Inessential PostScript*

(Revision: 0.14)

DRAFT COPY
BLEED ON ME!

The Student Information Processing Board
Kevin Iga
Tom Yu

August 6, 1992

Contents

List of Figures

1 Introduction

1.1 What is PostScript?

PostScript is a versatile page description language primarily used to communicate with printers and other graphics output devices. In particular, it is the language understood by the public laser printers on Athena, the Digital LN03R and the Digital LPS40. PostScript was developed by Adobe Systems Incorporated as a standardized device-independent page description language. As a result, a document written in PostScript will look almost exactly the same on any device capable of interpreting and creating images from PostScript.

Devices called *raster devices* are what are normally used to display PostScript. Raster devices display images by forming them in a grid of rectangular *pixels* (picture elements). Some examples of raster devices are a computer monitor, or a laser printer. Raster devices form the image of pixels by scanning horizontal lines of pixels, and vertically covering an area with these horizonal lines to form a grid of pixels.

1.2 Programs that output PostScript

Many programs are available that will write PostScript programs for you. For printing plain text, there is enscript. For writing formatted text, you can use LaTeX together with `dvips` or `dvi2ps`, `scribe`, `ez`, `troff`, and others. For drawing diagrams, `idraw` and `xfig` are available. For drawing graphs, there are xmath and matlab. Although you may not have heard of these programs before, they all can generate PostScript programs which will then be printed out, thus sparing you the chore of writing in raw PostScript.

1.3 Why program in raw PostScript?

So why would anyone want to program in PostScript directly? Often, you will find that unnecessary, because all of the programs mentioned earlier allow you to do most of the work that you will want to see on paper in PostScript. They make it easy, and most of the time those things are what you need to do. But every once in a while, you will want to make a fancy poster with shadowed letters. Or you will want to make a map with words in many different orientations. Or you will want to draw perspective line drawings according to some mathematical formula. Or, for some reason, the software out there just cannot get the effect you want on paper. Or maybe you just want to impress your friends.

There are several good ways to view PostScript. PostScript is a powerful computer programming language, just as genuine and powerful as C or Fortran. It is a tool for producing images, both on a video screen and of paper. It is an art form, allowing you to express yourself in ways that Leonardo da Vinci would never have dreamed of.

1.4 What this document does and does not offer

This document will teach you a few of the basics of PostScript, especially with regard to Athena. It does not promise to make you a wizard, although after a short time some people may believe you are one.

If you have never used Athena before, you should probably read *Getting Started on Athena*[1], available from Graphic Arts (in the basement of Building 10), and the *Inessential Guide to Athena*[2], available in the SIPB office. You should also have a little fun on Athena, just so you are comfortable with the environment. You might also try some of the programs that format files in PostScript to become familiar with printing on Athena.

There is no way that this document can tell you everything about PostScript. There already exist many good books on PostScript, and this document will make no effort to replace any of them. On the other hand, this document will explain enough of PostScript to enable you to do a number of simple, yet visually impressive displays.

2 Previewing PostScript

2.1 `gs`

Available on Athena is a program called Ghostscript, which is an interactive PostScript previewer. To run it, enter the following commands in an xterm window:

```
athena% add gnu
athena% gs
```

A large blank white preview window will appear, and the following message will appear *in your original xterm window*.

```
Initializing... done.
Ghostscript 2.3 Copyright (C) 1990, 1991 Aladdin Enterprises,
   Menlo Park, CA.  All rights reserved.
Distributed by Free Software Foundation, Inc.
Ghostscript comes with NO WARRANTY: see the file COPYING for details.
GS>
```

The `GS>` is the prompt for the Ghostscript interpreter. This interpreter will behave just like the PostScript interpreter in a typical PostScript laser printer in many ways, only at a much lower resolution (i.e. somewhat "chunkier"). You can type PostScript commands at the `GS>` prompt. To exit Ghostscript, you type `quit` at the `GS>` prompt. To run a file that is on disk under Ghostscript, you can type the following:

[1] This document is provided for no charge.

