8-1-1988

A Study of PostScript as a graphics programming language

Ruane Miller

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
A Thesis Submitted to the Faculty of
The College of Fine and Applied Arts
in Candidacy for the Degree of
MASTER OF FINE ARTS

A Study of PostScript® As A
Graphics Programming Language

By
Ruane Miller

July 1988
I, Ruane Miller, prefer to be contacted each time a request for reproduction is made. I can be reached at the following address.

21 Adams Drive
Princeton, NJ 08540
ACKNOWLEDGEMENTS

I would like to acknowledge several of the many people who helped me along this path of discovery and exploration.

Professor James VerHague, RIT College of Fine and Applied Arts, for my appreciation and love of programming as a creative endeavor.

Professor Norman Williams, RIT College of Fine and Applied Arts, for the initial suggestion, inspiration, and help in getting started.

Professor Michael Kleper, NTID and RIT College of Graphic Arts and Photography, for his knowledgeable practical and technical support.

Mike Looney, of Apple, Inc., and the support people at Apple in Rochester for their patience and generous offer of space, equipment, and technical support.

Don Beil from NTID, Larry McKnight and Harvey Carapella from RIT Instructional Media Services, and C.R. Myers from RIT Information Systems and Computing for their sharing of knowledge, time, equipment, and space.
My involvement with PostScript began while I was searching for a thesis topic during the summer of 1985. I wanted a topic that would combine the two areas I am involved in: computer programming and art. Programming was a recent involvement; it began in Jim VerHague’s “2D Programming in Basic” course. Through his inspired instruction I developed an interest in, and addiction to, that discipline. I proceeded to take as many programming courses as my program would allow. What I enjoyed about programming was the search for solutions and patterns, for things which repeat and can be generalized, for an underlying structure or underlying pattern upon which variations can be made. This aspect of programming is very similar to processes I use in making art.

I was introduced to the recently developed language “PostScript” in a casual conversation with Norm Williams. There was very little written about the language at that point, but with his help I found a few articles and some names of people at Adobe, the company which developed PostScript. The language’s structure was direct and elegant. I was intrigued. It was a fully developed programming language designed for graphics. Line art, images, and typography were treated the same—as mathematical descriptions, not as bitmaps. This meant that one could transform a character (typefaces) as easily as and creatively as one could a two-dimensional graphics shape. The potential of using the language as an artist’s tool was exciting. I proceeded to explore this potential.


My thesis is an investigation of some basic PostScript programming operations and their artistic applications.

I began work on this thesis in September 1985 with the programming research initiated in March 1986. I exhibited the graphics output from this research in May 1986 in the MFA exhibition in Bevier Gallery, College of Fine and Applied Arts. This report was written two years later. Within that time span many changes and advances in the application and device use of PostScript have taken place. It has become an industry standard and

---

its uses have been implemented in displays and in color printers. There are PostScript "clones" being developed—the language is being imitated. There is much software available that enables the designer to make use of the extensive graphics capabilities of PostScript, without needing to learn the language. However, for a designer involved with computer graphics, some basic knowledge of the PostScript language and its uses can only enhance his/her creative environment.

Within this written thesis report I will describe what I learned of PostScript, and how I used it in creating my visual thesis work. I will also delineate some overall guidelines for using PostScript to program graphics output.

This report was created in Microsoft® Word. The pages were composed in Aldus PageMaker®. The illustrations were PostScript program segments placed in the Pagemaker document as encapsulated PostScript files. It was printed on an AST TurboLaser®/PS. The typefaces are from Adobe Systems, Inc. The body type is Times-Roman with Italic, Bold Italic, and Bold. Chapter headings, sub-headings, and program listings are Helvetica with Bold and Bold Italic.

Ruane Miller
July, 1988
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ................................................................. iii
PREFACE ................................................................................ iv
LIST OF FIGURES ........................................................................ viii
LIST OF PROGRAM LISTINGS ................................................... ix

Chapter 1
THE POSTSCRIPT LANGUAGE: AN OVERVIEW .......... 1

Chapter 2
POSTSCRIPT'S STRUCTURE AS A PROGRAMMING LANGUAGE ........................................... 4
  2.1 Interpreted ................................................................. 4
  2.2 Stack Oriented .......................................................... 4
  2.3 Post-Fix Notation ....................................................... 5
  2.4 Data Structures .......................................................... 5
  2.5 Built-in Basic Operators .............................................. 5
  2.6 Structure and Style ..................................................... 6

Chapter 3
POSTSCRIPT'S STRUCTURE AS AN IMAGING LANGUAGE ................................................. 7
  3.1 Coordinate Systems ..................................................... 7
  3.2 Graphics State .............................................................. 7
  3.3 Image Construction ..................................................... 7
  3.4 Color, Halftones, and Scanned Images ......................... 8
  3.5 Text Operations ........................................................... 8

Chapter 4
POSTSCRIPT PROGRAMMING: OBSERVATIONS ........ 10

Chapter 5
POSTSCRIPT PROGRAMMING GUIDELINES FOR THE ARTIST AND DESIGNER .............. 38
  5.1 Coordinate Systems ..................................................... 38
  5.2 Stacks and Post-Fix Notation ....................................... 38
  5.3 Operators ................................................................. 39
  5.4 Style ........................................................................ 40
  5.5 Hardware and Software Configuration ......................... 40
  5.6 Programming ............................................................. 41
Appendix A
PRINTED POSTSCRIPT OUTPUT FROM PROGRAM VARIATIONS

Appendix B
PRINTED POSTSCRIPT OUTPUT FROM APPLE COMPUTER, INC. LASER PRINTERS
B.1 Apple LaserWriter PostScript Output
B.2 Apple LaserWriter IINTX PostScript Output

Appendix C
MFA THESIS MURAL EXHIBIT

BIBLIOGRAPHY
LIST OF FIGURES

Figure 4.1 Radiating Spheres ............................................ 13
Figure 4.2 Rotated Text on White ....................................... 20
Figure 4.3 Rotated Text on Black ....................................... 21
Figure 4.4 Random Spheres ............................................. 25
Figure 4.5 Random Spheres with Circular Clipping Path ........ 29
Figure 4.6 3-D RIT Logo .................................................. 32
Figure 4.7 Alphabet Blocks ............................................. 37
Figure 5.1 Stack Operations ............................................. 39
LIST OF PROGRAM LISTINGS

Listing 4.1  Radiating Spheres ........................................... 12
Listing 4.2  Rotated Text on White ................................. 18
Listing 4.3  Rotated Text on Black, script ...................... 19
Listing 4.4  Random Spheres, prologue ............................ 23
Listing 4.5  Random Spheres, script ............................... 24
Listing 4.6  Geometric Clipping Paths ............................. 27
Listing 4.7  Diagonal Clip, script ................................. 28
Listing 4.8  Diamond-Shaped Clip, script ....................... 28
Listing 4.9  Triangular Clip, script ............................... 28
Listing 4.10 Circular Clip, script ................................. 28
Listing 4.11 3-D RIT Logo ............................................. 30
Listing 4.12 Alphabet Blocks .......................................... 33
PostScript was created by Charles Geschke and John Warnock and developed by the company they formed, Adobe Systems, Incorporated, in 1982. This language grew out of years of research in interpretive graphics description and printing protocol languages. The original primary implementation of the language was as an interpretive page description language.

