A

PostScript Tutorial

The figures in this book are created using the PostScript language. This is a mini-guide to PostScript with the purpose of giving you enough information so that you can alter these images or create similar ones yourself.

PostScript is a page description language.[1] A PostScript program tells the output device (printer or previewer) how to format a printed page. Most laser printers today understand the PostScript language. A previewer, such as Ghostview, allows you to view your document without printing—a great way to save paper. Ghostview can be found at

http://www.cs.wisc.edu/˜ghost/index.html.

In case you would like more in-depth information about Post-Script, see [13, 12].

PostScript deals with 2D objects only. A 3D package exists at this site:

http://www.math.ubc.ca/people/faculty/cass/graphics/text/www/.

Before you attempt to go there, be sure that you understand this tutorial!

A.1 A Warm-Up Example

Let's go through the PostScript file that generates Figure A.1.

[1]Its origins are in the late seventies, when it was developed by Evans & Sutherland, by Xerox, and finally by Adobe Systems, who now own it.

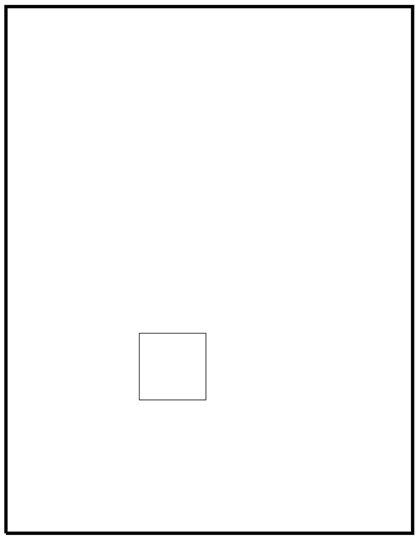

Figure A.1.
PostScript example: drawing the outline of a box.

```
%!
newpath
    200 200 moveto
    300 200 lineto
    300 300 lineto
    200 300 lineto
    200 200 lineto
stroke

showpage
```

Figure A.1 shows the result of this program: we have drawn a box on a standard size page. (In this figure, the outline of the page is shown for clarity; everything is reduced to fit here.)

The first line of the file is `%!`. For now, let's say that nothing else belongs on this line, and there are no extra spaces on the line. This command tells the printer that what follows is in the PostScript language.

The actual drawing begins with the `newpath` command. Move to the starting position with `moveto`. Record a path with `lineto`. Finally, indicate that the path should be drawn with `stroke`. These commands simulate the movement of a "virtual pen" in a fairly obvious way.

PostScript uses *prefix notation* for its commands. So if you want to move your "pen" to position $(100, 200)$, the command is `100 200 moveto`.

Finally, you need to invoke the `showpage` command to cause PostScript to actually print your figure. This is important!

Now for some variation:

```
%!
% These are comment lines because they
% begin with a percent sign

% draw the box
newpath
    200 200 moveto
    300 200 lineto
    300 300 lineto
    200 300 lineto
    200 200 lineto
stroke

80 80 translate
0.5 setgray

newpath
    200 200 moveto
    300 200 lineto
    300 300 lineto
    200 300 lineto
    200 200 lineto
fill

showpage
```

This is illustrated in Figure A.2. The `80 80 translate` command moves the origin of your current coordinate system by 80 units in both the $\mathbf{e}_1$- and $\mathbf{e}_2$-directions.

The `0.5 setgray` command causes the next item to be drawn in a gray shade (0 is black, 1 is white).

What follows is the same square as before; instead of `stroke`, however, there's `fill`. This fills the square with the specified gray scale. Note how the second square is drawn on top of the first one!

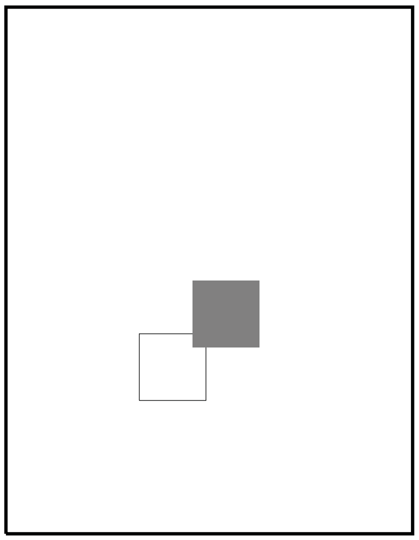

Figure A.2.
PostScript example: drawing two boxes. The outlined box is drawn first and the filled one second.