[2] This, and all other SIPB documentation, is available at no charge

```
GS>(foo.PS) run
```

to run the file `foo.PS`. You can also start up Ghostscript with a filename, as in `gs foo.PS`.

One of the many good things about Ghostscript is that it is interactive. You can type commands directly at it and it will behave almost exatcly like a PostScript printer will, executing the commands as it receives them. This way, you can experiment with PostScript without depleting your print quota or the Amazon rain forest. There are a few subtle differences (besides the resolution) between the output that you will see on the preview window of Ghostscript and the output that you will see if you actually send the commands to the printer. These will be explained as we come to them.

2.2 `dxpsview`

A PostScript previewer which is much superior to Ghostscript, is `dxpsview`. This program has two major disadvantages. It is not interactive, and it only runs on DECstations. You can start it up simply by typing the command `dxpsview&` in an xterm. A window will appear, and you can use the menus there to select a file to preview. One of the major advantages is that its rendering of fonts is much more accurate, since the actual font rendering code is provided by Adobe.

3 Printing out PostScript

On athena, the most common way to print files is by using the `lpr` command. For example, to print out the file `foo.PS` to the printer `bias`, you would use the following command:

```
athena% lpr -Pbias foo.PS
```

Now you probably know that you can send plain text to the printer in exactly the same way. You may now wonder how the printer tells the difference between plain text to be excuted and a PostScript program to be run. Actually, it turns out that the program which actually talks to the printer (called the *printer daemon*) checks the first two characters of the file. If the first two characters of the file are `%!`, often pronounced “percent bang”, then the daemon sends the file off to the printer without changes. If, on the other hand, the file does not start with `%!`, the daemon runs the file through a simple program which converts it into PostScript that the printer can then understand. The characters `%!` will be ignored by the printer, since any line that begins with the `%` character is a comment in PostScript.

As a result, any PostScript that you write and expect to be printed properly as a PostScript page description must begin with the characters `%!`, or the printer daemon will happily print out the text of the program instead of sending it to the printer directly.

4 Stacking the deck

One very important concept in PostScript is that it follows *Reverse-Polish Notation* (*RPN*), which may be familiar to you if you use a Hewlett-Packard calculator. It is the notation that tells you to add 2 and 3 by:

```
2 3 add
```

This notation is also called *postfix notation*, and is used in several other computer languages, notably FORTH. If you have not seen this before, you should learn it. The basic concept to keep in mind here is that of a stack.

Imagine a stack of plates. Plates can be added to the top, and must be removed only from the top of the stack. Think now of the stack as an abstract model, capable of holding many objects. Each time you enter an object, it gets "pushed" onto the top of the stack. Removing objects can only be done from the top of the stack, and is called (surprise!) "popping".

One useful thing to have when you are starting to play with the stack is something to print out the current contents of the stack. You can do this in Ghostscript by using the `pstack` command. Using the previous example, this is what happens when you use Ghostscript.

```
GS>2 3
GS<2>pstack
3
2
GS<2>add
GS<1>pstack
5
```

Notice that the prompt for Ghostscript changes when you have objects on the stack, and that the contents of the stack are listed from top to bottom. The output from `pstack` goes to the xterm, not to the preview window. There is no way to get `pstack` to work over a print spooler.

Operators will remove a certain number of operands in the stack and then act upon them. Some operators will also push some results on the stack. For example, the `add` operator would take the numbers 2 and 3, add them together, and push the result, 5, onto the stack.