PostScript is a graphics oriented language. It is high-level and device independent. It is intended primarily for machine generation, that is, PostScript page descriptions are generally produced by software applications rather than by direct programming. Adobe set out to have PostScript become the industry printing protocol standard. It has since become that due to both the logical and well-designed functionality of the language, and its answering the printing industry's need for a high quality, portable and integrated typographic and design solution for a digital page composition, proofing, and printing system.\(^1\) PostScript has been the impetus behind what is a very recent yet forceful segment of the personal computer industry—desktop publishing.\(^2\) Through the foresight, talent and entrepreneurship of Geschke and Warnock, our ability to produce high quality typographic and graphic digital raster output is much improved.

A PostScript program communicates a printable page (document) description from the host system to the raster printer or imagesetter over regular communication channels. The program is then executed, line by line, by the PostScript interpreter which resides in the printer controller. Here, what came in as a composed model of a page is translated into the special requirements of the specific printer, such as printer resolution,

---

\(^1\)PostScript has been adopted by Apple Computer Inc., AST Research Inc., Digital Equipment Corp., IBM, Linotype Company, NEC Information Systems, QMS, Inc., Texas Instruments, Varityper, Wang, and many other companies for personal, mini, and mainframe systems, laser printers and imagesetters. The preponderance of software developers for these systems and printers have adopted PostScript as an imaging language for use in a very wide range of application programs.

\(^2\)Desktop Publishing was made prominent by Apple Computer, Inc. when it introduced the Macintosh® and LaserWriter® (the LaserWriter was the first commercial desktop printer to implement the PostScript language) in conjunction with Adobe, Inc.'s PostScript®, Aldus Corporation's PageMaker®, the original page-layout program, and Linotype Company's Linotronic™ PostScript® Imagesetters.
page size, etc. The printer could be a laser, dot matrix, or inkjet printer or a digital imagesetter. Although PostScript was originally implemented in raster printers, the language is designed as a raster output oriented language. It’s output, then, could be to a raster display as easily as to a raster printer.  

A raster device is composed of an array or two dimensional matrix of units called pixels (picture elements). A black and white raster device would have one of only two possible values in each pixel’s corresponding memory cell. This is represented as a single bit in memory, 1 or 0, on or off, thus, black or white. A page then, on a raster device, is described in memory as a pattern of 1’s and 0’s; whether that pattern describes text, graphics, or both is unimportant. An application program could describe a page’s contents as a bitmap (a coded pattern of pixels). The description would have to be device specific, matched to its resolution, its number of pixels. This type of description is limited, costly in memory, and somewhat inefficient. PostScript does not describe the page as a bitmap; it, instead, describes the page components as mathematical, abstract entities. These descriptions can be used on any raster device, regardless of resolution. They need only to be interpreted for that specific device. PostScript describes a model of the page—how the page will look. It does not describe the dots which represent the page.

PostScript is a dynamic page description language. It makes use of procedures, variables, control constructs, and the modularity of high-level programming languages. This allows flexibility. The language can meet the needs of various application programs. It is not restricted by a static set of controls.

PostScript is graphically oriented. A PostScript program describes a page of graphic objects rather than a page of pixels. It uses powerful graphics primitives (operators) which correspond in their usage and terminology to traditional graphics terms and functions. The attributes of typefaces (their specific design features), not the actual characters, are described mathematically as outlines, not as bitmaps. This requires the storage of information on only one version of a typeface. This typeface version can then be adjusted by geometrically transforming the outline to produce characters in different sizes or rotations at any resolution with the same high visual quality as the original version. A bitmap must describe each size of each font and face; this is memory intensive and costly.

---

3Adobe Systems, Inc. is currently marketing the concept as the Display PostScript system and has recently entered into contracts with NEXT, Inc. (Steve Jobs’ newly formed company) to use PostScript as a display generator.
Text, line art, and graphic images are treated alike by PostScript. This allows for an easy integration of text and graphics on a page. One can manipulate text, line art, or scanned input with the PostScript operators; they can be rotated, scaled, skewed, reversed, shaded, filled, etc. The page itself can be treated as an object and the same PostScript operators applied to it that are applied to individual page elements. All of this allows the user greater flexibility along with high quality output. For the designer, PostScript opens a world of creative possibilities.
Chapter 2  POSTSCRIPT'S STRUCTURE AS A PROGRAMMING LANGUAGE

2.1 INTERPRETED

An interpreted language program is executed line by line as in BASIC, rather than as a whole as are compiled language programs such as in Pascal, C, and most other high level programming languages. The interpreter processes each element sequentially. PostScript is an interpreted language. This offers flexibility to the programmer in its immediate feedback of errors and executions. It makes a PostScript program easier to debug. It allows an interaction with the program and its output. The PostScript interpreter scans characters, and based upon the syntactical rules, will suspend scanning and execute the object. It then resumes scanning. The standard character set for PostScript programs is the printable characters of the ASCII character set, a standard in communication transfer.

2.2 STACK ORIENTED

PostScript is based on stacks which are fixed data structures, defined areas for storage of data that will be used by the interpreter. Objects (values) are stored on the stacks temporarily, on top of one another. The last object put on the stack is the first object taken out (known as LIFO, last in, first out). There are four different and independent stacks used by the PostScript interpreter: execution stack, operand stack, dictionary stack, and graphics state stack. The execution stack contains the object which is currently being executed along with the standard input file object, and an infinite loop known as the server loop which isolates batch jobs being sent to the interpreter. The operand stack is used, with a few exceptions, for data objects which are not directly executable; the executing program operators get their data (the operands) from here and return their operation results. The dictionary stack holds only dictionary objects: strings, variables, and procedures. All PostScript operations are defined and libraries of procedures saved here. The graphics state stack holds the current graphics control parameters such as color, position, font, line width. The graphics operators obtain this information from the graphics state stack as the current graphics state.
2.3 POST-FIX NOTATION

In PostScript the operators (the commands) are preceded by their operands (what the commands operate on). Thus, when the program is executed, the operand is reached first and put on the operand stack by the interpreter. The operator is then executed. For example, in the notation \(1 \ 2 \ ADD\), \(1\) and \(2\) are the operands and \(ADD\) is the operator. This is called post-fix notation. It is quite different from our usual notation, infix, where the operator is between the operands. In \(1 \ + \ 2\), \(1\) and \(2\) are the operands and \(+\) is the operator. An understanding of these concepts of stack structure and post-fix notation are crucial to working with PostScript. Our usual sense of sequencing is turned backwards, so to speak, and one must develop this way of thinking when programming in PostScript.

2.4 DATA STRUCTURES

Integers, reals, strings, arrays, identifiers, and files are standard data structures provided for in the PostScript language. These are all considered PostScript objects and are manipulated on the operand stack. PostScript objects can be one of two kinds: simple or composite. A simple object has its value integral to it; it is self-contained. Integers, reals, and names (identifiers) are simple objects. A composite object is comprised of two parts: a value and, because these are stored in different areas, a pointer to that value. Strings, dictionaries, and arrays are examples of composite objects. Procedures exist in PostScript as executable arrays. Curly braces, {}, mark the beginning and end of a procedural body. When the interpreter encounters a procedure it will place everything within that procedure on the operand stack as one large object, an executable array. The procedure can be named (defined); that name will be stored on the dictionary stack to be referenced and used as any of the built-in PostScript operators are used. PostScript is an extensible language; one can define procedures and then add them to other procedures as part of those procedures.

