray

gsave

% make clipping path of left circle
newpath 72 90 36 0 360 arc clip

% draw shaded intersection of left and right circles
newpath 108 90 36 0 360 arc fill

grestore

gsave

% make clipping path of left circle
newpath 72 90 36 0 360 arc clip

% draw shaded intersection of left and bottom circles
newpath 90 54 36 0 360 arc fill

grestore

gsave

% make clipping path of right circle
```

```
newpath 108 90 36 0 360 arc clip

% draw shaded intersection of right and bottom circles
newpath 90 54 36 0 360 arc fill

grestore

0.5 setgray

gsave

% make clipping path of intersection of left and right circles
newpath 72 90 36 0 360 arc clip
newpath 108 90 36 0 360 arc clip

% draw shaded intersection of all circles
newpath 90 54 36 0 360 arc fill

grestore

0 setgray

2 setlinewidth

% draw circumference of left circle
newpath 72 90 36 0 360 arc stroke

% draw circumference of right circle
newpath 108 90 36 0 360 arc stroke

% draw circumference of bottom circle
newpath 90 54 36 0 360 arc stroke
```

The diagram is shown in Figure 4.

Note. The operator `gsave` saves the current graphics state (including current path, current point, font, and coordinate system), and `grestore` restores the graphics state to the last saved state.

4 How to add text to pictures

Text is inserted into a picture by applying a `show` operator to a text string. But first, a font must be chosen using `findfont`, `scalefont`, and `setfont` operators. For example, the command

```
/Courier findfont 12 scalefont setfont
```

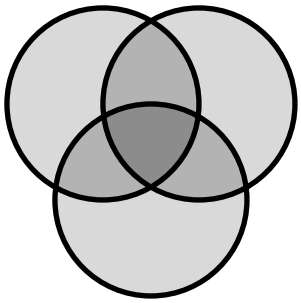


Figure 4: Venn diagram.

chooses a Courier font of size 12 points. (The `findfont` operator puts the Courier font on the stack. This is followed on the stack by 12. The `scalefont` operator takes the font and the 12 and replaces them on the stack with a scaled font. Finally, this scaled font is made the current font by the `setfont` operator.)

Labels

Centering of text is a factor to keep in mind when creating labels. In the following example, we deal with this issue by defining a centering procedure.

Example 4.1. We put labels on the square of Example 3.1. The result is shown in Figure 5.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Square with labels
%%BoundingBox: 0 0 144 144

72 72 translate

% draw square
newpath
-36 -36 moveto
36 -36 lineto
36 36 lineto
-36 36 lineto
-36 -36 lineto
stroke

% make labels
/Courier findfont 12 scalefont setfont
/center {dup stringwidth pop 2 div neg dup rmoveto} def
```

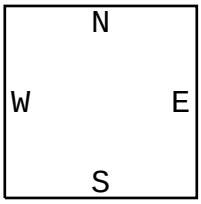


Figure 5: A labeled square.

```
0 30 moveto (N) center show
0 -30 moveto (S) center show
30 0 moveto (E) center show
-30 0 moveto (W) center show
```

5 How to use programming constructs

PostScript supports the usual programming constructs such as loops and arrays

Loops

PostScript allows for loops via a `for` construction. The arguments for this operator are

```
min increment max {procedure}
```

(causing values to be placed on the stack starting with `min`, incrementing by `increment`, until reaching `max`).

Example 5.1. The following program uses a loop to graph a function by "plotting points."