Other simple arithmetic operators that are handy to know include `sub`, `mul`, and `div`. For example:

```
GS>10 5 div
GS>pstack
2
```

There are a large number of calculations and operations that you can do with the stack. For example, to discard the top value from the stack, you can use `pop`.

```
GS>2
GS<1>pstack
2
GS<1>pop
GS>pstack
GS>
```

There is another command that you can use to exchange the top two objects on the stack, the `exch` command.

```
GS>1 2
GS<2>pstack
2
1
GS<2>exch
GS<2>pstack
1
2
```

You can also duplicate the top object on the stack, using the `dup` command.

```
GS>5
GS<1>pstack
5
GS<1>dup
GS<2>pstack
5
5
```

On the more esoteric side, it is possible to `roll` things about on the stack using the syntax x y `roll`, where x is the number of items to roll and y is how much to roll it. Positive y rolls up on the stack, negative y rolls down.

```
GS>1 2 3 4 5
GS<5>5 1 roll
GS<5>pstack
4
3
2
1
5
```

5 Get moving—graphics basics

Now that you are familiar with some of the basic syntaax of PostScript, you are probably wondering how to get PostScript to draw graphics. There are a number of key concepts that you must grasp before you can effectively program using the graphics operators of PostScript.

5.1 The coordinate system

PostScript uses a cartesian coordinate system, with the default origin at the lower-left hand corner of the paper. The x axis points to the right, and the y axis points to the top, by default. There are 72 units in an inch, known as *points* in printers' lingo.[3] You do not have to use the default coordinate system if you do not want to. In fact, it is possible to use any rectangular coordinate system that you can derive from this one using linear two-dimensional transformations. The coordinate system that you normally use when programming in PostScript is called *user space*. At this point, in user space, no device-dependent characteristics apply yet. One way to visualize user space is as an imaginary page upon which you can draw a path with a "pen".

5.2 Path construction

What PostScript calls this "pen" is the *current point*. At first, the location of the current point is not defined. To begin a path, you must define the current point by using the `moveto` operator. You supply the x and y coordinates for the current point to be moved to in user space, like *x y* `moveto`. For example, `72 144 moveto` will move the current point to a location 1 inch to the right and 2 inches up from the lower-left corner of the page.

After establishing the location of the current point, you can go on to construct a *current path* in user space by using operators called *path constructors*. Almost all of the path constructors require that a current point be defined, or they will produce an error. You can think of the current path as the path traced out by this imaginary pen as it moves around in user space.

5.3 Painting

After you have a curent path in user space, you must render it onto the *current page*, which is a virtual image of what will end up on the physical page. The current page uses a coordinate system called *device space*, where each pixel is one unit on a side. This coordinate system is device dependent, and essentially, the current page is as memory image of every pixel that winds up on the physical page. You can draw onto the current page by using *painting operators*. Most of the more useful painting operators take paths from user space and render them onto the current page, although some of them will go directly to the current page without

[3]Actually, there are not exactly 72 points in an inch, but then again the definition of a point is not that standard. For more details, see Chapter 10 of *The TEXbook*

using anything from user space. These include `stroke` and `fill`, which respectively draw a line along a path and fill a path. These will be described in more detail later on. Most commands that operate on the current path to render it onto the current page will clear the path and reset the current point.

5.4 The physical page

Although you can draw images onto device space, they will not appear until you copy device space to a physical page. You use the `showpage` operator to do this. Not all PostScript previewers will wait until you execute `showpage` before they show what is being drawn. `dxpsview` will show you the physical page, but there is a menu option to get it to show you the current page as it is constructing it. Ghostscript will show you what the image in device space is, rather than what the physical page is. Ghostscript will display the following when it encounters a `showpage`.

```
GS>showpage
>>showpage, press <return> to continue<<
```

and then clear the page after you press `<return>`.

5.5 `moveto`

To actually start drawing, you have to set the current point. The `moveto` command will set the current point. The syntax for this command is *x y* `moveto`. This will move the current point to the coordinates (x, y). You may be wondering what coordinate system these are in. It so happens that the default coordinate system is cartesian, with the origin at the lower-left hand corner of the paper, the x axis along the short edge of the paper, and the y axis along the long edge of the paper. For example, this command:

```
72 144 moveto
```

will cause the current point to be set to a point one inch right and two inches up from the lower-left hand corner of the page.