2.5 BUILT-IN BASIC OPERATORS

PostScript has an extensive set of built-in operators which perform standard programming tasks and graphics and font operations. They can be broadly grouped into the following categories: stack and mathemati-
cal operators, program control operators, device and status operators, graphics operators, font operators. All of the individual operators are executable name objects stored in dictionaries. When this named object, an operator, is executed, its defined commands are executed.

2.6 STRUCTURE AND STYLE

PostScript does not require a specific structure. Because it is an interpreted language, one could easily avoid using loops and the creation of procedures and simply execute commands one by one. This could lead to redundancy and inefficiency in programs. It is advisable that the programmer impose and adhere to a structure.

Adobe Systems does have a set of structuring conventions for PostScript programs. A PostScript program can be conforming or non-conforming to these structuring standards. These standards have no effect on the execution of the program; they are comments that define and elaborate the program. These structuring comments must begin with the two characters %% followed by the key word, such as %%EndPrologue. An explanatory comment is delimited in PostScript by the character %; all characters between it and the next newline are comments and are not executable. Adobe recommends that all programs begin with a comment line that starts with %! as in %!PS-Adobe-2.0.

The organizational model recommended for a PostScript program is to divide it into two parts, the prologue and the script. The prologue consists of only definitions, whether procedures, variables, or constants, which will be used by the rest of the program. It does not contain any executable code. The script contains only executable code and operand data. A header can be at the top of the program, above the prologue. Here one would put specific program information such as title, programmer’s name, date, etc. A trailer, written as %%Trailer, can be placed at the end of the program; this ends the script portion of the program and contains program information not in the header.
Chapter 3  POSTSCRIPT’S STRUCTURE AS AN IMAGING LANGUAGE

3.1 COORDINATE SYSTEMS

PostScript makes use of two coordinate systems: the user coordinate system (user space) and the device coordinate system (device space). The user space is independent of any physical device. The user coordinate system is then mapped to the device coordinate system of the particular output device by the PostScript interpreter. PostScript’s default user space has its origin in the lower left hand corner with a unit of measure of 1/72 of an inch. This default unit corresponds closely to a standard printing unit, the point (1/72.27 inch). User space is fully variable. Thus, it may be changed in position (translated), orientation (rotated), or size (scaled). Lines and shapes are defined in terms of points on the current page as x and y coordinates within the defined user coordinate system. There are a full set of coordinate system operators which operate on the PostScript transformation matrix (Current Transformation Matrix, CTM) to manipulate user coordinates into device coordinates (translate, rotate, scale). This matrix maps the user space to device space.

3.2 GRAPHICS STATE

The PostScript data structure, the graphics state stack, holds the current graphics control parameters. This data describes how PostScript operators will affect the current page—what line width, color, font, user coordinate system, etc. they will use. There are graphics state operators (gsave, grestore) which allow one to save the current graphics state, modify it, then restore it. In essence, one can contain certain page elements within their own graphics state while retaining the original graphics state for the rest of the page or document.

3.3 IMAGE CONSTRUCTION

PostScript creates graphics (whether text or line art) by means of path construction and painting. This path can be comprised of lines, arcs, or Bezier curves, and uses the defined or default user coordinate system as
a reference grid. The paths are then “painted” with a current color or gray scale (black is the default). The painting is opaque—it covers any object beneath it. Paths may be used for clipping or masking graphics as well. They may be drawn (stroked) with a line, filled as areas, or both. There are path construction operators (moveto, lineto, newpath, closepath, arc) as well as painting operators (fill, stroke, image, show). When the page is composed, the showpage operator renders it on the output device and then clears the current page to white.

3.4 COLOR, HALFTONES, AND SCANNED IMAGES

The PostScript language has the ability to use three different full color models to specify color: CYMK (cyan, magenta, yellow, and black); HSB (hue, saturation, brightness); RGB (red, green, blue). Color models and colors are specified by color operators (setcmykcolor, sethsbcolor, setrgbcolor). Black, white, and intermediate shades of gray use the setgray operator. One can program color separations or spot color for a document which will employ full process color presses or spot color printing.\(^1\)

Digital halftoning is manipulated in PostScript through a specialized operator (setscreen). This operator allows for a simulation of traditional methods to change the size of a halftone screen through frequency (the number of halftone cells per inch), angle (the angle of cell rotation), and spot function (the shape of the halftone spot) parameters.

Sampled (scanned) images can be printed by PostScript in up to 256 gray levels with its image operator. The images can be rendered at different sizes and/or orientations with the use of the PostScript transformation operators.

3.5 TEXT OPERATIONS

A full set of specialized operators which handle text and typography exist in the PostScript language. These can be combined with other PostScript operators to increase type manipulation and control. A character is defined in PostScript as any other graphical object—a collection of lines and/or curves. A font, in PostScript, is a dictionary object which holds the

---

\(^1\)The majority of PostScript output devices are black and white and gray-scale. Color PostScript printers are in development; QMS, Inc. has just released the first color PostScript printer, the ColorScript\(^{TM}\) 100.
mathematical definition of the specific character within an executable procedure. A PostScript font is an executable program, one which, when executed by the interpreter, draws the typographic outline using the path construction operators. Each PostScript implementation contains a number of built-in fonts. Additional fonts can be downloaded from software or even defined by a program. They could be created from graphics and the program that defines those graphics shapes can be saved as a font dictionary and placed on the page using text operators.
Chapter 4

POSTSCRIPT PROGRAMMING: OBSERVATIONS

The process of learning PostScript programming was one of trial and error. I began the process by reading what was written on the language in order to develop a basis of understanding. Then, when I was able to access a PostScript printer, I began to experiment directly with the language. As mentioned earlier, the language resides in the printer’s controller; thus, in order to execute a PostScript program one needs a PostScript printer. There are two ways a program can be sent to the printer: as a batch file or interactively. A batch file is a PostScript program written in a text editor and saved as a text (ASCII) file. Even though the PostScript interpreter executes this batch file line by line, it is sent to the printer as a file, in a batch. In interactive mode the PostScript code (program) is entered on the host computer one executable line at a time. This line of code is sent to the printer and processed by the interpreter. A communications channel must be set up between the computer and the PostScript printer for both methods. The manner in which this is accomplished is dependent upon the specific hardware and software configuration.

My first programs were examples of code I found in my readings. I used both methods described above to execute the code and also combined them to make interactive program testing more efficient. I would create a file which contained the prologue (all the required and optional procedures and definitions) and not the script (the executable code). This file was downloaded through the communications software to the printer. The procedures and definitions contained within the file would be scanned into memory. I could then execute them interactively by typing and entering the script commands. This allowed me to change the arguments sent with the procedural calls and, thus, vary the size, color, or placement, in essence, the visual effect of the graphic object(s). In this way I could test variations on the main graphics concept. Listing 4.1 is an example of a prologue which contains the procedures and definitions to draw tonally shaded spheres of various sizes and placement and a script which allows one to change the placement and scale of the rays of spheres.  