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Function plot
%%BoundingBox: 0 0 350 150

% define function, interval endpoints, and magnification
/function {180 mul 3.14 div sin} def
/leftendpoint 0 def
/rightendpoint 6.28 def
/magnification 50 def
```

```
% make transformations
magnification dup scale
1 magnification div setlinewidth
0.5 1.5 translate

% draw coordinate axes
newpath
-0.5 0 moveto 6.5 0 lineto
0 -1.5 moveto 0 1.5 lineto
stroke

% draw ticks
newpath
-0.1 -1 moveto 0.1 -1 lineto
-0.1 1 moveto 0.1 1 lineto
3.14 -0.1 moveto 3.14 0.1 lineto
3.14 2 mul -0.1 moveto 3.14 2 mul 0.1 lineto
stroke

% make labels
/Courier findfont 12 magnification div scalefont setfont
-0.25 0.95 moveto (1) show
-0.40 -1.05 moveto (-1) show
0.1 -0.25 moveto (0) show
3.25 1.25 moveto (y = sin x) show
/Symbol findfont 12 magnification div scalefont setfont
3.15 -0.25 moveto (\160) show
6.25 -0.25 moveto (2\160) show

% plot function
/point {0.5 magnification div 0 360 arc} def
leftendpoint 0.01 rightendpoint {newpath dup function point fill} for
```

The graph is shown in Figure 6.

Note. You can create similar programs for plotting parametric functions, functions in polar coordinates, and even implicitly defined curves.

The definition of polynomials can be effected particularly well using Horner's method.

Arrays

PostScript allows for creation and manipulation of arrays. An array of size n is indexed as `[0 1 2 ... n-1]`. Its values are assigned and retrieved with the `put` and `get` operators, respectively. An array's entries can be any objects, e.g., numbers, strings, even other arrays.

Example 5.2. We create a complete graph on 17 vertices.

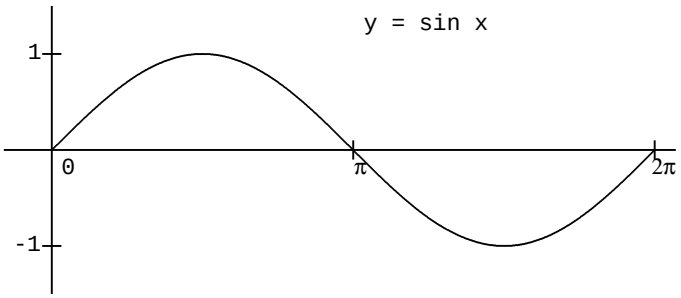


Figure 6: A sine curve.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Complete graph of order 17
%%BoundingBox: 0 0 180 180

% define constants and array
/radius 72 def
/angle 360 17 div def
/vertex 17 array def

% make transformations
90 90 translate
0.5 setlinewidth

% define vertices
0 1 16 {
  /i exch def
  vertex i [i angle mul cos radius mul i angle mul sin radius mul] put
} for

% draw vertices
0 1 16 {
  /i exch def
  newpath vertex i get 0 get vertex i get 1 get 2 0 360 arc fill
} for

% draw edges
newpath
0 1 16 {
  /i exch def
  i 1 add 1 16 {
    /j exch def
    vertex i get 0 get vertex i get 1 get moveto
```

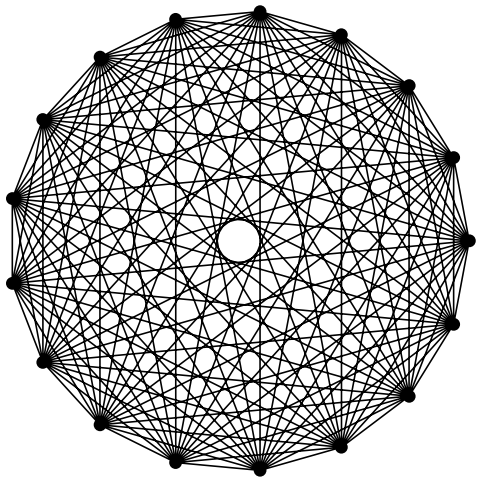


Figure 7: A complete graph of order 17.

```
    vertex j get 0 get vertex j get 1 get lineto
  } for
} for
stroke
```

The graph is shown in Figure 7.

The above example has symmetry that allows us to write the code easily. A "generic graph" can be specified by a list of vertices and a list of edges.

