FREE eBook

LEARNING postscript

Free unaffiliated eBook created from
Stack Overflow contributors.

#postscript

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: postscript

It is an unofficial and free postscript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official postscript.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with postscript

Remarks

PostScript is a reverse-polish stack-based, dynamically-typed, dynamic-namespacing, scripting language with built-in primitives for generating rendered images from vector descriptions. PostScript employs the same "Adobe Image Model" as the PDF file format.

PostScript is used as an output format by many programs since it is designed to be easily machine-generated.

Like LISP, PostScript is *homoiconic* and code and data share the same representation. Procedures can take procedures as data and yield procedures as results, lending itself to techniques from *concatenative-programming* as well.

Examples

Installation or Setup

The authentic Adobe PostScript interpreters are available in high-end printers, the Display PostScript (DPS) product, and the Acrobat Distiller product. As authors of the standard, these products are considered "the standard implementation" for the purpose of describing differences among PostScript implementations.

The Standard interface to the interpreter defined in the PLRM is the *program-stream* which may be either text or binary depending upon the details of the underlying channel or OS/controller. Acrobat Distiller has a GUI front-end to select the input postscript program and render its output as a pdf. Distiller also has some limited support for using the output text stream for reporting errors and other program output. GSView provides a similar GUI front-end for a similar workflow using Ghostscript as the interpreter.

Ghostscript and Xpost both work in a command-line mode. The postscript program file to run can be mentioned on the command-line (`gs program.ps` or `xpost program.ps`) which will open a graphics window to display the graphical output. Options may be used to render the graphics somewhere else like a disk file or suppress the graphics entirely and use postscript just as a text scripting language.

The various interpreters each have their own installation and setup instructions and it would be wasteful (and prone to falling out-of-date) to reproduce them here.

Freely-available PostScript interpreters

- Ghostscript is available for all major platforms and Linux distributions, in source or binary form, under the GNU license or under other license arrangements with the authors, Artifex software. Ghostscript implements the full PostScript 3 standard.

- Xpost is available in source form for all major platforms, under the BSD-3-clause license. It implements the Level-1 standard with some Level-2 extensions and some DPS extensions.

General Description of PostScript

PostScript is a Turing-complete general programming language, designed and developed by Adobe Systems. Many of the ideas which blossomed in PostScript had been cultivated in projects for Xerox and Evans & Sutherland.

Its main real-world application historically is as a *page description language*, or in its single-page EPS form a vector-graphics image-description language. It is dynamically-typed, dynamically-scoped, and stack-based which leads to a mostly Reverse Polish syntax.

There are three major releases of PostScript.

1. **PostScript Level 1** — this was released to the market in 1984 as the resident operating system of the Apple LaserWriter laser printer, inaugurating the Desktop Publishing Era.
2. **PostScript Level 2** — released in 1991, this contained several important improvements to Level 1, including support for image decompression, in-RIP separation, auto-growing dictionaries, garbage collection, Named Resources, binary encodings of the PostScript program stream itself.
3. **PostScript 3** — the latest and perhaps most widely adopted version was released in 1997. It too contains several import improvements over Level 2 such as Smooth Shading. The term “level” has been dropped.

Though PostScript is typically used as a page description language -- and therefore is implemented inside many printers to generate raster images -- it can also be used for other purposes. As a quick reverse-polish calculator with more memorable operator names than `bc`. As an output format generated by another program (usually in some other language).

Though PostScript file are typically 7-bit-clean ASCII, there exist several kinds of binary encoding described in the level 2 standard. And being programmable, a program may implement its own arbitrarily-complex encoding scheme for itself. There is an International Obfuscated Postscript Competition, somewhat less active than the C one.