1This program uses a procedure with a for loop and the rotate

1The examples of program listings included in this thesis conform to the following format: the file name and remarks, a brief definition of each subsequent procedure with a list of its arguments ordered to correspond to their placement on the stack: left to right: bottom to top, and brief comments on variables and operators. LLx and LLy stand for lower left coordinates, ULx and ULy for upper left coordinates, with respective codes
operator to draw rays of spheres rotated about a center point. Figure 4.1 is the output from this basic program. Appendix A contains other output examples of variations on this basic shaded sphere drawing program. As seen in these examples, the size, placement, and complexity of the rays of spheres can be easily changed by the arguments sent with the procedural calls from the script.

for the right coordinates. Where appropriate, each listing is followed by an example of that listing's output; these examples are scaled to fit within the text. Full size output can be found in Appendix A.
Radiating Spheres

(rays of spheres in graduated radii)

%PSAdobe-2.0
%Title: sph-rays.ps
%EndComments

/csp{%def

gsave
rotate
6 2 30{
/r exch def
/y 0 def
/x r 16 mul def
0 1 9{
9 div setgray
/x x .3 add def
/y y .3 add def
/r 16 ge {/r r 1.6 sub def}if
/r 12 ge {/r r 1.2 sub def}if
{/r r .8 sub def}ifelse
x y r 0 360 arc fill
}for
}for
grestore
}bind def

/tsc{%def

translate
scale
}bind def

%EndPrologue

1 1 306 396 tsc
0 15 345 csp
showpage

%Trailer
figure 4.1 Radiating Spheres

(output of listing 4.1, Radiating Spheres)
Because of the directness of communications between written code and printed output, it was easy to vary these initial attempts and begin to build on them. This direct interaction between my graphics concepts and the printed page answered a need for printed output and control that I had from my traditional art background and had found difficult to answer within the medium of computer graphics. I was not drawing with a pen on paper; however, I was controlling rather directly the printer’s “pen” on paper. This was equally creative.

I found PostScript to be easily understandable in its basic concepts. PostScript’s terminology and structural ideas are based on traditional graphics concepts. Thus, constructing (drawing) a shape with PostScript is comprehensible and familiar. There is a clear logic to the basic constructs of PostScript. You define your space (or use the default space) according to the Cartesian coordinate system. You then create your shape or line (using that defined space) by constructing a path and stroking it with options as to color, line width, style, etc. The process is similar to beginning a drawing on paper. You have a defined space, the paper size; you decide the bottom, left, right, and top of that space; you begin to draw choosing your color, pencil weight, line type, etc.

This initial contact with PostScript was positive and met my early expectations of the language; it also made me much more appreciative of the language’s potential. I was also aware that, given the time frame within which I had to work (just two and one half months), I would be unable to explore many of the possibilities of this powerful language. I chose then to concentrate on a limited number of operators and develop procedural control over their use. It was easy enough to create a graphic object in PostScript; I wanted to explore the manipulation and visual complication of that object. This involved constructing a step-by-step method of defining a basic shape (the procedure) and then varying it in size, shape, placement, rotation, number, and color through the use of other procedures and PostScript operators.

These first programs took a very long time to process. These were not typical page descriptions; they demanded many calculations to form the images—one uses a random number generator to place hundreds of tonally graduated spheres with clipping and large areas of black. This was a frustrating aspect of working with PostScript; twenty to thirty minutes of processing time per image was not uncommon for some of these programs. This slowness could have been improved through more efficient methods of procedural use.² PostScript processing itself was slow in its

²As a novice PostScript programmer, I was unaware of certain structuring styles and operators which would have improved the processing time. From recent readings in two
original implementation in the Apple LaserWriter® and Adobe has since streamlined the language several times; hardware has also been improved as to processing speed and memory capacity.³ Nonetheless, a wait time, however minimized, still exists in the printing of graphics.

Another area I found to be problematic involved the printing of blacks. This was not a software (PostScript) problem per se but a hardware (printer) problem. Large areas of black could not be printed evenly. They would gray-out with streaks. This had to do with the Canon marking engine used in the original LaserWriter and the manner in which it printed blacks.⁴ In order to create the solid blacks I needed for the thesis exhibit images, I had to print the image twice. Added to this inconvenience was the lack of an exact registration method. Interesting results were generated but it was not the best solution. Small areas of black, however, as in text or linear graphics, would print without the gray streaking.

Two other hardware related limitations existed in this medium. These had to do with the size and color of output. The LaserWriter is a black and white printer which requires 8 1/2 x 11 inch paper. In that I wanted to develop a large color image for my visual thesis I felt these limitations as restrictions and experimented with different approaches to overcome them.

I tested different papers from hand-made to glossy cover stock. The toner did not adhere well to many of these; its surface would rub off. The coated

³The PostScript version I am currently working with is #47, on an AST Research, Inc. TurboLaser®/PS.

⁴There have been several enhancements made to this original Canon engine which have improved upon this side effect. Other types of marking engines have also been developed; one in particular by Ricoh, Inc., lays down the toner differently than the Canon does. This produces a heavier black which works better for large area coverage. It doesn’t, however, produce as fine a line as the Canon engine. The printer used to output this written thesis report, the AST TurboLaser®/PS, has a Ricoh engine. Appendix B contains examples of output from the Apple Computer, Inc. LaserWriter® and the LaserWriter® IIINTX for comparison purposes.
stocks worked best (other than the standard copy paper and laser paper) from the many types which I tried. I used a special color film, sold as Omnicrom®. It is normally used with a heat sealing unit. However, since the laser printer uses heat in its adherence of toner to paper, the printer could be substituted as the “sealer”. One would print out the color-intended piece in black, place the color film on top, and run the two sheets through the printer together with the PostScript showpage command. This command, by itself, sends a piece of paper through the printer without marking it. The color from the sheet adheres to the black toner. The output from the process I was using was not precise for large areas of color. It was difficult to keep the film and paper together; wrinkling and irregular color lay-down would occur. I looked to another more uniform method. There were color toner cartridges available which could replace the black toner cartridge in the printer. However, I was unable to locate these. Eventually I obtained color in the output imagery by printing on color-coated stock.

Through the reversing of negative and positive space, I could obtain very different visuals from the same basic programs: I could “color” the objects or the background by changing the “painting” color (white, shades of gray, black) and defining the clipping path. The programs in listings 4.2 and 4.3 below demonstrate this difference. Examples of their output are shown in figures 4.2 and 4.3 below. A procedure to draw a filled rectangle can be called with different gray values, as can the outline and fills of the character. The font as well as the character string can be determined in the script allowing the program to be used with greater flexibility than if these arguments were set in the prologue.

In listing 4.2 the first three arguments of procedure “b” define the weight of the line, the color of the line, and the color of the fill. The script of listing 4.2 sets a linewidth of .5, a line color of black, and a fill color of white, while the script in listing 4.3 sets a fill of black. The drawing procedures (“os”, “tc”) are each contained within a gsavel/grestore bracket which saves and then resets the graphics state to its previous state. This allows each procedure to be independent. Procedure “b” draws a rectangular path using relative coordinates (rlineto) based on a width and height and the location of the lower left corner (LLx and LLy). It makes use of the stack operators index and exch to manipulate the data on the stack and create a more efficient and globally applicable procedure; it is a good example of programming modularity.5 The appropriate arguments can be changed to produce whatever size and shape rectangle is desired. Procedure “f” sets the typeface and pointsize which can be varied when “f” is called. Procedure “os”, using the PostScript boolean operator, true, and the path

construction operator, charpath, creates an outline of the character(s) suitable for filling or clipping but not for stroking. The boolean value determines what type of outline to leave. Procedure “tc” calls “os” for each rotation specified in the call. The PostScript rotate operator is put within a for loop and uses as its chosen angle of rotation the loop’s current value. The character(s) will be filled with the color code passed to “os”. In the script, these procedures are called with the appropriate and desired arguments. Listing 4.3 is an example of how a script can be changed to alter the output of a prologue. The line and fill colors of both background and characters are changed as well as are the fonts, point sizes, and character strings.

