PostScript

Sushanth K Reddy and Ravi Sheshu Nadella

Contents

1 Introduction

Postscript is a programming language and a page description language used mainly for publishing via printers. Postscript is much more than a printer control language. It is a programming language by itself. Applications can transform a document into a PostScript program which can be sent to a printer. The interpreter in the printer can interpret the program and print the document. A Postscript program can also be used to display a document on the screen. The power of postscript to generate the same document on different devices using the same program makes it a device-independent language.

2 History

The basic idea/concepts of Postscript were developed in 1976 when John Warnock conceived the Design System Language to process graphics. After Adobe was found in 1982, Postscript was created as a simpler version of Interpress, which was then the existent language for graphics printing. Around 1985 it was used as the language to drive laser printers. An interpreter for Postscript referred as the Raster Image Processor was a component of the laser printers in the 1990s. The wide spread availability of Postscript made it the de Facto standard for electronic document distribution for final versions of documents meant for publication. There were two upgrades for the language.

The first upgrade improved speed and reliability, included support for in-RIP separations, decompression of images, support for composite fonts, and form mechanism for caching reusable content. The second upgrade came along with many new dictionary-based versions of older operators, improved color handling, new filters to allow in-program compression and decompression, program chunking, and advanced error-handling.

3 Printing

Before Postscript, printers were designed and used to print only text (ASCII characters). A number of technologies existed for this task. However, most of them had the properties of a standard type writer (glyphs). The main problem was that, characters could not be changed easily in terms of font and size. And also Pictures could not be printed. With the advent of Dot Matrix Printers this changed. Dots were used to print characters and so different fonts could be printed from tables stored in the printer. These tables could also be uploaded by the user which facilitated introduction of more fonts. Dot Matrix Printers printed characters as series of dots, thus making them able to print raster graphics. However to print vector graphics, plotters had to be used. Also printer control languages were different for different printers. This made the tasks tedious for authors as they had to write different drivers for different printers.

3.1 Postscript Printing

Laser Printers could print high quality art, text and raster graphics. They could include text and graphics on the same page. Postscript exploited this capability by providing features to support both text and graphics and also offering a single control language for different brands of printers. Also the postscript program used to print a document can be used to display it on-screen making postscript a device independent language. Postscript has the feature of on-the-fly rasterization which gives it the flexibility for scaling, rotation and other transformations.

3.2 Display Postscript

One language which used Postscript as a display technology on screen was Display Postscript (DPS). DPS uses the language and the imaging model of Postscript to generate graphics on screen. Some changes were made to Postscript to support on-screen model. These changes include procedure addition to handle interaction and the support for multiple execution contexts.

4 Language

4.1 Interpreter

The language postScript is interpreted and does not generate any intermediate code. It is a stack based language and mainly designed for graphics and typography. The reason this language is interpreted, is mainly to support device independence. As long as it is interpreted, one need not worry about the object file format on the machine where it is compiled and if it would work on any other machine or not. An interpreter of the language can read the same program on any machine.

4.2 Objects

Data in Postscript is stored in the form of Objects. Each object has type, attributes and value. All objects are of same size immaterial to the size of data. Variable length data is stored in a separate location and the pointer to that location is stored as the value of the object.

4.3 Dictionaries

All the names and values of objects in Postscript are stored in Dictionaries. There are three basic kinds of dictionaries called system dictionary, global dictionary and user dictionary which store system related names, global names and local names respectively. Also dictionaries can be created inside procedures to enforce scoping. Whenever a name is encountered during execution it is checked in the appropriate dictionary on the dictionary stack.

4.4 Stack Based Execution

The execution of Postscript is stack based. And the notation used to write statements is postfix notation. All the operands of any operation are pushed onto the stack. The operator then pops out all the necessary operands and pushes the result.

There are mainly four stacks in Postscript.

- Operand Stack
- Dictionary Stack
- Graphics Stack
- Execution Stack.

4.4.1 Operand Stack

This is the stack used mainly by many operators in Postscript. This is the stack on which operators find the operands and put results. This is the main work horse of the execution of a postscript program.

4.4.2 Dictionary Stack

This is the stack on which dictionaries are placed. The three dictionaries at the bottom of the stack are system systemdict (system dictionary), gloabaldict and userdict. This stack is used for name/object searching and scoping. The current set of dictionaries on the stack determines the environment for name searches. A dictionary can be placed using the begin operator and has to popped off using the end operator.

4.4.3 Graphics Stack

This is the stack on which the current graphics state can be pushed and later restored. The use of operators like 'fill', 'stroke' etc changes the current graphics state. To return back to the original state, the save and restore feature can be used.

4.4.4 Execution Stack:

This is analogous to the call stack in languages like C. Whenever the execution of current object needs to be suspended (to execute a new object), the new object is pushed on top of the current object. When the execution of new object is done, it is popped off and the execution of the current object is resumed.