Example 5.3. We create a "generic graph."

```
%!
%%Title: generic graph
%%BoundingBox: 0 0 216 216

% specify number of vertices and edges
/numberofvertices 12 def
/numberofedges 8 def

% define arrays
/vertex numberofvertices array def
/edge numberofedges array def
```

```
% define vertices
vertex 0 [50 50] put
vertex 1 [100 100] put
vertex 2 [150 50] put
vertex 3 [75 200] put
vertex 4 [25 75] put
vertex 5 [50 200] put
vertex 6 [75 175] put
vertex 7 [175 25] put
vertex 8 [150 175] put
vertex 9 [200 100] put
vertex 10 [100 225] put
vertex 11 [150 125] put

% define edges
edge 0 [2 4] put
edge 1 [2 11] put
edge 2 [3 5] put
edge 3 [4 1] put
edge 4 [2 5] put
edge 5 [3 10] put
edge 6 [4 9] put
edge 7 [6 7] put

% draw vertices
  0 1 numberofvertices 1 sub {
  /i exch def
  newpath vertex i get 0 get vertex i get 1 get 2 0 360 arc fill
} for

% draw edges
0 1 numberofedges 1 sub {
  /i exch def
  newpath
  vertex edge i get 0 get get 0 get vertex edge i get 0 get get 1 get moveto
  vertex edge i get 1 get get 0 get vertex edge i get 1 get get 1 get lineto
  stroke
} for
```

The graph is shown in Figure 8.

6 How to add color to pictures

Color is defined with the `setrgbcolor` operator. This operator takes a triple of numbers which denote an additive mixture of red, green, and blue color light. The numbers are

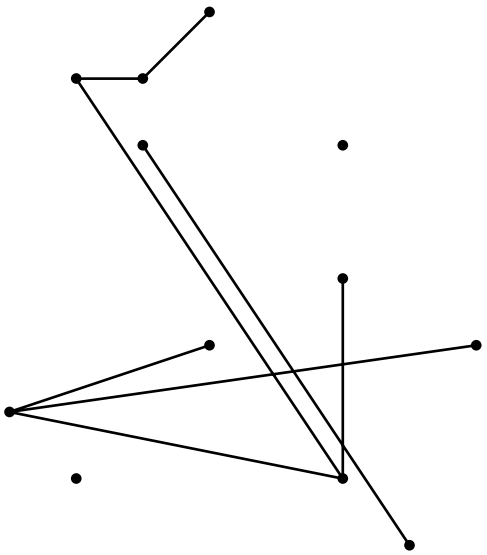


Figure 8: A "generic graph."

between 0 and 1, with 0 denoting no light of the given color and 1 denoted the maximum amount of light. For example, `0 0 1 setrgbcolor` corresponds to pure blue.

Example 6.1. We make a color wheel.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Color wheel
%%BoundingBox: 0 0 270 270

% make transformations
144 144 translate
30.0 setlinewidth

% draw green--red third
0 1 3 {
  /i exch def
  /lambda i 4 div def
  lambda 1 lambda sub 0 setrgbcolor
  newpath 90 0 moveto 108 0 lineto stroke
  30 rotate
} for

% draw red--blue third
0 1 3 {
```

```
  /i exch def
  /lambda i 4 div def
  1 lambda sub 0 lambda setrgbcolor
  newpath 90 0 moveto 108 0 lineto stroke
  30 rotate
} for

% draw blue--green third
0 1 3 {
  /i exch def
  /lambda i 4 div def
  0 lambda 1 lambda sub setrgbcolor
  newpath 90 0 moveto 108 0 lineto stroke
  30 rotate
} for