5.6 `lineto`

Drawing line segments is almost as simple. To do so, you can use the `lineto` command, whose syntax looks like this: *x y* `lineto`. What this will do is draw a line segment from the current point to the point (x, y) and set the current point to (x, y).

You will encounter an error if you attempt to use a path contruction operator without first setting the current point. For example, Ghostscript will do something like the following if you execute a `lineto` without first doing a `moveto`.

```
GS>100 100 lineto
Error: /nocurrentpoint in --lineto--
Operand stack:
    100 100
Execution stack:
    operator_403588 --nostringval-- --nostringval-- operator_409f10
--nostringval-- --nostringval-- false --nostringval-- --nostringval--
--nostringval--
Dictionary stack:
    337/401 3/200
```

5.7 stroke

If you tried doing stuff with moveto and lineto in Ghostscript, you probably noticed that nothing got drawn. There is a good reason for this. The moveto and lineto commands do not actually do any drawing. Instead, they update something that PostScript keeps track of that is called the *current path*. To actually put something on the page you have to do something with the current path. One of the things you can do is to stroke the path. For example:

```
100 100 moveto 200 300 lineto stroke
```

will end up drawing a diagonal line near the lower-left part of the page. The stroke command also clears the current path, which is usually what you want it to do.

5.8 showpage

Of course, all that stroke does is update the *current page*, which PostScript keeps around until you do something with it. Usually what you want to do with the current page is print it out, so you can use the showpage command to actually cause the current page to be output from the printer, and also causes the current page to be erased, which is usually what you want to do.

One thing worth noting is that the showpage command under Ghostscript does not actually do anything. What Ghostscript does instead is pause and wait for you to press return before continuing, displaying the following message:

The reason for this is that Ghostscript does not show the page when you do a showpage, rather, it displays the current page as it is being drawn (each time you use stroke or fill, for exmple).

5.9 closepath

You may notice something when you try to draw a closed path. Even if you finish the path with the same point that you start it off with, the corner may look a bit strange.

If you look at Figure 1 carefully, you will notice that the lower-left hand corner of the square does not close completely. The reason for this is that the ends of a path do not extend beyond their endpoints. Lines do have a real thickness, and the thickness of the lines in Figure 1 have been increased to exaggerate the problem. To get around such a situation, you should use the `closepath` command to finish off your path, instead of using an explicit `lineto` to end it.

[h]

The `setlinewidth` command sets the line width to be used in the next `stroke` command. You should be able to see what happens if you try to close the path without using the `closepath` command. Compare this to Figure 2, which does use the `closepath` command. The two major advantages of using `closepath` are that you do not get messy joins on closed shapes, and you do not have to remember where the path started out.

6 Less filling! Tastes great! (`fill` and other fun)

You can do other things with paths than just `stroke` them. It is also possible to `fill` them. In this case, on a path where you might want to use `closepath` if you were using `stroke`, the use of `closepath` is normally not necessary. As a matter of fact, you can leave out the closing segment of a path that you are going to `fill`, since it gets put in anyway. The way you fill a path is with the `fill` command. For an example, see Figure 3

Another useful thing is the ability to control the level of gray which something gets filled with. You can do this using the `setgray` command. The syntax for this command is x `setgray`, where x is a number between 0 and 1 specifying the amount of gray. A value of 0 signifies black, and a value of 1 signifies white. For example, a square filled at 0.75 can be seen at Figure 4.

Of course, it is also possible to use the `setgray` command with something like the `stroke` command. Incidentally, one thing you should watch out for is that large areas of black are not good for the printer. At best, you end up wasting toner. At worst, you can actually damage the printer. In addition, large areas of black do not get printed very well, even if they do not happen to damage the printer.

