Seminar Report

Adobe PostScript

Stefan Widerin (0017313)

16. Februar 2011

Supervisor: Dr. René Thiemann

Abstract

This report provides some insight into the programming language PostScript. Basic concepts and ideas are reflected. PostScript functionalities and the purpose it has been developed for are also described. The focus is on PostScript as a programming language rather than explaining the abilities of drawing graphics and text. Most features of PostScript are illustrated by code-examples to improve their understanding.

Inhaltsverzeichnis

1 Was ist PostScript

PostScript ist eine Turing-vollständige, geräteunabhängige, interpretierte Programmiersprache welche den Text, geometrische Formen und Grafiken eines Dokuments beschreibt. Ein PostScript Programm wird üblicherweise an einen Drucker übermittelt um die Inhalte des Dokuments zu drucken, es ist aber auch möglich, den Inhalt von Ausgabegeräten wie einem Monitor zu kontrollieren.

1.1 Geschichte von PostScript

In den späten 70er Jahren arbeiteten John Warnock und Chuck Geschke für das Unternehmen Xerox PARC an einer Sprache um Inhalte von Dokumenten zu beschreiben, sodass diese Dokumentbeschreibung an einen Drucker übermittelt und von diesem ausgedruckt werden konnte. Das Ergebnis dieser Zusammenarbeit war die Beschreibungssprache InterPress. Geschke und Warnock konnten Xerox PARC jedoch nicht davon überzeugen, dass InterPress kommerziell erfolgreich sein könnte; sie verließen das Unternehmen und gründeten 1982 Adobe Systems um eine an InterPress angelehnte Sprache namens PostScript zu entwickeln und zu vertreiben.
Im Jahr 1985 erschien die erste Version von PostScript. Im selben Jahr kam der erste PostScript-kompatible Laserdrucker auf den Markt, der Apple LaserWriter. Somit war der Grundstein des Erfolgs von PostScript gelegt und bereitete den Weg des Desktop Publishing (Gestaltung von Druckunterlagen am Computer unter Verwendung von Layout Software).

1.2 Wichtigste Funktionen

Mittels PostScript können beliebige Formen wie Vektorgrafiken, aber auch Glyphen und Rastergrafiken beschrieben werden. Diese werden durch Geraden, Bögen, Rechtecke und kubische Kurven erstellt. Diese Formen können beliebig eingefärbt werden.
Textzeichen werden in PostScript wie graphische Objekte behandelt, wodurch alle Grafik-Operatoren auch auf Textzeichen angewandt werden können.
Digitale Bilder können in jedweder Auflösung und in verschiedenen Farbmodellen repräsentiert werden.
Das Koordinatensystem in PostScript unterstützt jede Kombination von Transformationen wie Translation, Skalierung, Rotation, Reflektion oder Scherung. Eine solche Transformation wirkt sich auf alle Elemente des Dokuments aus, seien es Texte, Formen oder Rasterbilder.

1.3 Seitenbeschreibungssprachen

Es wäre theoretisch möglich jede Seite als Array von Pixeln zu repräsentieren, was allerdings ineffizient wäre, vor allem betreffend des Speicherbedarfs. Eine Seitenbeschreibungssprache soll also möglichst kompakt den Seiteninhalt beschreiben, sodass die Dateigröße klein bleibt, wovon auch die Übertragung zu Ausgabegeräten profitiert. Die Repräsentation der Seite soll unabhängig vom

Endgerät sein.
Für die Beschreibung einer Seite werden abstrakte grafische Elemente verwendet anstatt Pixel für Pixel zu beschreiben, wodurch die Unabhängigkeit vom Endgerät erreicht wird. Die Rasterung und Ausgabe der Seite wird vom Interpreter des jeweiligen Geräts durchgeführt.

Statische und dynamische Seitenbeschreibungssprachen

Seitenbeschreibungssprachen werden in die zwei Klassen *statisch* und *dynamisch* unterteilt.

Statische Sprachen bieten eine fixierte Anzahl an Operationen und eine Syntax, um auszuführende Operationen und deren Argumente festzulegen. Die Verarbeitung eines Dokuments, welches mit einer statischen Sprache beschrieben wurde, entspricht der Verarbeitung eines Datenstroms.