% make labels
/Courier findfont 12 scalefont setfont
/centeringoffset {dup stringwidth pop 2 div neg} def
0 setgray
30 rotate 0 120 moveto (red) centeringoffset 0 rmoveto show
-120 rotate 0 120 moveto (green) centeringoffset 0 rmoveto show
-120 rotate 0 120 moveto (blue) centeringoffset 0 rmoveto show
```

The result is shown in Figure 9.

Note. Please keep in mind that a color wheel is but a one-dimensional slice of the three-dimensional space of color combinations in the rgb model.

7 How to produce complex pictures

We show a few more examples of PostScript figures.

Logic and branching

PostScript allows for conditional branching via an `if` construction.

Example 7.1. We create a chessboard with algebraic notation, using loops, branching, color, and a procedure.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Chessboard
%%BoundingBox: 0 0 180 180

% square procedure
/square {
```

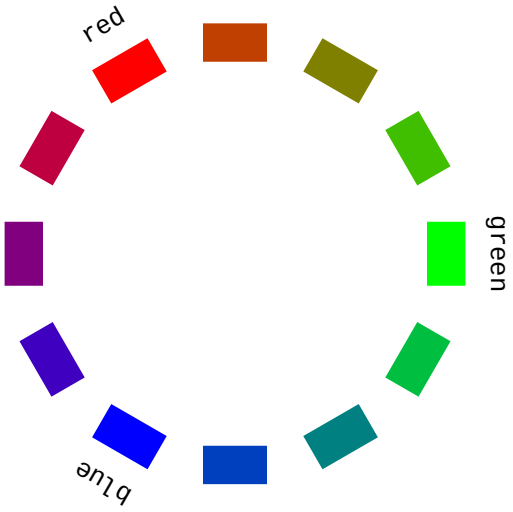


Figure 9: A color wheel.

```
  /sidelength exch def
  sidelength 0 rlineto
  0 sidelength rlineto
  sidelength neg 0 rlineto
  0 sidelength neg rlineto
} def

% draw chessboard
1 1 8 {
  /i exch def
  1 1 8 {
    /j exch def
    i 2 mod j 2 mod eq {0 0.25 0 setrgbcolor} if % dark green
    i 2 mod j 2 mod ne {0.95 0.95 0.40 setrgbcolor} if % light yellow
    newpath i 20 mul j 20 mul moveto 20 square fill
  } for
} for

% make labels
/Courier findfont 12 scalefont setfont
0 setgray
(a) (b) (c) (d) (e) (f) (g) (h) 165 -20 25 {5 moveto show} for
(1) (2) (3) (4) (5) (6) (7) (8) 165 -20 25 {5 exch moveto show} for
```

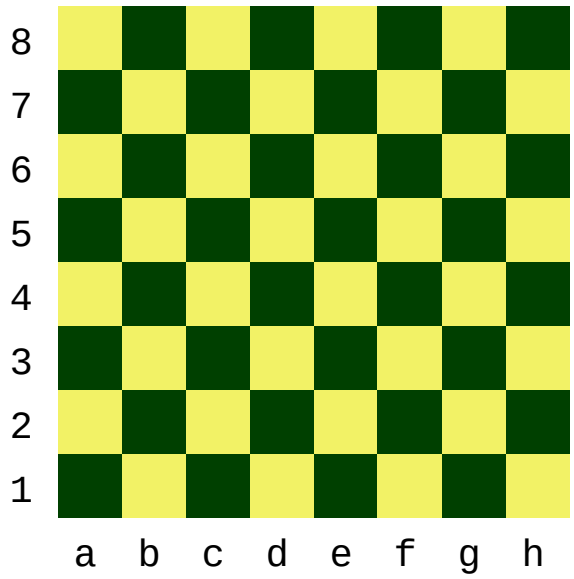


Figure 10: A chessboard with algebraic notation.

The finished product is shown in Figure 10.

Affine transformations

In PostScript, we can make convenient affine transformations, i.e., translations, rotations, scalings, and reflections, of the plane. The common operators for the first three of these operations are `translate`, `rotate`, and `scale`. The general operator is `concat`, which takes six arguments defining the affine transformation.