Now that we have introduced most of the fundamental commands in PostScript, Figure 5 is a quick summary of what PostScript does with the path, the current page, and the physical page. Also, take a look a Figure 6 to see an example of how the PostScript graphics commands behave like they are using opaque ink.

7 Running around in circles (`arc` and `arcn`)

There are several commands in PostScript which are useful in drawing circles or arcs. These behave differently depending on whether or not the current point is defined. One of these is arc. The syntax for this one is *xcenter ycenter radius start end* arc. For example:

```
300 300 100 0 360 arc
```

will draw a complete circle centered on the point (300,300), and having a radius of 100 points. The other arc-drawing command is arcn. The only difference between the two is the direction in which they draw the arc. The arc command draws counter-clockwise, while the arcn command draws clockwise. If, however, there is a current point defined when you call arc or arcn, there will be a line segment connecting the previous current point and the starting point of the arc.

For an example of using arc with a current point defined, see Figure 7. For an example of using arcn without the current point defined, see Figure 8. Notice that the *start* and *end* parameters specify the starting and ending angles, specified counter-clockwise from the positive x axis.

8 A word is worth a thousand pictures—setting text

At this point, you may be wondering how to do things like get text on the page. There are a number of things that you must do before you can get started drawing text. One of them is to select a font, ie. typeface, in which to draw the text. The next thing is to select a size in which to draw the text. For example, you would use the following to select twelve-point Times-Roman:

```
/Times-Roman findfont 12 scalefont setfont
```

The findfont command actually goes and fetches the font from wherever it lives, and returns it. Most fonts start out scaled to one point: not a very useful size. The scalefont command takes a font and a point size, and proceeds to scale the font to that point size. Finally, the setfont command actually tells PostScript to use that font as its current font.

To actually put text on the page, you must first have a current point defined. The show operator takes a string and displays it. For example:

```
100 100 moveto (This is a test) show
```

Notice that strings are enclosed in parentheses. You can include balanced parentheses within strings, but if you want to include unbalanced parentheses you should precede them with a backslash, like this:

```
(some unbalanced \(parentheses\()
```

Figure 9 is an example of using the `show` command to display text.

Not all fonts are available everywhere. Typically a laser printer will have the fonts Times-Roman, Times-Italic, Times-Bold, Times-BoldItalic, Courier, Helvetica, Helvetica-Oblique, Helvetica-Bold, Helvetica-BoldOblique, and Symbol. There are a number of other fonts out there, but these are the most widely available.

9 Moving in with relatives (`rmoveto` and `rlineto`)

There are ways to construct path by specifying coordinates relative to the current point. One of the advantages of this is that you do not have to constantly remember where the current point is, or if you want to draw something that might get drawn in a variety of positions around the page. The commands to do this are `rmoveto` and `rlineto`. You use these commands much as you would use `moveto` and `lineto`. For an example of how to use `rlineto`, see Figure 10.

10 `def` awareness—defining stuff and why you should

PostScript has a command that lets you define things. This can prove to be very useful when you find yourself repeating command or numbers often. The `def` command is used to define a variable. It turns out that PostScript variables and procedures are practially indistinguishable. The way that you use the `def` command is *key value* `def`. For *key*, you should specify something begining with the / character. *value* can be just about anything. For example:

```
/pi 3.1415926 def
```

will define a variable that can be referenced by using `pi`.

On the other hand, you can also define procedures in PostScript by using a slightly different syntax. One thing that people use rather often is the following:

```
/inch {72 mul} def
```

This is rather convenient, because instead of writing something like `72 72 moveto`, you can use this: `1 inch 1 inch moveto`. Curly braces are special to PostScript. When the PostScript interpreter sees an open curly brace, it enters a *deferred execution mode*, where commands are not executed, but rather saved up to be assembled into one object later. When the interpreter reaches the close curly brace, it takes everything between the two curly braces and treats it as one *procedure object*. Procedure objects are used for things other than defining procedures, but we can get to that later.