5 Data Types

5.1 Classification

Postscript is a strongly typed language. There are different data types like int, real, string and Array. Types like integer, real, string are called simple types and types like array and dictonary are called complex types. There are also operators provided for conversion between some data types. All data objects are statically typed. A variable is defined using the operator 'def'.

Example:
An integer is defined by the statement: **/a 30 def**
An array can be declared as: **/exarray 10 array def** or implicitly as: **/ar [1 2 (ravi) 4.5] def**
Dictionary objects can be declared to hold name value pairs like **<<1 (luca) 2 (ppl)>>**

5.2 Equivalence

The equivalence of two objects in Postscript can be checked through the operator 'eq' which pops two objects off the stack and pushes a Boolean value as the result. Two simple objects are equal if the types and values match. However sometimes type conversions are done during equality checks.

For example, Integers and Reals are equal if they hold the same value.
1 1 eq – true and **1 1.0 eq -true**

Two complex objects are equal only if they point to the same value. Two complex objects are not equal even though they contain the same value.

Example:

[1 (ravi) 3] dup eq-true (dup creates a another pointer to the same object)

[4 5] [4 5] eq – false (because both are two different objects)

5.3 Assignment

When one simple Object is assigned to another, the value gets copied. However when one complex object is assigned to another, the pointer to the location of the value gets copied.

/a 23 def, **/b a def** (a=23 and b=23)

/a [1 2 3] def, **/b a def** % a and b both point to same array [1 2 3]

5.4 Polymorphism

In Postscript, a variable can refer to objects of different types. There is no explicit data type declaration like in 'C'. Assigning a value of a different type, changes the type of the variable.

/a 23 def (a declared a integer with value 23)

/a (PPL) def (a now a string with value PPL)

5.5 Procedures

In Postscript, procedure is a data type and is called packed executable array. Packed Arrays use less space but cannot perform random access. Executable arrays are the ones whose contents are executable. Since procedures have both these characteristics, they are called packed executable arrays. Postscript also supports recursion.

5.5.1 Parameter Passing

Parameter Passing in Postscript is done via stack. All objects are passed by value. In fact, there is no separate ‘pass by value’ and ‘pass by reference’ mechanism. However, since complex objects store the pointer in the dictionary instead of value, they are implicitly passed by reference.

6 Memory Management

6.1 Storage Allocation

All variables of simple types are stored in the dictionaries on the dictionary stack. This is analogous to the static storage area. All variables of complex types and literals are stored in a memory area which is called virtual memory. This is analogous to dynamic storage area or heap.

6.2 Virtual Memory

The region where data of complex objects like arrays, dictionaries (not the ones on the dictionary stack) and literals is stored is called virtual memory. There are two such regions called local and global virtual memory where local and global data is stored.

6.3 Garbage Collection

Whenever the Virtual memory reaches a limit, Garbage collector is run. Objects which are no longer in scope or no longer used, are cleaned and the memory is released. This includes the memory allocated to literals. Garbage collector can be explicitly run through the operator **vmreclaim.** The other operator which cleans virtual memory is **restore**. It locates all the memory that has been allocated since the last **save** (operator) and cleans it. Memory allocated to variables can also be de-allocated explicitly using operator **undef**.

7 Name, Scope and Binding

7.1 Binding

In postscript the binding of a name to a value is done when a variable is defined. If the value is changed later, it is updated in the dictionary for the appropriate variable. However Postscript provides a feature for early binding with the operator **bind**.

With early binding, the name of an object in a procedure is replaced with the current value of that object at the time of the definition of the procedure. Therefore subsequent calls to the procedure would use that value, even though the value of object is changed later during execution.

7.2 Scope

Scope of an object is determined by dictionaries on the dictionary stack. Global variables are visible anywhere in the program. Local variables inside a method are visible globally but not to other methods. Local variables can be scoped to be visible only inside a method by using a dictionary.

```
/a (PPL) def
/methodb
{
1 dict                                %(declaration of a dictionary to contain one variable)
begin
          /d (PPL method) def
          a show                      % (prints PPL)
end
}def
d show    %(results in error because d is scoped by a dictionary in methodb)
```

8 Graphics

8.1 Introduction

Graphic is one the most important aspects of this language. PostScript provides wide range of graphics operator. But there are few graphics concepts that this language uses which make it a powerful graphics language. The following are few graphics concepts used by PostScript to describe and manipulate images on a page.

8.1.1 User Space

It’s the coordinate system used by PostScript. User space is same as the first coordinate of the standard coordinate system. The origin of the coordinate system will be the lower left corner of the page. Coordinates can be specified by integer or real numbers.

8.1.2 Device Space

It’s the coordinate system of the device. Every device has its own way of addressing pixels in their image area. The user must not be bothered about device space. PostScript transforms the user space into device space. This makes PostScript device independent page description language.