---

listing 4.2  Rotated Text on White

(script prints black outlined characters)

%!PS-Adobe-2.0
%!Title: txt-cir-w.ps
%!EndComments

/b{%def
moveto
1 index 0 rlineto
0 exch rlineto
neg 0 rlineto
closepath
gsave setgray fill grestore
setgray setlinewidth stroke
}bind def

/f{%def
findfont exch scalefont setfont
}bind def

/os{%def
%"os" prints outline of string's characters
%arguments: lineweight, linegray, fillgray
%string, LLx, LLy
gsave
moveto
true charpath
gsave setgray fill grestore
setgray setlinewidth stroke
grestore
}bind def

/tc{%def
%rotates coordinate system; calls "os"
%arguments: startangle, increment,
%lineweight, endangle,
%linegray, fillgray, string, xc, yc
/y exch def
/x exch def
/strg exch def
/sfg exch def
/slg exch def
/slw exch def
{/bind def

%begins for loop
%initial, incremental, and limit values of
%for loop are passed as arguments
%begins prologue
%defines a rectangle
%arguments: lineweight, linegray,
%fillgray, width, height, LLx, LLY
%sets placement of first point (LL)
%sets second point (LR) relative to first
%sets third point (UR) relative to second
%sets fourth point (UL) relative to third
%closes path from fourth to first point (LL)
%sets fill color
%sets color, lineweight, strokes line
%ends "b"

%"f defines font and point size
%arguments: pointsize, font
%ends "f"

%"os" prints outline of string's characters
%arguments: lineweight, linegray, fillgray
%string, LLx, LLy
gsave
moveto
true charpath
gsave setgray fill grestore
setgray setlinewidth stroke
grestore
}bind def

%begins for loop
%initial, incremental, and limit values of
%for loop are passed as arguments
%ends "os"
%calls PostScript "rotate" operator, uses % for loop arguments for degree values %calls "os" with arguments in variables %restores previous graphics state %ends "rotation" for loop %end "tc"
/tsc{%def
translate
scale
}bind def
%translates and scales coordinate system %arguments: x scale, y scale, x, y %calls PostScript "translate" operator %calls PostScript "scale" operator %ends "tsc"

%%EndPrologue %begins script 1 1 306 400 tsc % translates and scales coordinate system 24/Helvetica f %passes arguments (24 point size, %Helvetica font) calls "f" 0 20 340 .5 0 1 ( Adobe PostScript) 0 0 tc %passes arguments, calls "tc" 60/Helvetica-Bold f 2 0 1 (RIT) 0 0 os %calls "f" with new arguments %passes arguments, calls "os" %outlines with black; fills with white %prints out page showpage %ends script

listing 4.3  Rotated Text on Black, script

(script prints white outlined characters on a black rectangle)

%%%EndPrologue %begins script 1 0 0 612 792 0 0 b %passes arguments, calls "b" 1 1 306 400 tsc % translates and scales coordinate system 24/Times-Roman f %passes arguments (24 point size, %Helvetica font) calls "f" 0 20 340 .5 1 0 ( Rochester Institute of Technology) 0 0 tc %passes arguments, calls "tc" 60/Times-Roman f %calls "f" with new arguments 2 1 1(CFAA) -22 -20 os %passes arguments, calls "os" %outlines with white; fills with white %prints out page showpage %ends script
figure 4.2  Rotated Text on White

(output of listing 4.2, Rotated Text on White)
To overcome the limitation of the standard 8 1/2 x 11 paper size I worked on a PostScript program which would create a large image and divide it into the proper number of standard sheets. These sheets could then be assembled to form the larger sized image. I began to experiment with this concept in order to be able to create a poster size image for the MFA exhibit. This initial "poster" program output an enlarged version of the RIT logo and inspired a solution for the exhibition piece.

I was concentrating on three types of image generating programs (sphere patterns, RIT logos, and character transformations) simultaneously—manipulating their procedures and variables while learning PostScript. They were, however, disparate groups. The following listings (4.4 - 4.12) and figures (4.4 - 4.7) are each examples of one of the many variations of the respective program types. I decided to create a collage of the three groups of imagery to form a mural-size piece. I printed on a combination of papers (laser, handmade, and color-coated stock) for a variety and richness of texture and color. It was an exciting concept for I had not worked on such a large scale since I began my involvement with computer graphics.

My thesis research into the PostScript programming language was a very intense and creative process. It generated more ideas than I had time to develop. This was frustrating but it has allowed me to continue the development of PostScript programming with interest and involvement.

---

7 Most graphics software applications which use PostScript, such as Cricket Draw™ and Adobe Illustrator® now have this feature included in their options.

8 The last example in Appendix A is output from a variation on this "logo poster" program. It consists of six 8 1/2" x 11" sheets. This example uses clipping for the placement of text within the logo itself and a variation of the "radiating spheres" program for the background.

9 Reference Appendix A for output examples of other variations.

10 The finished piece was comprised of 6 panels each three feet high by four foot wide, stacked in two rows of three each; overall measurement: 6'H x 12'W.

11 Appendix C contains color copies of slides of the Thesis Exhibition Mural.
listing 4.4 Random Spheres, prologue

(procedures to create random spheres)

%!Adobe-2.0
%!Title: ran-sph.ps
%!EndComments

%prologue

/rg{%def
rand
1000 mod
1000 div
}bind def

/sp{%def
{It
20
rg
mul
def
/y
792
rg
mul
def
/x
612
rg
mul
def
}

r
8
ge{
0
1
9
{
9
div
setgray
/x
.x
3
add
def
/y
.y
3
add
def
r
16
ge{/r
1.6
sub
def}if
r
12
ge{/r
1.2
sub
def}if
{/r
.8
sub
def}ifelse
x
y
r
0
360
arc
fill
}for
}if

}repeat

}bind def
The *script* portion of the above program is shown below in *listing 4.5*. This is the executable code of the program: it calls "sp" which calls the random procedures and creates the number of shaded spheres specified in the argument to its call. *Showpage* prints the current page. The printed output from combining the *prologue* in *listing 4.4* and the *script* from *listing 4.5* is shown in *figure 4.4*.