You "pass" values to a procedure through the stack. Procedures really do not take parameters or arguments as you may be familiar with from other programming languages. If you are going to use a number of things many times within a procedure, you can use something like:

```
/foo exch def
```

to define the top thing on the stack to the key `foo`. At this point, if you understand how stack operators work, you should be able to understand why this works.

One thing to watch out for is that PostScript *does not* have any form of lexical scoping, as many languages do. There is no way to make local variables without making them on the stack in some way.[4]

11 Front and center—how to center text

PostScript has a command that will let you find out how much the current point will travel while displaying a string. This is the `stringwidth` operator. The syntax for it is *string* `stringwidth`. It does take *string* off of the stack, so you should have another copy somewhere, either defined to something, or another copy on the stack. It returns both the x and the y widths for the string. Usually, the y width does not mean much to you, so you can safely discard it. Unlike what you may think it is, it has nothing to do with how tall the string is.

For example, the following procedure will take a string, an x and y coordinate, and center the string around it. Note that it will not center the string about the y coordinate, but rather place its baseline on the y coordinate.

```
/center { moveto dup stringwidth pop neg 0 rmoveto show } def
```

You can invoke this by using *string x y* center, once you have defined it. For an example of this, see Figure 11.

12 Transformers—you control the coordinate system

You do not have to always use the coordinate system that PostScript starts you out with. In fact, it is possible to apply any[5] linear transformation to the coordinate system that you have at the moment and get a new one. One thing to keep in mind is that the only thing that changes when you alter the coordinate system is the coordinates; the current path, the current point, as well as anything that has already been set on the current page, remain unaltered by the transformation.

[4]Actually, this is not true. It is possible to define local variables by using dictionaries, but we won't talk about that here. Look in the Red Book if you want more details.

[5]This document will not go over all possible linear transformations. See the Red Book for more.

12.1 `scale`

One of the many things that you can do is to scale the coordinate system by some arbitrary amount in both x and y directions. You can also use a negative scaling factor to cause something to be drawn in a mirror image. The syntax for this is *x y* `scale`, where x is the amount to scale in the horizontal direction and y is the amount to scale in the vertical direction. For example, the following:

```
2 2 scale
```

scales the coordinate system uniformly in both directions.

12.2 `translate`

The `translate` command moves the origin of the coordinate system to a new location. The syntax is *x y* `translate`, where x and y are the horizontal and vertical displacements to move the origin. For example, the following:

```
4.25 inch 5.5 inch translate
```

will move the origin of the coordinate system to the center of the page.

12.3 `rotate`

You can also rotate the coordinate system about its origin. The syntax for this is *theta* `rotate`, where *theta* is the angle, in degrees, to rotate the coordinate system counter-clockwise around the origin. For example, the following will rotate the coordinate system 30° counter-clockwise:

```
30 rotate
```

For an example of rotating text, see Figure 12.

12.4 landscape mode

A quick way to get a landscape mode (where the coordinate system is oriented so that the long side of the paper is the x axis) is to combine `translate` and `rotate`, as follows:

```
90 rotate 0 -8.5 inch translate
```

13 Saving Massachusetts (or any graphics state)

PostScript has a concept of a *graphics state*, which consists of a number of parameters that stay mostly the same as long as you are on the same page. Most of the graphics state gets reset when you execute a `showpage`. The graphics state consists of things like the current path, the current transformation, gray value, line width, font, and several other things that we have not mentioned yet.[6] Anyway, it is possible to save and restore this graphics state. One common reason to do this is to reuse a path for more than one thing. The command to save a graphics state is `gsave`, and the command to restore a graphics state is `grestore`. These do not return any values on the operand stack, but rather push and pop the graphics states from a special stack called the *graphics state stack*.

For an example of how to use `gsave` and `grestore` to reuse a path, see Figure 13.