Im Gegensatz dazu verhält sich eine dynamische Seitenbeschreibungssprache wie PostScript wie ein ausführbares Programm. Es gibt Variablen, Prozeduren und Konstrukte zur Kontrollflußsteuerung. Eigene Operationen können definiert und verwendet werden.

2 Die Sprache PostScript

Bevor wir uns der Programmierung mit PostScript widmen, werden noch einige Basiskonzepte von PostScript besprochen.

2.1 Grundlagen

PostScript ist eine stackbasierte Sprache und nutzt die postfix Notation. Man spricht auch von umgekehrter polnischer Notation. Die Mittel die PostScript zur Verfügung stellt, wie Schleifen und Rekursion, sowie bedingte Verzweigungen, machen die Sprache Turing-vollständig, was in Zuge dieser Arbeit aber nicht formal bewiesen wird. Die Dateiendung ist .ps und der Code muss mit den zwei Zeichen *%!* eingeleitet werden, damit der nachfolgende Code als PostScript-Code interpretiert wird.
Die Addition der Zahlen 3 und 4 wird wie folgt notiert:

```
3 4 add
```

Bei der Interpretation würde die Zahl 3 auf den Stack gelegt werden, dann die Zahl 4. Beim Verarbeiten der *add* Anweisung werden die zwei obersten Stackeinträge vom Stack genommen, addiert und das Ergebnis auf den Stack gelegt.

2.2 Stack

Insgesamt werden von PS (PS für PostScript) fünf verschiedene Stacks geführt: Der *Operanden Stack*, *Dictionary Stack*, *Execution Stack*, *Graphics State Stack* und *Clipping Path Stack*.
Die gezeigte Addition benutzt nur den Operanden Stack. Wie der Name schon andeutet, stehen auf diesem Stack die Operanden einer Operation. Das Resultat wird wieder auf diesen Stapel gelegt.

2.3 PostScript Interpreter

Wie schon eingangs erwähnt ist PostScript eine Sprache die interpretiert wird. Der Interpreter führt sogenannte PostScript-Objects aus. Ein solches Objekt kann ein Operator, eine Prozedur, eine Zahl, ein String, ein boolscher Wert oder ein Array sein. Beim Scannen des Eingabestroms wird eine Sequenz von PS-Objekten erzeugt. Sobald ein Objekt gescannt wurde, wird dieses augenblicklich ausgeführt.
Anders als bei der Programmiersprache C wird also nicht ein gesamtes Programm gelesen und kompiliert. Vielmehr wird eine Sequenz von Anweisungen Stück für Stück konsumiert.

2.4 Datentypen in PostScript

In PostScript unterscheidet man zwischen zwei Arten von Datentypen: einfache Datentypen und zusammengesetzte Datentypen.
Zu den einfachen Datentypen gehören: *integer, real, boolean, name, mark, null, operator, fontID*

Die Datentypen *integer*, *real* und *boolean* Datentypen sind selbsterklärend. Der Typ *name* ist der Datentyp von Bezeichnern, z.B. von Variablennamen oder Prozedurnamen.
Ein *mark* Objekt kennzeichnet eine Position am Operandenstack. Die eckigen Klammer die ein Array deklarieren sind beispielsweise solche mark-Objekte.
Nicht initialisierte, zusammengesetzte Datentypen enthalten *null* Objekte. Der Datentyp von PostScript Befehlen wie *add* oder *sub* ist *operator*. Objekte vom Typ *fontID* werden benötigt um Schriftarten zu spezifizieren und zu verwenden.