Another handy basic geometry element is a circle. To create a circle, use the following commands:

```
250 250 50 0 360 arc stroke
```

This generates a circle centered at $(250, 250)$ with radius 50. The `0 360 arc` indicates we want the whole circle. To create a white filled circle change the command to

```
1.0 setgray
250 250 50 0 360 arc fill
```

Figure A.3 illustrates all these additions.

A.2 Overview

The PostScript language has several basic graphics operators: shapes such as line or arc, painting, text, bit mapped images, and coordinate transformations. Variables and calculations are also in its repertoire; it is a powerful tool. Nearly all the figures for this book are available as PostScript (text) files at the book's website. To illustrate the aspects of the file format, we'll return to some of these figures.

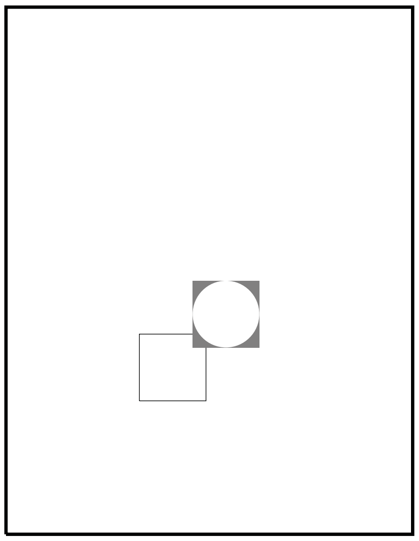

Figure A.3.
PostScript example: building on Figure A.2, a white-filled circle is added.

We use three basic scenarios for generating PostScript files.

1. A program (such as C) generates geometry, then opens a file and writes PostScript commands to plot the geometry. This type of file is typically filled simply with move, draw, and style (gray scale or line width) commands. Example file: `Bez_ex.ps` which is displayed in Section 18.2.

2. The PostScript language is used to generate geometry. Using a text editor, the PostScript program is typed in. This program might even use control structures such as "for loops." Example file: `D_trans.ps` which is displayed in Section 6.3.

3. A print screen creating a bit mapped image is made part of a PostScript file. Example file: `Flight.ps` which is the introductory figure of Chapter 12.

There are also figures which draw from more than one of these scenarios.

Since PostScript tells the printer how to format a page, it is necessary for the move and draw commands to indicate locations on the piece of paper. For historical reasons, printers use a coordinate system based on *points*, abreviated as pt.

$$1 \text{ inch} \equiv 72 \text{ pt.}$$

PostScript follows suit. This means that an $8\frac{1}{2} \times 11$-inch page has the following extents:

$$\text{lower left}: \ (0, 0) \qquad \text{upper right}: \ (612, 792).$$

Whichever scenario from above is followed, it is always necessary to be sure that the PostScript commands are drawing in the appropriate area. Keep in mind that you probably want a margin. Chapter 1 covers the basics of setting up the dimensions of the "target box" (on the paper) and those of the "source box" to avoid unwanted distortions. Getting the geometry from the source box to the target box is simply an application of affine maps!

A.3 Affine Maps

In our simple examples above, we created the geometry with respect to the paper coordinates. Sometimes this is inconvenient, so let's discuss the options.

If you create your geometry in, say, a C program, then it is an easy task to apply the appropriate affine map to put it in paper coordinates. However, if this is not the case, the PostScript language has the affine map tools built-in.[2]

There is a matrix (which describes an affine map, i.e., a linear map and a translation) in PostScript which assumes the task of taking "user coordinates" to "device coordinates." In accordance to the terminology of this book, this is our source box to target box transformation. This matrix is called the *current transformation matrix*, or *CTM*.

In other words, the coordinates in your PostScript file are always multiplied by the CTM. If you choose not to change the CTM, then your coordinates must live within the page coordinates, or "device coordinates." Here we give a few details on how to alter the CTM.

There are two "levels" of changing the CTM. The simplest and most basic ones use the `scale`, `rotate`, and `translate` commands.

[2]There are many techniques for displaying 3D geometry in 2D; some "realistic" methods are not in the realms of affine geometry. A graphics text should be consulted to determine what is best.

As we know from Chapter 6, these are essentially the most basic affine operations.

Unless you have a complete understanding of the CTM, it is probably a good idea to use each of these commands only once in a PostScript file. It can get confusing! (See Section A.6.)