Example 7.2. We draw an integrated circuit known as a full adder.

```
%!PS-Adobe-3.0  EPSF-3.0
%%Title: Full adder
%%BoundingBox: 0 0 504 216

% AND gate procedure
/andgate {
  currentpoint currentpoint translate
  2 {
    [1 0 0 -1 0 0] concat
    0 10 moveto 10 10 lineto
    10 0 moveto 10 15 lineto
    25 15 lineto
    40 15 40 0 15 arcto pop pop pop pop
```

```
    50 0 lineto
  } repeat
  neg exch neg exch translate
} def

% OR gate procedure
/orgate {
  currentpoint currentpoint translate
  2 {
    [1 0 0 -1 0 0] concat
    0 10 moveto 14 10 lineto
    16 0 moveto
    16 10 12 15 25 arcto pop pop pop pop
    31 15 41.2 -5.5 30 arcto pop pop pop pop
    50 0 lineto
  } repeat
  neg exch neg exch translate
} def

% NOT gate procedure
/notgate {
  currentpoint currentpoint translate
  2 {
    [1 0 0 -1 0 0] concat
    0 0 moveto 10 0 lineto
    10 15 lineto
    35 0 lineto
    35 2.5 37.5 2.5 2.5 arcto pop pop pop pop
    40 2.5 40 0 2.5 arcto pop pop pop pop
    50 0 lineto
  } repeat
  neg exch neg exch translate
} def

% junction procedure
/junction {currentpoint 2 0 360 arc} def

% make labels
/Courier findfont 12 scalefont setfont
0 185 moveto (input1) show
0 165 moveto (input2) show
3 85 moveto (carry) show
455 145 moveto (output) show
455 50 moveto (nextcarry) show

% draw logic gates
```

```
newpath
75 175 moveto orgate
200 165 moveto andgate
300 155 moveto orgate
400 145 moveto andgate
75 110 moveto andgate
125 110 moveto notgate
300 75 moveto andgate
350 75 moveto notgate
175 40 moveto andgate
225 50 moveto orgate
stroke

% draw connections
newpath
50 185 moveto 75 185 lineto
50 165 moveto 75 165 lineto
125 175 moveto 200 175 lineto
250 165 moveto 300 165 lineto
350 155 moveto 400 155 lineto
65 100 moveto 75 100 lineto
175 110 moveto 200 110 lineto
50 85 moveto 300 85 lineto
125 60 moveto 225 60 lineto
100 30 moveto 175 30 lineto
275 50 moveto 450 50 lineto
275 65 moveto 300 65 lineto
65 185 moveto 65 100 lineto
75 165 moveto 75 120 lineto
175 175 moveto 175 50 lineto
200 155 moveto 200 110 lineto
275 165 moveto 275 65 lineto
300 145 moveto 300 85 lineto
400 135 moveto 400 75 lineto
125 110 moveto 125 60 lineto
100 85 moveto 100 30 lineto
stroke

% draw junctions
newpath 65 185 moveto junction fill
newpath 75 165 moveto junction fill
newpath 175 175 moveto junction fill
newpath 275 165 moveto junction fill
newpath 125 110 moveto junction fill
newpath 100 85 moveto junction fill
newpath 300 85 moveto junction fill
```

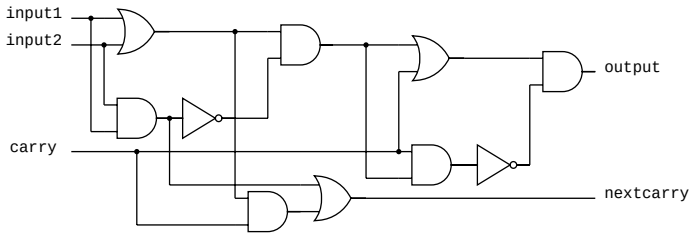


Figure 11: A full adder.

The finished product is shown in Figure 11.

Recursion

PostScript allows for procedural recursion.