---

**listing 4.5**  **Random Spheres, script**

(draws random tonally gradated spheres on full page)

```
%%EndPrologue
1000 sp
showpage
%%Trailer
```

%begins *script*
%calls "sp" with *repeat* loop argument
%prints the page
%ends *script*

---
figure 4.4  Random Spheres

(output of listing 4.4, Random Spheres, prologue and listing 4.5, Random Spheres, script)
If I added the procedures in listing 4.6 below to the prologue of listing 4.4, I would then have the option of calling them from the script (which I would type in, interactively, from the keyboard) to vary the output. They each define a different geometric clipping path—diagonal, diamond-shaped, triangular, circular. These path definitions are used by the PostScript clip operator to define a region within which other graphic output can be printed and, conversely, outside of which it cannot be printed. This determines where the graphics will or won’t print. Any path can be used in PostScript as a clipping path. The use of the clip operator can cause unexpected results because it appends the current path to the current clipping path and does not destroy it as do other operators such as the paint operators stroke and fill. It is usually a good practice to initialize a new path after using the clip operator with a call to the newpath operator. In the listed procedures newpath is used after the clipping path is defined. The interactive scripts would then be as follows in listings 4.7, 4.8, 4.9, and 4.10 below. The calls to the procedures are preceded by the arguments which define the x and y locations of the points needed by the moveto, rlineto, and arc operators as well as the required angles and radius in the case of the call to the circle procedure. Figure 4.5 is the printed output when the program is executed with listing 4.10.

\[Procedure\]

The last example in Appendix A uses clipping to fill the RIT logo with repeating lines of text which was customized using a PostScript program. The title pages of the appendices also make use of this clipping procedure.
**listing 4.6 Geometric Clipping Paths**

(defines four paths: diagonal, diamond-shaped, triangular, circular)

```
/dclp {%def
  moveto 0 rlineto 0 exch rlineto closepath
  clip newpath)
}bind def

/diclcp {%def
  moveto 1 index 1 index rlineto
  1 index 1 index neg rlineto
  exch neg exch neg rlineto closepath
  clip newpath)
}bind def

tclp {%def
  moveto 1 index 0 rlineto
  exch 2 div neg exch rlineto closepath
  clip newpath)
}bind def

cclp {%def
  0 360 arc clip newpath
}bind def
```
Listing 4.7  Diagonal Clip, script

(clips a diagonal path/fills with spheres)

\[
\begin{align*}
%\%\text{EndPrologue} & \quad \text{%begins script} \\
792 & 612 \ 0 \ 0 \ \text{dclp} \quad \text{%calls dclp with arguments} \\
1000 & \text{sp} \quad \text{%calls sp with repeat loop argument} \\
\text{showpage} & \quad \text{%prints the page} \\
%\%\text{Trailer} & \quad \text{%ends script}
\end{align*}
\]

Listing 4.8  Diamond-Shaped Clip, script

(clips a diamond-shaped path/fills with spheres)

\[
\begin{align*}
%\%\text{EndPrologue} & \quad \text{%begins script} \\
234 & 324 \ 72 \ 390 \ 0 \ 0 \ \text{diclp} \quad \text{%calls diclp with arguments} \\
1000 & \text{sp} \quad \text{%calls sp with repeat loop argument} \\
\text{showpage} & \quad \text{%prints the page} \\
%\%\text{Trailer} & \quad \text{%ends script}
\end{align*}
\]

Listing 4.9  Triangular Clip, script

(clips a triangular path/fills with spheres)

\[
\begin{align*}
%\%\text{EndPrologue} & \quad \text{%begins script} \\
468 & 648 \ 72 \ 72 \ \text{tclp} \quad \text{%calls tclp with arguments} \\
1000 & \text{sp} \quad \text{%calls sp with repeat loop argument} \\
\text{showpage} & \quad \text{%prints the page} \\
%\%\text{Trailer} & \quad \text{%ends script}
\end{align*}
\]

Listing 4.10  Circular Clip, script

(clips a circular path/fills with spheres)

\[
\begin{align*}
%\%\text{EndPrologue} & \quad \text{%begins script} \\
306 & 396 \ 300 \ \text{cclp} \quad \text{%calls cclp with arguments} \\
1000 & \text{sp} \quad \text{%calls sp with repeat loop argument} \\
\text{showpage} & \quad \text{%prints the page} \\
%\%\text{Trailer} & \quad \text{%ends script}
\end{align*}
\]
Figure 4.5  Random Spheres with Circular Clipping Path

(output of listing 4.4, Random Spheres, prologue,
listing 4.6, Geometric Clipping Paths,
and listing 4.10, Circular Clip, script)
listing 4.11 3-D RIT Logo

(draws RIT logo in a three-dimensional form)

%IPSA:doe-2.0
%%Title:3d-rit-logo.ps
%%EndComments

%beings prologue

/lgb{ %draws a filled RIT logo shape
%arguments: lineweight, line gray,
%fill gray, the five basic lengths of
%logo, LLx, LLy
moveto 0 Index 0 exch rlineto
0 rlineto 0 exch neg rlineto
2 Index neg 0 rlineto
1 Index 0 exch rlineto
0 index neg 0 rlineto
0 exch neg rlineto
0 rlineto
0 exch neg rlineto
closepath

gsave setgray fill grestore
setgray setlinewidth stroke

}bind def

/3d{ %sets variables for x, y, points, gray value
%calls "lgb" with arguments
/x 0 def %variable for x value
/y 0 def %variable for y value
/g 1 def %variable for gray value
/a 216 def %variable for length #1
/b 144 def %variable for length #2
/c 174 def %variable for length #3
/d 123 def %variable for length #4
/e 21 def %variable for length #5

19{ %begins repeat loop which changes gray
%values and x and y values of first point
%for middle and inner logo shapes based
%on original values for outer logo shape
}repea

}%ends "3d"

30
/g1 g .1 sub def %gray value for middle shape
/x1 x 25.5 add def %x value for middle shape
/y1 y 25.5 add def %y value for middle shape
/g2 g .2 sub def %gray value for inner shape
/x2 x 51 add def %x value for inner shape
/y2 y 51 add def %y value for inner shape

.2 g g e d c b a x y lgb %calls "lgb" with arguments for outer shape
.2 g1 g1 e d 25.5 sub c 51 sub b 25.5 sub a 51 sub x1 y1 lgb %calls "lgb" with arguments for middle shape
.2 g2 g2 e d 51 sub c 102 sub b 51 sub a 102 sub x2 y2 lgb %calls "lgb" with arguments for inner shape

/g g .04 sub def %decrements initial gray value by .04 points
/x x .84 sub def %decrements initial x value by .84 points
/y y .84 sub def %decrements initial y value by .84 points
/repeat %ends repeat loop

.2 0 0 e d c b a x y lgb %calls "lgb" to fill outer shape with black
.2 1 0 e d c b a x y lgb %calls "lgb" to stroke outer shape w/white

.2 0 0 e d 25.5 sub c 51 sub b 25.5 sub a 51 sub x1 y1 lgb %calls "lgb" to fill middle shape with black
.2 1 0 e d 25.5 sub c 51 sub b 25.5 sub a 51 sub x1 y1 lgb %calls "lgb" to stroke middle shape w/white

.2 0 0 e d 51 sub c 102 sub b 51 sub a 102 sub x2 y2 lgb %calls "lgb" to fill inner shape with black
.2 1 0 e d 51 sub c 102 sub b 51 sub a 102 sub x2 y2 lgb %calls "lgb" to stroke inner shape w/white

/bind def %ends "3d"
/tsc{
  translate
  scale
/bind def

%%EndPrologue %begins script
1 1 200 275 tsc %sets coordinate system, calls "tsc"
3d %calls "3d"

%%Trailer %ends script
figure 4.6 3-D RIT Logo

(output from listing 4.11, 3-D RIT logo)
listing 4.12  Alphabet Blocks

(creates characters on clear blocks)