A scale command such as

```
72 72 scale
```

automatically changes one "unit" from being a *point* to being an inch. A translate command such as

```
2 2 translate
```

will translate the origin to coordinates $(2, 2)$. If this `translate` was preceded by the `scale` command, the effect is different. Try both options for yourself!

A rotate command such as

```
45 rotate
```

will cause a rotation of the coordinate system by 45 degrees in a counterclockwise direction.

These commands are used in the website files `D_scale.ps`, `D_trans.ps`, and `D_rot.ps`.

Instead of the commands above, a more general manipulation of the CTM is available (see Section A.6).

A.4 Variables

The figures in this book use variables quite often. This is a powerful tool allowing a piece of geometry to be defined once, and then affine maps can be applied to it to change its appearance on the page.

Let's use an example to illustrate. Take the file for Figure A.2. We can rewrite this as

```
%!
% define the box
/box {
        200 200 moveto
        300 200 lineto
        300 300 lineto
        200 300 lineto
        200 200 lineto
} def

newpath
box
stroke

80 80 translate
0.5 setgray

newpath
box
fill

showpage
```

The figure does not change. The box that was repeated is defined only once now. Notice the `/box {...} def` structure. This defines the variable `box`. It is then used without the forward slash.

A.5 Loops

The ability to create "for loops" is another very powerful tool. If you are not familiar with the prefix notation this might look odd. Let's look at the file `D_trans.ps`, which is displayed in Figure 6.3.

```
%!PS-Adobe-2.0
%%BoundingBox: 90 80 350 270
/Times-Bold findfont
70 scalefont setfont
/printD {
  0 0 moveto
  (D) show
} def

100 100 translate

2.5 2.5  scale
1 -.05 0.35 {setgray printD 3 1 translate } for

showpage
```

Before getting to the loop, let's look at a few other new things in the file. The `BoundingBox` command defines a minmax box that is intended to contain the geometry. This command is not used by PostScript, although it is necessary for correct placement of the figure in a LaTeX file. The `PS-Adobe-2.0` string on the first line will allow you to see the "BoundingBox" in Ghostview. The closer the bounding box is to the geometry, the less white space there will be around the figure. The `/Times-Bold findfont` command allows us to access a particular set of fonts—we want to draw the letter D.

Now for the "for loop." The command

```
1 -.05 0.35 { . . . } for
```

tells PostScript to start with the value 1.0 and decrement by 0.05 until it reaches 0.35. At each step it will execute the commands within the parenthesis. This allows the D to be printed 13 times in different gray scales and translated each time. The gray scale is the same as the "for loop" parameter.

A.6 CTM

PostScript always has a CTM which is currently active. If we want to apply a specific matrix to our geometry, then we need to concatenate that matrix with the CTM. The following PostScript code (which generates Figure A.4) gives an example.

```
%!PS-Adobe-2.0
%%BoundingBox: 40 80 400 160
% Demonstrates a shear in the e1-direction
% by modifying the CTM

/shear[1 0  1 1  0 0] def %the shear matrix

/pla {/Helvetica-Bold findfont
    70 scalefont setfont (PLA) show
    } def                 %command to print "PLA"

    70 100 translate
    0 0 moveto
    pla                   %print the left PLA

    150 0 translate       %go to new position to the right
    0 0 moveto
```

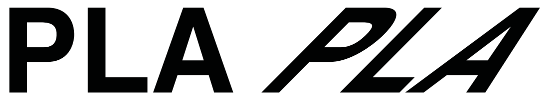

Figure A.4.

PostScript example: draw “PLA" and then concatenate the CTM with a shear matrix and redraw “PLA."

```
    shear concat          %apply shear matrix
    pla                   %print PLA again, now sheared

showpage
```

This piece of code shows you how to use a matrix in PostScript. In this book, we write transformations in the form $\mathbf{p}' = A\mathbf{p}$, which is called “right multiply.” PostScript, however, uses a transformation matrix for “left multiply.” Specifically, this book applies a transformation to a point $\mathbf{p}$ as

$$\mathbf{p}' = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \end{bmatrix},$$

whereas PostScript would apply the transformation as

$$\mathbf{p}' = \begin{bmatrix} p_1 & p_2 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

So the PostScript CTM is the transpose of the matrix as used in this book.

A matrix definition in PostScript has the translation vector included. The matrix has the form

$$[a, b, c, d, t_x, t_y].$$

The matrix *shear* in our example is defined as [1 0 1 1 0 0] and corresponds to “our” matrix form

$$\mathbf{p}' = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$