Die zusammengesetzten Datentypen sind: *string, array, packedarray, dictionary, file, gstate, save*

Die einfachen Datentypen zeichnet aus, dass der Wert eines Objekts, der Datentyp und zugehörige Attribute aneinander gebunden sind, zu einem Gesamtpaket. Wird ein solches Objekt kopiert, werden sowohl Wert, Datentyp und Attribute kopiert.
Die Attribute eines Objekts setzen sich zusammen aus der Art wie darauf zugegriffen werden kann (ohne Einschränkung, nur lesend, nur ausführend, gar nicht) und ob es sich bei dem Objekt um ein *Literal* oder ein *ausführbares Objekt* handelt. Literale sind Datenobjekte die beim Verarbeiten auf den Stack gelegt werden, ausführbare Objekte hingegen werden ausgeführt.
Wird jedoch ein zusammengesetztes Objekt kopiert, werden die Werte nicht kopiert, die Kopie des Objekts zeigt auf die ursprünglichen Werte. Dies ist vergleichbar mit dem Kopieren von Referenztypen in Java, bzw. mit dem Kopieren von Pointern in C.

Im Gegensatz zu einfachen Datentypen, bei denen der Wert an derselben Stelle liegt wie das Objekt (z.B. auf dem Stack), liegen die Werte von zusammengesetzten (oder auch composite) Datentypen in einem speziellen Speicherbereich, nämlich dem *Virtual Memory*.

Der Datentyp *string* wird wie in C als Array von Zeichen behandelt. Ein String muss in PostScript von runden Klammern eingeschlossen werden. Ein Array ist eine Sammlung von Objekten die über einen Index angesprochen werden können. Anders als bei Sprachen wie C oder Java können in einem PS-Array Daten verschiedenen Typs zusammengefasst werden. Beispiel:

```
[ 1 (hallo) 3.14159 ]
```

Dieser Code würde ein Array mit dem Integerwert 1, dem String hallo, und dem real-Wert 3.14159 auf den Stack legen.

Ein *packedarray* ist eine kompaktere Form eines Arrays. Innerhalb von geschwungenen Klammern wird eine Folge von Anweisungen angeführt, vergleichbar mit einem Anweisungsblock in der Sprache C. In PostScript spricht man auch von ausführbaren Arrays. packedarrays benötigen weniger Speicher als gewöhnliche Arrays.

Für die weiteren composite-Typen wird auf [1] verwiesen.

2.5 Konstrukte einer Programmiersprache

Variablen

Eine Variable wird in PostScript wie folgt deklariert:

```
/x 10 def
```

Beim Verarbeiten dieser Anweisung wird x als Literal auf den Operanden Stack gelegt, mit dem Wert 10 wird genauso verfahren. Die *def* Anweisung nimmt diese zwei Argumente wieder vom Stack und legt das neue Key-Value Paar (Key: x, Value: 10) auf den Dictionary Stack. Der Schrägstrich vor dem x ist notwendig, damit der Interpreter den nachfolgenden Identifier auf den Stack legt, anstatt diesen im Dictionary zu suchen.
Der Typ einer Variable wird abgeleitet und braucht nicht angegeben zu werden. Eine Variable kann dann wie folgt verwendet werden:

```
11 x sub
```

Der Interpreter würde 11 auf den Operanden Stack legen, dann aus dem Dictionary Stack den Wert für den Bezeichner x laden und diesen auf den Operanden Stack legen. *sub* würde die Werte 11 und 10 vom Stapel nehmen, subtrahieren und das Ergebnis, also 1, auf den Stapel legen.
Die Variable x zu inkrementieren ist etwas aufwändiger zu notieren als in einer Sprache wie C:

```
/x x 1 add def
```

2.6 Konditionale Statements

Vergleichsoperatoren

Für gewöhnlich werden in konditionalen Statements Vergleichsoperatoren verwendet. In PS stehen folgende Operatoren für Vergleiche zur Verfügung: *eq* um auf Gleichheit zu prüfen, *ne* um Ungleichheit zu prüfen; *gt*, *ge*, *lt*, *le* prüfen ob das erste Argument *größer, größer-gleich, kleiner, kleiner-gleich* als das zweite Argument ist.
Der Rückgabewert dieser Operatoren ist erwartungsgemäß ein boolscher Wert der auf den Stack gelegt wird.

if-Statement

Das *if*-Statement nimmt einen boolschen Wert und ein ausführbares Array (Sprich eine Sequenz von Anweisungen) vom Stack und führt die Anweisungen aus, wenn der boolsche Wert true ist. Beispiel:

```
x 0 gt { x sqrt } if
```