Online References

- Index Pages of Adobe Documentation:
  https://www.adobe.com/products/postscript/resources.html
  http://www.adobe.com/devnet/postscript.html
  http://www.adobe.com/devnet/font.html

- PostScript Language Reference Manual, 3ed - The PostScript 3 standard. (7.41MB pdf) (Supplement, Errata)

- PostScript Language Reference Manual, 2ed - The PostScript Level 2 standard. (includes Display PostScript documentation.) (3.29MB pdf)

- Postscript Tutorial and Cookbook - The blue book. (847KB pdf)
- Postscript Language Program Design - The green book. (911KB pdf)
- Thinking in Postscript - By the author of the green book and the blue-book's tutorial. (826KB pdf)
- PostScript Language Document Structuring Conventions Specification 3.0 (521KB pdf)
- Adobe Type 1 Font Format (444KB pdf)
- Encapsulated PostScript File Format Specification 3.0 (185KB pdf)
- PostScript Printer Description File Format Specification 4.3 (186KB pdf) (Update)
- Troubleshoot PostScript errors - Debugging tips. (158KB html)
- Acumen Journal - Archive of Postscript and PDF programming articles. (html directory of zipped pdfs)
- Mathematical Illustrations: A Manual of Geometry and Postscript - by Bill Casselman. (html directory of pdf chapters and code downloads)
- Thread with many sorting algorithm implementations (usenet archive)
- Don Lancaster's Guru Pages
- Anastigmatix's Direct use of the Postscript Language
- Open-source step-wise Debugger for Postscript Code

FAQs

- Usenet FAQ (circa 1995)
- Wikibooks PostScript FAQ
- SO PostScript questions sorted by most-frequently viewed

Books

- Postscript Language Reference Manual, 1ed, 1985. Recommended for its small size, and easy operator index from the summary pages (missing from later editions).
- Real World Postscript. Chapters by various authors on various topics, including excellent coverage of halftoning.

Local namespaces for functions

Postscript is a dynamic-namespacing or *LISP 1* language. But it provides the tools to implement

local variables in procedures and other effects needed to implement algorithms.

For local names in a procedure, make a new dictionary at the start and pop it at the end.

```
/myproc {
    10 dict begin
    %... useful code ...
    end
 } def
```

You can also combine this nicely with a shortcut to define the function's arguments as variables.

```
% a b c  myproc  result
/myproc {
    10 dict begin
    {/c /b /a} {exch def} forall
    %... useful code yielding result ...
    end
 } def
```

If you need to update a *"global" * variable while the local dictionary is on top, use `store` instead of `def`.

Hello World example

Select a font and fontsize, select location, `show` string.

```
%!PS
/Palatino-Roman 20 selectfont
300 400 moveto
(Hello, World!) show
showpage
```

Notes and common pitfalls:

- Failing to set a font (resulting in either no text or a default (ugly) font)
- Using `findfont` and `setfont` but forgetting to `scalefont` in between. Using the level-2 `selectfont` avoids this problem and is more concise.
- Failing to set a point with `moveto`, or setting the point outside of the page. For US letter paper 8.5x11 is 792x612 ps points. So it's easy to remember roughly 800x600 (but a smidge shorter and wider). So `300 400` is roughly the center of the page (little high, little left).
- Forgetting to call `showpage`. If you preview a ps program with `gs` and it does not end in `showpage`, `gs` may display an image for you. And yet, the file will mysteriously fail to produce any output when you try to convert to pdf or something else.

Curriculum

Read the documentation in this order to easily learn postscript:

1. Paul Bourke's excellent tutorial: http://paulbourke.net/dataformats/postscript/

2. Blue Book, first half, the original official tutorial:
   http://www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF

3. Green Book, how to use postscript effectively:
   http://www-cdf.fnal.gov/offline/PostScript/GREENBK.PDF

4. Thinking in Postscript, 'nuff said: http://wwwcdf.pd.infn.it/localdoc/tips.pdf

5. Mathematical Illustrations. Start small, build big. The math behind Bezier Curves. The Hodgman-Sutherland polygon clipping algorithm. Affine transformations and *non-linear* transformations of the path. 3D drawing and Gouraud shading. From the preface:

   > Which [of the many tools to help one produce mathematical graphics] to choose apparently involves a trade-off between simplicity and quality, in which most go for what they perceive to be simplicity. The truth is that the trade-off is unnecessary — once one has made a small initial investment of effort, by far the best thing to do in most situations is to write a program in the graphics programming language PostScript. There is practically no limit to the quality of the output of a PostScript program, and as one acquires experience the difficulties of using the language decrease rapidly. The apparent complexity involved in producing simple figures by programming in PostScript, as I hope this book will demonstrate, is largely an illusion. And the amount of work involved in producing more complicated figures will usually be neither more nor less than what is necessary.