8.1.3 Transformation Matrix

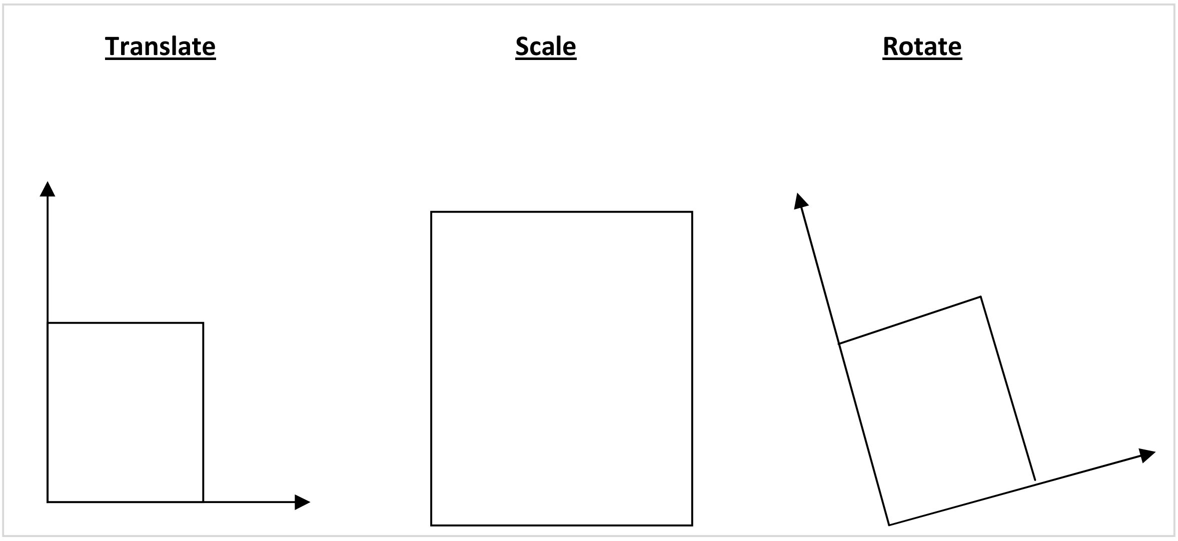
Transformation matrix specifies a way to transform coordinates of one coordinate space to another. The user space coordinates are transformed into device space using transformation matrix called Current transformation matrix (CTM).

8.1.4 Coordinate Transformation Operators

PostScript can modify the user space to be more suitable to its needs by using coordinate transformation operators like translate, rotate and scale.

- **translate** moves the user space origin to new position with respect to the current page. Orientation of the axes and unit lengths are unchanged.

- **rotate** turns the user space axes by some angle with respect to the current axis. Here also the orientation of the axes and the unit lengths are unchanged.
- **scale** leaves origin and orientation of axes unchanged but can change the unit lengths independently along x and y axes.



8.1.5 Graphics State

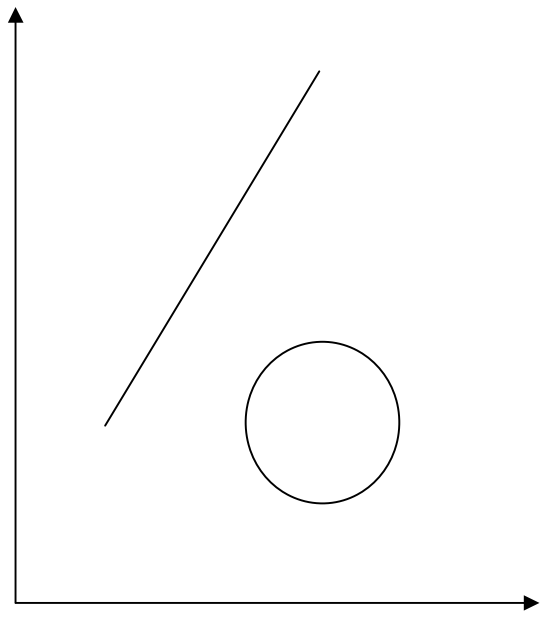
This describes the current state of a graphics system which includes current path, current font and current transformation matrix etc. While drawing on the image area, you often have to temporarily save the graphics state to be used later. PostScript maintains a special graphics state stack which is used to save the state. This can be done using gsave and grestore operators.

8.1.6 Clipping Path

PostScript allows you to set a clipping path that limits the region of page affected by the painting operators.