14 Where to Get More Information

If you want more information, you can look in the *PostScript Language Reference Manual* (second edition), by Adobe Systems. This is often called the "Red Book" by PostScript hackers. If you want more of a beginner's guide, you should look at the *PostScript Language Tutorial and Cookbook*, also by Adobe Systems. This is often called the "Blue Book" by PostScript hackers.

You can also get help by coming by the SIPB office (in room W20-557, just outside the student center Athena cluster), or by giving us a call at x3-7788. We suggest that you do this if you happen to have a complicated problem.

15 Acknowledgments

Kevin Iga wrote the original version of this document. That was later cleaned up by Dave Cho, and then later rewritten by Tom Yu. Many useful comments have been provided by Richard Barbalace, Mark Eichin, Sybil Shearin, and others. Thanks to Kevin Iga for the amusing jokes.

16 About this document

This document was prepered with LaTeX, using the font NewCenturySchlbk-Roman as a text font, and Courier as a monospaced display font for examples. Almost all of the figures in this document were prepared as PostScript inserts into the LaTeXdocument. The same PostScript code was used to set both the graphics output of the examples as well as to typeset the commands used to generate them.

[6] There is a complete list of things in the graphics state in the Red Book.

```
100 100 moveto
400 100 lineto
400 400 lineto
100 400 lineto
100 100 lineto
40 setlinewidth
stroke
showpage
```

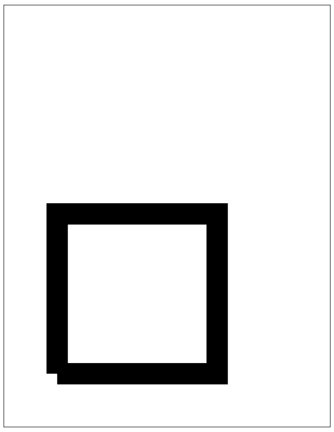

Figure 1: Drawing a box, not using `closepath`

```
100 100 moveto
400 100 lineto
400 400 lineto
100 400 lineto
closepath
40 setlinewidth
stroke
showpage
```

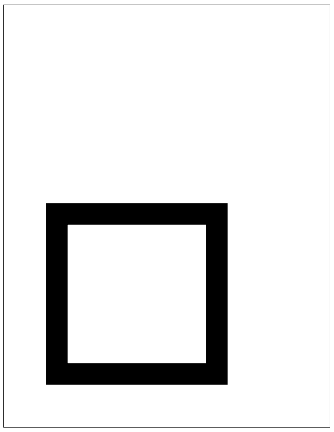

Figure 2: Another box, using `closepath`

```
100 100 moveto
300 100 lineto
300 300 lineto
100 300 lineto
fill
showpage
```

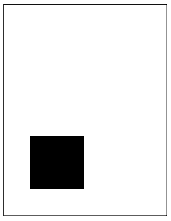

Figure 3: Example of using `fill`

```
100 100 moveto
300 100 lineto
300 300 lineto
100 300 lineto
0.75 setgray
fill
showpage
```

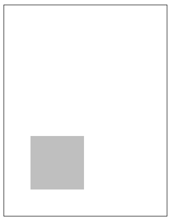

Figure 4: Example of using `setgray`

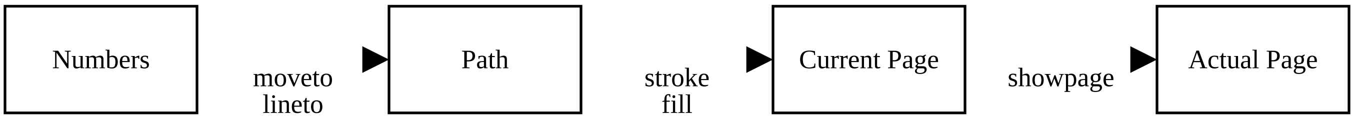

Figure 5: The purpose of `stroke` and `showpage`

```
0.75 setgray
100 100 moveto
300 100 lineto
300 300 lineto
100 300 lineto fill
1 setgray
200 100 moveto
300 200 lineto
200 300 lineto
100 200 lineto fill
showpage
```