Read Getting started with postscript online: https://riptutorial.com/postscript/topic/5616/getting-started-with-postscript

Chapter 2: Error Handling

Syntax

- { *-code-* } stopped { *-error-* }{ *-no-error-* } ifelse % error catching frame
- $error /errorname get % stackunderflow typecheck rangecheck etc
  $error /newerror get % bool. put false to deactivate error
  $error /ostack get % copy of operand stack at point of error
- errordict /stackoverflow { *-additional-code-* /stackoverflow signalerror} put
  % execute additional code on types of errors (here, the /stackoverflow error).

Remarks

There are two levels to error handling in postscript. This dichotomy applies both to the way the interpreter handles errors as well as the means available to the user (programmer) to control the handling.

The lower level is an unusual control structure `stop ... stopped`. `stopped` behaves much like a looping construct in that it establishes a mark on the execution stack that can be jumped-to if the `exit` operator (for a loop) or `stop` operator (for a `stopped`-context) is called. Unlike a looping construct, `stopped` yields a Boolean on the stack indicating whether `stop` was called (otherwise the procedure passed to `stopped` is known to have executed to completion.

When a PostScript error occurs, like `stackunderflow` maybe, the interpreter looks up the error's name in `errordict` which lives in `systemdict`. If the user has not replaced the procedure in `errordict`, then the default error procedure will take snapshots of all the stack and place them in `$error`, another dictionary in `systemdict`. Finally, the default procedure will call `stop` which pops the user program from the exec stack and executes the interpreter's error printing procedure called `handleerror` in `errordict`.

So using all of this knowledge, you can *catch* errors by wrapping a section of code in `{ ... } stopped`. You can *rethrow* an error by calling `stop`. You can determine what type of error occurred with `$error /errorname get`.

You can also change the default behavior for a specific type of error by replacing the procedure with that name in `errordict`. Or change the format of printing the error report by replacing `/handleerror` in `errordict`.

Examples

Is there a currentpoint?

Yield `true` on the stack if `currentpoint` executes successfully, or `false` if it signals a `/nocurrentpoint`

error.

```
{currentpoint pop pop} stopped not  % bool
```

Sequence of events when an error is signaled

The sequence for an error is usually:

1. error is triggered by looking up the error name in `errordict` and executing this procedure.
2. the `errordict` procedure calls `signalerror`, passing it the error name.
3. `signalerror` takes snapshots of the stacks, saving the snapshots in `$error`, and then calls `stop`.
4. `stop` pops the exec stack until the nearest enclosing stopped context established by the stopped operator.
5. if the program has not established its own stopped context to catch the error, it will be caught by an outer-level `stopped { errordict /handleerror get exec } if` which was called by the startup code to bracket the whole user program.
6. `handleerror` uses the information in `$error` to print an error report.

Signalling (throwing) an error

Most of the tools are standardized with the exception of the name of the operator to throw an error. In Adobe interpreters, it is called `.error`. In ghostscript, it is called `signalerror`. So with this line you can use `signalerror` in postscript code for Adobe interpreters or ghostscript or xpost.

```
/.error where {pop /signalerror /.error load def} if
```

commandname errorname **signalerror** —
Take snapshots of the stack in `$error`, then `stop`.