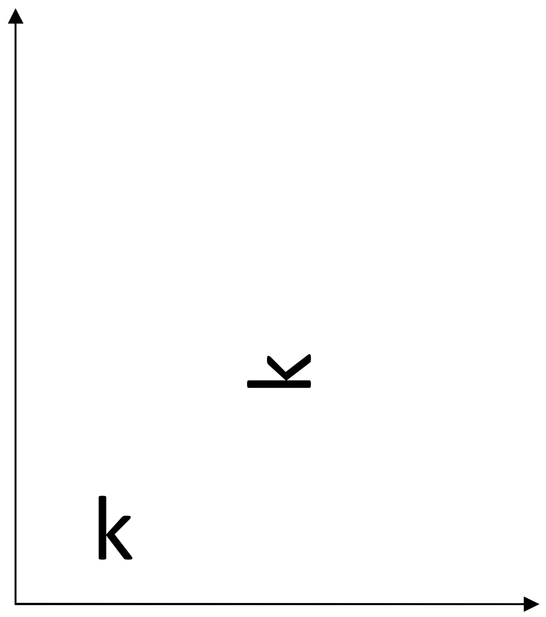
8.2 Graphics Programming

Graphics programming is the most important aspect of PostScript. Let us see some simple examples to see how it works.



Drawing a line and an arc

```
% move to (100,100)

100 100 moveto

% draw a line to (450,600)

 450 600 lineto

% draw a circle at (300,300) as
center and % 100 as radius

300 300 100 0 360 arc
stroke
```

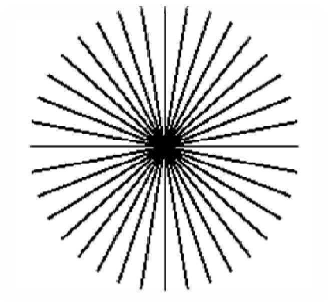


Writing an alphabet and rotating it

```
% move to (100,100)

100 100 moveto

% show K

show(k)

% translate to (200,300) and rotate it
by 90%

200 300 translate
90 rotate
stroke
```

Example of translation and rotation

```
0 10 360 {          % Go from 0 to 360 degrees in 10 degree steps
newpath             % Start a new path

 gsave              % Keep rotations temporary

 144 144 moveto

 rotate             % Rotate by degrees on stack from 'for'

 72 0 rlineto

 stroke

 grestore           % Get back the unrotated state

} for               % Iterate over angles
```



9 Error

Errors can occur during the execution of the postscript program. Some errors like interpreter stack overflow are detected by postscript interpreter. Other errors like no operands or wrong type of operands occur during execution of built-in operators. Error handling is done in a uniform fashion in Postscript. Each error is associated with a name which is a key in a 'error dictionary (errdict)'. The value is the error handler.

Error handling is mainly in two phases 1. Error Initiation and 2.Error Handling.

9.1 Error Initiation

In this phase, the interpreter takes the following steps.

1. Restore the state of operand stack to what it was before the execution of the current object.
2. The object is pushed on the operand stack.
3. The appropriate error handler is executed by looking up into the errdict.

The error handler then takes over. Custom error handlers can be handled for existent error names by associating the new handler with an error. Errors or external events like interrupt or timeout are treated specially. The steps 1 and 2 mentioned above are skipped for these. The handlers of these errors are executed sandwiched between executions of two objects.

9.2 Error Handling

Errdict provides standard handlers which can be overwritten. However this dictionary resides in local VM and so care needs to taken while using save and restore operators when the current state is being changed.

Default error handling procedures operate in the following way.

- Information about error is recorded in a special dictionary called $error.
- VM allocation mode is set to local.
- The stop operator is invoked.

The innermost context established by the stopped operator is exited on the invocation of stop. Interpretation continues on the job server assuming that the user program has not invoked the stopped operator.

A name **handleerror** is executed as a part of the standard error handling procedure. This access the information about the error in the dictionary $error and reports the error depending on the installation. This standard name **handleerror** can be given a new custom value (procedure). For example in some environment this can be changed to print the error on the screen.

10 Summary

Postscript is a language which is a combination of both simplicity and power. It was the first widely used printer control language which facilitated printing both text and images together. It has more than 300 built-in operators to support various operations especially related to graphics. On top of it, Postscript is flexible to add new operators and fonts via dictionaries. PostScript printers transfer the workload involved in printing and rendering images from CPU to printer. However, Postscript was eventually replaced by PDF which is also a format supported by Adobe. This was partly because of the growing competition from non-postscript printers which were available at a lower cost. A major component in the cost of Postscript printers was the postscript interpreter. Also, PDF contains information not only about how the document looks but also about its behavior and content.

11 References

1. http://en.wikipedia.org/wiki/PostScript

2. Thinking in PostScript(First Print), Glenn C.Reid, Addison-Wesley Publications, 1990

3. Postscript Language Tutorial and Cookbook(Second Print), Adobe Systems, Addison-Wesley Publications, 1985.

4. Postscript Language Reference (Third Edition), Adobe Systems, Addison-Wesley Publications, 1999.