Hier wird überprüft ob die Variable x größer als 0 ist. Ist das der Fall, wird die Wurzel der Variable x berechnet.

ifelse-Statement

Das *ifelse*-Statement erweitert das *if*-Statement um einen weiteren Anweisungsblock welcher ausgeführt wird, wenn der boolsche Wert auf dem Stack false ist. Beispielsweise könnte eine Betragsfunktion folgendermaßen realisiert werden:

```
x 0 ge { x } { x -1 mul }  ifelse
```

Falls der Wert x größer oder gleich 0 ist, wird er auf den Stack gelegt. Ist x kleiner als 0 wird der zweite Anweisungsblock ausgeführt und x wird mit -1 multipliziert und auf den Stack gelegt.

Es existiert kein *switch*-Statement in PostScript.

2.7 Schleifen

repeat-Schleife

Eine sehr einfach zu verwendende Form eines Loops ist der *repeat*-Loop. Die repeat Anweisung nimmt einen Integer Wert und ein ausführbares Array vom Stack. Die Anweisungssequenz wird dem Integer Wert entsprechend oft ausgeführt. Beispiel:

```
/x 4 def
/count 0 def
x { /count count 1 add def } repeat
```

Dieser Code würde die Variable count viermal um den Wert 1 inkrementieren. Nachdem der repeat-Loop terminiert hat, ist der Wert der count Variable 4.

for-Schleife

Der *for*-Loop nimmt drei Integer Werte als Input und führt die angegebene Prozedur bzw. Anweisungsfolge aus. Der erste Integerwert ist der Startwert der implizit vorhandenen Zählvariable, der zweite Wert wird nach jedem Durchlauf zu dieser addiert und der dritte Wert bildet das Limit welches den Schleifenabbruch auslöst.
Der aktuelle Wert der Zählvariable wird vor dem Ausführen der Anweisungen auf den Stack gelegt. Beispiel:

```
0 1 10 { == } for
```

Diese Codezeile funktioniert nur im sogenannten interaktiven Modus, also wenn sie zum Beispiel in den GhostScript Interpreter eingetippt wird. Der Stack-Operator == nimmt den obersten Wert vom Stack und gibt in auf der Konsole aus.
Die Ausgabe lautet: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Man erkennt daran, dass der Wert 10 also noch von der Zählvariable angenommen werden darf, erst danach bricht die Schleife ab.

loop-Schleife

Eine *loop*-Schleife wird solange ausführt, bis ein *exit* Befehl die Schleife abbricht.

```
/x 0 def
{
  x
  /x x 1 add def
  x 10 eq { exit } if
} loop
```

In der Schleife werden die Werte 0 bis 9 auf den Stack gelegt. Sobald x den Wert 10 annimmt, wird die Schleife mittels *exit* abgebrochen.

forall-Operator

Die *forall*-Anweisung wendet die Anweisungen innerhalb der geschwungenen Klammern auf jedes Element in einem Array an.

```
0 [ 1 2 3 4 ] { add } forall
```

Bevor über das Array iteriert wird, wird die Zahl 0 auf den Stack gelegt. Dann wird das erste Element des Arrays auf den Stack gelegt und *add* ausgeführt, was die Summe von 0+1 auf den Stack legt. Für das zweite Element wird nun 1 und 2 addiert usw.
Nachdem alle Elemente im Array verarbeitet wurden, liegt der Wert 10, also die Summe der Elemente im Array, auf dem Stack.

2.8 Prozeduren

Prozeduren sind nicht vergleichbar mit einer Funktion wie in C. Prozeduren sind eine Serie von Anweisungen, die gewöhnlich auf dem Stack operieren. Man kann sich vorstellen, dass diese Instruktionen an die Stelle des Prozeduraufrufs kopiert werden. Die Anzahl der Parameter die eine Prozedur erwartet, hängt von den Befehlen in der Prozedur ab. Besteht die Prozedur beispielsweise aus zwei *add* Operationen, müssen drei Werte auf dem Stack liegen wenn die Prozedur aufgerufen wird. Eine Prozedur kann durchaus auch mehrere Ergebnisse auf den Operandenstack legen.

```
/avg {add 2 div} def
20 40 avg
```