Eg.

```
% my proc only accepts integer
/proc {
    dup type /integertype ne {
        /proc cvx /typecheck signalerror
    } if
    % ... rest of proc ...
} def
```

Catching an error

Since the final action of the default error handler is to call `stop`, you can catch errors from operators by enclosing code in a `{ ... } stopped` construct.

```
{
    0 array
    1 get
} stopped {
```

```
    $error /errorname get =
} if
```

will print "`rangecheck`", the error signaled by `get` when the index is outside the acceptable range for that array.

Re-throwing errors

This snippet implements a procedure which behaves like a postscript looping operator. If the user `proc` calls `exit`, it catches the `invalidexit` error to fix the dictstack for the `end` at the end. Any other error except `invalidexit` is re-thrown by calling `stop`.

```
% array n proc  .  -
% Like `forall` but delivers length=n subsequences produced by getinterval
/fortuple { 4 dict begin
    0 {offset proc n arr} {exch def} forall
    /arr load length n idiv
    {
        {
            /arr load offset n getinterval
            [ /proc load currentdict end /begin cvx ] cvx exec
            /offset offset n add def
        } stopped {
            $error /errorname get /invalidexit eq
            { 1 dict begin exit }{ stop } ifelse
        } if
    } repeat
end
} def

%[ 0 1 10 {} for ] 3 {} fortuple pstack clear ()=
```

Read Error Handling online: https://riptutorial.com/postscript/topic/6199/error-handling

Chapter 3: Path Construction

Examples

Drawing (describing) a polygon

This example attempts to mimic the behavior of the built-in path construction operators like `arc`.

If there is a current point, `poly` first draws a line to (x,y)+(r,0), otherwise it starts by moving to that point.

Instead of `gsave` ... `grestore` (which has the undesirable effect of discarding the very changes to the current path which we want), it saves a copy of the current transformation matrix (CTM) as it exists when the function starts.

Then it does `lineto` to each succeeding point, which by scaling and rotating the matrix is always at (0,1). Finally, it calls `closepath` and then restores the saved matrix as the CTM.

```
% x y n radius  poly  -
% construct a path of a closed n-polygon
/poly {
    matrix currentmatrix 5 1 roll  % matrix x y n radius
    4 2 roll translate             % matrix n radius
    dup scale                      % matrix n
    360 1 index div exch           % matrix 360/n n
    0 1 {lineto currentpoint moveto}stopped{moveto}if   % start or re-start subpath
    {                              % matrix 360/n
        dup rotate                 % matrix 360/n
        0 1 lineto                 % matrix 360/n
    } repeat                 % matrix 360/n
    pop                      % matrix
    closepath                % matrix
    setmatrix                %
} def
```

Iterating through a path

This snippet dumps the contents of the current path to stdout. It uses the ghostscript procedure `=only` which may not be available on all interpreters. An equivalent procedure on Adobe interpreters is called `=print`.

`pathforall` is a looping operator which takes 4 procedure bodies as arguments which are called for the specific types of path elements, the result of `moveto`, `lineto`, `curveto`, `closepath`, and all other path-contruction operators which boil-down to these elements.

```
{ exch =only ( ) print =only ( ) print /moveto =}
{ exch =only ( ) print =only ( ) print /lineto =}
{ 6 -2 roll exch =only ( ) print =only ( ) print
  4 2 roll exch =only ( ) print =only ( ) print
  exch =only ( ) print =only ( ) print /curveto =}
```

```
{ /closepath = }
pathforall
```

Graph Paper

```
/in {72 mul} def
/delta {1 in 10 div} def
/X 612 def
/Y 792 def
0 delta Y {
    0 1 index X exch  % i 0 X i
    moveto exch       % 0 i
    lineto
    stroke
} for
0 delta X {
    0 1 index Y  % i 0 i Y
    moveto       % i 0
    lineto
    stroke
} for
showpage
```

Read Path Construction online: https://riptutorial.com/postscript/topic/6679/path-construction

Credits

S. No	Chapters	Contributors
1	Getting started with postscript	Community, Kurt Pfeifle, luser droog
2	Error Handling	luser droog
3	Path Construction	luser droog