%!PSAdobe-2.0
%Title:alphablocks.ps
%EndComments %begins script

/mb{%def
    moveto
    1 index 0 rlineto
    0 exch rlineto
    neg 0 rlineto
    closepath
    gsave stroke grestore
}bind def

/ms{%def
    [.5 .29 0 1 0 0] concat
    120 120 0 0 mb
}bind def

/mc{
    120 120 0 0 mb
    gsave
    120 0 translate
    ms
    grestore
    gsave
    ms
    grestore
    gsave
    60 34.8 translate
    120 120 0 0 mb
    grestore
}bind def

%ends script
% calculates (x) position of letter using PostScript charwidth operator
/sx{
  % sets variable for length of cube plane
  /lx exch def
  /dx lx 2 div charwidth 2 div sub def
  % calculates distance from center of plane to start (x) position of letter
  % places value in variable “dx”
  % ends "sx"
}bind def

/wck2{
  charwidth 60 le
  {/x charwidth def /y 30.4 def}
  {/x 60 dx sub def /y 36.4 def}ifelse
  % adjusts as necessary and
  % sets (x) and (y) position on side 2
  % ends "wck2"
}bind def

/wck4{
  charwidth 60 le
  {/x charwidth 2 div 120 add def /y charwidth 2 div def}
  {/x 120 dx add def /y 10.4 def}ifelse
  % adjusts as necessary and
  % sets (x) and (y) position on side 4
  % ends "wck4"
}bind def

/bk{
  % determines character transformation
  % and placement in cube
  gsave
  % saves current graphics state
  .2 setgray
  % sets gray value; bottom of cube (side 1)
  ft findfont [pt neg 0 69.3 40 0 0] makefont setfont
  % transforms character with skew on matrix
  120 sx 120 dx sub 3.4 moveto ch show
  % calls "sx" to determine (x) position of letter
  grestore
  % restores graphics state

  gssave
  % saves current graphics state
  .4 setgray
  % sets gray value; left side of cube (side 2)
  ft findfont [pt neg 2 div pt neg 2 div .577 mul 0 145 0 0 ] makefont setfont
  % transforms character with skew on matrix
  120 sx wck2 x y moveto ch show
  % calls "sx" and width check procedure
  % to determine (x) position of letter
  grestore
  % restores graphics state
gsave %saves current graphics state
.6 setgray %sets gray value; back of cube (side 3)

ft findfont [pt neg 0 0 150 0 0] makefont setfont
%transforms character with skew on matrix

120 sx 180 dx sub 40.8 moveto ch show
% calls "sx" to determine (x) position of letter
grestore %restores graphics state

gsave %saves current graphics state
0 setgray %sets gray value; right side of cube (side 4)

ft findfont [pt 2 div pt 2 div .577 mul 0 145 0 0] makefont setfont
%transforms character with skew on matrix

120 sx wck4 x y moveto ch show %calls "sx" and widthcheck procedure
%to determine (x) position of letter
%restores graphics state

grestore
gsave %saves current graphics state
%front of cube (side 5)

ft findfont [pt 0 0 150 0 0] makefont setfont
%transforms character with skew on matrix

120 sx 0 dx add 6 moveto ch show %calls "sx" to determine (x) position of letter
grestore %restores graphics state

gsave %saves current graphics state
%top of cube (side 6)

ft findfont [pt 0 69.3 40 0 0] makefont setfont

120 sx 0 dx add 123.4 moveto ch show
grestore %restores graphics state

mc %calls procedure to draw wireframe cube
)bind def %ends "bk"
% determines width of letter and scales it
% if necessary to fit in plane of cube
% arguments: point size, character

gsave
/ch exch def
/pt exch def
/ft exch def

ft findfont pt scalefont setfont

% saves graphics state
% defines character for block (ch)
% defines point size of character (pt)
% defines font (ft)

/charwidth ch stringwidth pop def

charwidth 108 ge
{gsave
108 charwidth div
/sc exch def
/pt pt sc mul def
}

/grestore

% calculates width of character; stack: x
% begins if/loop (if too wide(108), then scale)
% saves graphics state
% 108 divided by character width
% becomes scale factor (sc)
% point size becomes original point size
% multiplied by scale factor
% character width becomes original width
% multiplied by scale factor
% restores previous graphics state
% ends if/loop
% restores previous graphics state

/grestore

% binds "chsc"

%%End Prologue

gsave

125 500 translate 10 rotate
/Helvetica-Bold 168 (R) chsc bk

/grestore

% saves current graphics state
% translates and rotates coordinate system
% calls "check width" procedure with point
% size and character; calls "bk"
% restores graphics state

/gsave

200 340 translate - 20 rotate
/Helvetica-Bold 168 (I) chsc bk

/grestore

% saves current graphics state
% translates and rotates coordinate system
% calls "check width" procedure with point
% size and character; calls "bk"
% restores graphics state

/gsave

350 110 translate
/Helvetica-Bold 168 (T) chsc bk

/grestore

% prints page

%%Trailer

% ends script
figure 4.7  Alphabet Blocks

(output from listing 4.12, Alphabet Blocks)
Chapter 5

POSTSCRIPT PROGRAMMING
GUIDELINES FOR THE ARTIST
AND DESIGNER

My intent within this chapter is not to teach PostScript programming but to make suggestions and note areas of importance and consideration. My programs in this written thesis report are not presented as examples of the best PostScript programming style; they are early programming attempts in the learning of the language. The combination of my background as an artist and designer and my efforts to learn rudimentary PostScript programming have resulted in some insight and experience which could be of value to someone of similar background and inclination. The person who is interested in learning PostScript should read, study, and use the texts listed in the Bibliography for a more detailed and thorough explanation of the language and use this report as a reference to the language from a particular viewpoint.

5.1 COORDINATE SYSTEMS

Review your knowledge and understanding of (or learn) the Cartesian coordinate system. Transformations are relative to this system and PostScript path construction is based on point locations. Absolute and relative coordinates are used by the PostScript operators lineto and rlineto. An understanding of their differences can make their appropriate uses more effective and result in a more efficient program. Remember that PostScript defines two coordinate systems: user space and device space. User space is what it says: it’s yours to define and use. The manipulation of this space can yield graphic effects which would be much more difficult if not impossible to obtain through traditional tools. Rotating is a simple procedure (listing 4.2, procedure “tc”; figures 4.2 and 4.3) as is skewing text or graphics (listing 4.12, procedure “ms”; figure 4.7).

5.2 STACKS AND POST-FIX NOTATION

Read about PostScript stacks and post-fix notation. Practice performing simple mathematical operations using both. When working with several operators and their operands it is advantageous to diagram the stack usage, as in figure 5.1. This is helpful in avoiding confusion. Unusual results are often due to improper stack manipulation. What is on top of the stack is
executed first. The contents of a PostScript stack are typically printed with the top of the stack to the right, as in figure 5.1.1 As my programs became more complex, I continued to find this method helpful in sorting out the state of the operand stack. The “backwards” thought processing involved in these manipulations goes against the norm. Only after I was immersed in the language for a time did this method of thinking become second nature.