Example 7.3. We create Sierpiǹski's triangle using recursion.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Sierpinski's triangle
%%BoundingBox: 0 0 200 200

% Sierpinski triangle procedure
/sierpinskitriangle {
  /n exch def
  /y3 exch def /x3 exch def
  /y2 exch def /x2 exch def
  /y1 exch def /x1 exch def

  % fill triangle black
  0 setgray
  newpath
  x1 y1 moveto
  x2 y2 lineto
  x3 y3 lineto
  closepath fill

  % fill center triangle white
  1 setgray
  newpath
  x1 x2 add 2 div y1 y2 add 2 div moveto
  x2 x3 add 2 div y2 y3 add 2 div lineto
  x3 x1 add 2 div y3 y1 add 2 div lineto
```

```
  closepath fill

  % call this function recursively while n>0 to do the subtriangles
  n 0 gt {
    x1 y1
    x1 x2 add 2 div y1 y2 add 2 div
    x1 x3 add 2 div y1 y3 add 2 div
    n 1 sub

    x2 y2
    x1 x2 add 2 div y1 y2 add 2 div
    x2 x3 add 2 div y2 y3 add 2 div
    n 1 sub

    x3 y3
    x1 x3 add 2 div y1 y3 add 2 div
    x2 x3 add 2 div y2 y3 add 2 div
    n 1 sub

    sierpinskitriangle
    sierpinskitriangle
    sierpinskitriangle
  } if
} def

% GO!
0 0 100 173 200 0 8 sierpinskitriangle
```

The picture is shown in Figure 12.

Note that all the black fills except the first are superfluous. (We could step outside the recursion to fill the outer triangle black.)

8 How to learn more

There are many aspects of PostScript that are not discussed in this introduction, such as word art and dictionaries. Here are some resources for you to investigate to learn more.

The definitive books about PostScript are the "Blue Book" (tutorial) [3], the "Red Book" (reference manual) [2], and the "Green Book" (programming guide) [1]. Some good beginning books (in addition to the "Blue Book") are [6], [5], and [4].

A good way to learn PostScript is to use it to make figures. You might try producing a figure illustrating, for instance, the Fano Configuration, the Desargues Configuration, a tiling of Poincaré's hyperbolic plane, or an approximation to a Peano area-filling curve.

For a foray into connections between PostScript and geometry, please see "A Manual for Mathematical PostScript" at

<http://www.sunsite.ubc.ca/DigitalMathArchive/Graphics/text/www/>.

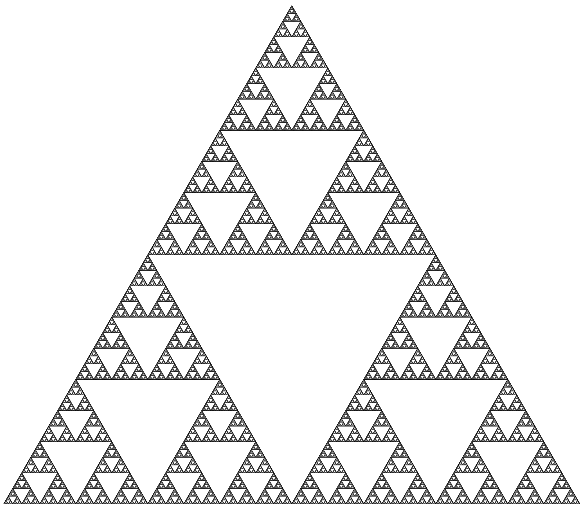


Figure 12: Sierpiǹski's triangle.

References

[1] Adobe Systems Incorporated. *PostScript Language: Program Design*. Addison–Wesley, Reading, 1985.

[2] Adobe Systems Incorporated. *PostScript Language: Reference Manual*. Addison–Wesley, Reading, 1985.

[3] Adobe Systems Incorporated. *PostScript Language: Tutorial and Cookbook*. Addison–Wesley, Reading, 1985.

[4] H. McGilton and M. Campione. *PostScript by Example*. Addison–Wesley, Reading, 1992.

[5] G. C. Reid. *Thinking in PostScript*. Addison–Wesley, Reading, 1990.

[6] R. Smith. *PostScript: A Visual Approach*. Peachpit Press, Berkley, 1990.