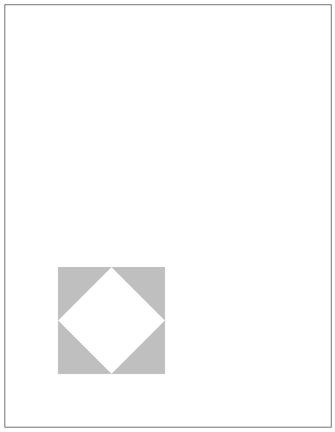

Figure 6: Example of the opaque nature of PostScript graphics operators

```
300 300 moveto
300 300 200 0 90 arc
stroke
showpage
```

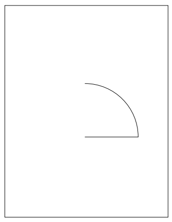

Figure 7: Using `arc` with the current point defined

```
300 300 200 0 90 arcn
stroke
showpage
```

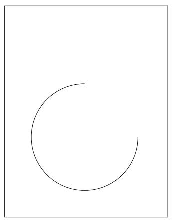

Figure 8: Using `arcn` without the current point defined

```
100 100 moveto
/Times-Roman findfont
144 scalefont setfont
(testing) show
showpage
```

Figure 9: Example of using `show` to display text

```
100 100 moveto
100 0 rlineto
0 100 rlineto
-100 0 rlineto
closepath
stroke
showpage
```

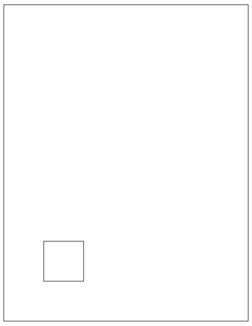

Figure 10: Using `rlineto`

```
/Times-Roman findfont
144 scalefont setfont
(centered) 4.25 inch 5.5 inch center
showpage
```

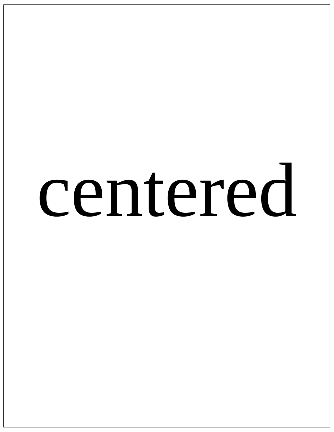

Figure 11: Example of centering text

```
100 100 moveto
30 rotate
/Times-Roman findfont
144 scalefont setfont
(rotation!) show
showpage
```

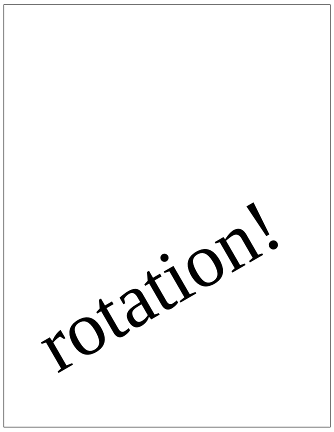

Figure 12: Example of rotating text

```
100 100 moveto
500 100 lineto
500 500 lineto
100 500 lineto
closepath gsave
0.75 setgray fill
grestore
10 setlinewidth stroke
showpage
```

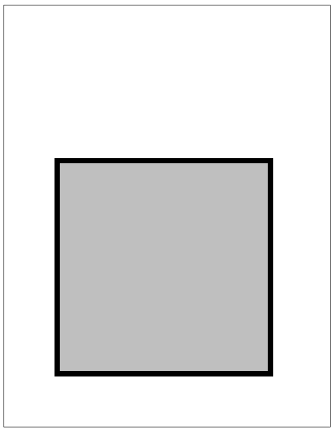

Figure 13: Using `gsave` to save the path