---

**figure 5.1 Stack Operations**

(from listing 4.3, arguments from call to “b”)

<table>
<thead>
<tr>
<th>1 0 0 612 792 0 0 b</th>
<th>0</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Arguments and call</td>
<td>792</td>
<td>Stack</td>
</tr>
<tr>
<td>to procedure “b” ;</td>
<td>612</td>
<td></td>
</tr>
<tr>
<td>arguments are placed</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>on stack as called,</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>beginning from left.</td>
<td>1</td>
<td>Bottom</td>
</tr>
</tbody>
</table>

---

5.3 OPERATORS

Begin your programming attempts with simple operations. Imitate what you read. Seeing the process at work and generating output is the best teacher. You can then begin to experiment with those basic operators and add other operators to your PostScript vocabulary. This will help to expand the variation possibilities. The *PostScript Language Tutorial and Cookbook* presents a well-structured, logical path to the development of an understanding of the workings of the language. It contains simple code

---

that is expanded upon step by step. David Holzgang’s book, Understanding PostScript Programming, is very informative and is also a good reference. The Cookbook, I feel, is more appropriate for the artist/designer in its attention to graphics programming and its visual approach to description and explanation. Familiarize yourself with path construction operators (moveto, rlineto, arc), painting operators (fill, stroke), and text operators (findfont, scalefont, show) before adding other concepts such as transformations or clipping paths.

5.4 STYLE

As you now develop more involved programs look for repetitive patterns that you can distill into condensed procedures. Simplify. Think modularly. Glenn Reid recommends the usage of brief identifiers such as /F for a font function because it lessens transmission time.² End a procedure with the bind operator, mentioned above, as in ]bind def. I have incorporated this suggestion into some of my earlier programs. The Reid book, although aimed at application developers, has many worthy and helpful examples of good PostScript programming style as well as important observations on the language’s functioning.

5.5 HARDWARE AND SOFTWARE CONFIGURATION

There are, as mentioned earlier, two methods of sending PostScript program code to the printer: batch and interactive. A batch file can be created and sent by application programs such as Microsoft® Word or Aldus PageMaker® or by the individual programmer. This file is sent over a communications cable: AppleTalk® on the Macintosh® or parallel or serial on a PC setup. Different printers have different combinations of communications ports which would determine the cabling.

If you wish to send a file containing your own page description, you would create the file in a text editor and save it as text (ASCII code). You must then determine the appropriate manner in which to send it to the printer. The printer’s manual will usually explain how to do this. An application program, a communication program, or, in some cases, a DOS (Disk Operating System) command on a PC can accomplish this.

To program interactively in PostScript, you must setup a communications channel that allows you to receive (from the printer’s controller) and send (from the host computer) PostScript program code. Your computer becomes the host through which you “talk” to the PostScript interpreter in the printer. A simple communication program will suffice; it is only used for basic communication purposes. I’ve used ProComm® on a PC and MacTerminal® on a Macintosh II®. When you have installed the proper cable configuration, you set the communications settings, typically at 9600 baud, no parity, 8 data bits, 1 stop bit (9600, n, 8, 1). You then type executive, lower case, and enter it. You won’t see it on the screen because the screen echo should be turned off. Once you enter the command you should receive a message from the printer similar to the one below.

PostScript® Version 47.0
Copyright©1985 Adobe Systems Incorporated
PS>

This indicates that you are “connected” and can start issuing your PostScript commands. As mentioned earlier, I use a method which combines sending the prologue part of a file with typing the executable commands directly. This seems an efficient method for debugging and for trying variations. The printer will communicate errors in your program as it scans the program.

Setting up the system—the communications between the computer and the printer—can be a frustrating experience. The Macintosh® configuration, perhaps because it is more straightforward, seems to be consistently easier than a PC configuration. Although the guidelines for setting up both systems are similarly documented, I had more problems on the PC system than on the Mac. Again, the specifics of settings and cables can be found in the texts mentioned in the Bibliography and in the printer’s manual.3

5.6 PROGRAMMING

Have fun with the language. PostScript offers great flexibility in manipulating text and graphics, so experiment with it to get a feel for what it can do. As with any programming language, PostScript is unforgiving in its

3David Holzgang’s book, Understanding PostScript Programming, has excellent sections on the configuration for both batch and interactive communication on both types of systems.
syntactical rules (the spelling of names, appropriate use of delimiters—{}, [], /%, etc.). Due to its directness, however, PostScript is an easy language with which to experiment and to become involved. You can test code line by line and get immediate results. It speaks graphics and works wonders with them.
Appendix A

PRINTED POSTSCRIPT OUTPUT FROM PROGRAM VARIATIONS
Appendix B

PRINTED POSTSCRIPT OUTPUT FROM APPLE COMPUTER, INC.
LASER PRINTERS
B.1 APPLE, INC. LASERWRITER® POSTSCRIPT OUTPUT

The following output examples are originals produced during my thesis research at RIT. Some variation exists in the examples included in the different copies of this written thesis report as I was unable to have access to the LaserWriter® to reprint the programs.

I've included the examples for comparison with the printed output in Appendix A which were printed on the AST TurboLaser®/PS. As you see, the blacks in the examples which follow do not have the richness and covering power of the those in Appendix A. This is why I found it necessary to double-print the originals for the thesis exhibit mural. The last examples in this section (B.1) are ones which were printed twice.
The printed output included in this section was printed on the new Apple, Inc. laser printer, the LaserWriter® IIINTX. This printer uses a new version of the Canon marking engine found in the original LaserWriter®. It does an improved job in solid area coverage (blacks and grays). However, if you compare the following examples with those in Appendix A, you will see less coverage and some streaking. As footnoted in the text (page 15), the Ricoh engine in the AST TurboLaser®/PS uses a different method of laying down the toner. It would appear that this gives better results if one's interest is in printing areas of gray or black of a size larger than text or small graphics. The LaserWriter IIINTX, however, has a good deal more processing speed than the AST TurboLaser/PS. It's 68020 processor is the same processor that the Macintosh®II uses. This is of great advantage when printing PostScript code. I noticed a marked increase in the processing time of the programs included in this report when I printed them using the LaserWriter IIINTX.
CFAA

Chester Institute of Technology
BIBLIOGRAPHY

BOOKS


NEWS PUBLICATIONS


ARTICLES

Burns, Diane, and S. Venit. "The premier page-composition language; PostScript, the language used by Apple’s LaserWriter, may become the industry standard because it’s flexible and has logical commands." PC Magazine, 10 December 1985, 209-10.

Canon, Maggie, and Frederic E. Davis. "The Mac Office; Apple’s new office-automation system can connect Macs, Lisas, Apple IIs, IBM PCs, printers, file servers, and many more other devices." A+, March 1985, 26-33.


Jeffries, Ronald K. "An important PostScript: PostScript, a printer-control language, is a flexible and powerful means of bringing images and text to paper. To date, only the Apple LaserWriter uses it, but others will follow." *PC Magazine*, 17 September 1985, 77-80.


Mello, Adrian. "Rounding the bezier curve. (Cricket Draw program points the way to the future of Macintosh graphics applications." *Macworld*, May 1987, 148-54.


_________. "Shaping PostScript to your bidding; the second part of a four-part tutorial on using the language of the LaserWriter printer." *A+*, July 1985, 61-64.


_________. "PostScript: the medium and the message; the fourth part of a four-part tutorial on using the language of the LaserWriter printer." *A+*, September 1985, 81-84.