In der ersten Codezeile wird eine Prozedur mit Namen *avg* definiert. Innerhalb der geschwungenen Klammern werden die auszuführenden Anweisungen notiert. Diese Prozedur führt einen add Befehl aus, das Resultat der Addition wird anschließend durch 2 dividiert. Die add Anweisung nimmt die obersten zwei Einträge vom Operanden Stack. Diese werden in der zweiten Codezeile auf den Stack gelegt und dann von der avg Prozedur verarbeitet.
Eine Prozedur, welche den Durchschnitt zweier Zahlen berechnet und anschließend 10% zum Durchschnittswert addiert, ist aufgrund der stackbasierten Ausführung etwas komplizierter zu schreiben als z.B. in C.

```
/avgPlusTenPercent {add 2 div dup dup 10 div add} def
20 40 avgPlusTenPercent
```

Da 10% vom Durchschnitt berechnet und zum ursprünglichen Wert addiert werden (anstatt mit 1.1 zu multiplizieren), ist es notwendig, den errechneten Durchschnittswert am Stack zu duplizieren, um davon die zehn Prozent zu errechnen und zu addieren. Mittels dem *dup* Befehl wird das oberste Element auf dem Stack dupliziert, sodass er zweimal für weitere Berechnungen verwendet werden kann. Durch den zweifachen Aufruf von dup befinden sich nach dem Abarbeiten der Prozedur sowohl der Durchschnittswert 30, als auch der um 10% erhöhte Wert, also 33 am Stack.

2.9 Rekursive Prozeduren

Auch rekursive Prozeduren sind in PostScript möglich. Folgendes Beispiel aus [2] berechnet rekursiv die Fakultät des Prozedurarguments:

```
/factorial
{
  dup 1 gt
  {
    dup 1 sub factorial mul
  } if
} def
4 factorial % Fakultaet von 4 berechnen
```

Die Zeile

```
dup 1 gt
```

dupliziert das oberste Stackelement, dann wird 1 auf den Stack gelegt. *gt* nimmt die obersten Werte vom Stack und legt true auf den Stack wenn der Stackwert unterhalb von 1 größer als 1 ist, ansonsten false. Das *if*-Statement prüft diesen boolschen Wert, nimmt ihn vom Stack und führt folgenden Code aus, wenn er true ist:

```
dup 1 sub factorial mul
```

Der um 1 verminderte Wert des obersten Stackelementes wird auf den Stack gelegt. Der darauffolgende Prozeduraufruf *factorial* wiederholt diesen Vorgang so lange, bis der oberste Stackwert 1 ist. Schließlich sind noch die *mul* Operationen abzuarbeiten. Wenn das Argument der Fakultät *n* war, sind *n-1* Multiplikationen auszuführen.

3 Grafikprogrammierung mit PostScript

Nachdem wir bisher PostScript rein als Programmiersprache betrachtet haben, widmen wir uns nun der Grafikprogrammierung mit PS.

3.1 Koordinatensystem

Das Koordinatensystem in PS hat seinen Ursprung in der linken, unteren Ecke der Seite. Die x,y Werte wachsen nach rechts oben. Die Rasterung ist standardmäßig 1/72 Zoll; dies ist somit die Einheit des Koordinatensystems und wird als 1 Punkt bezeichnet. Ein wichtiges Objekt, genauer Operator, ist *currentpoint*. Der Aufruf des Operators currentpoint legt zuerst die x-Koordinate, dann die y-Koordinate auf den Stack. Beim Schreiben bzw. Zeichnen ist es wichtig zu wissen, wo sich der currentpoint gerade befindet.
Auf das Koordinatensystem können Transformationen angewandt werden: *translate* verschiebt den Ursprung des Koordinatensystems, *rotate* rotiert das Koordinatensystem, *scale* skaliert das Koordinatensystem. Weitere Transformationen des Koordinatensystems sind in [1] zu finden.

3.2 Dokumentstruktur

Ein Programm in PostScript wird in zwei Bereiche unterteilt: den *Prolog* und den *Script* Teil.
Im Prolog werden Prozeduren definiert die bei der Ausführung des Skripts verwendet werden können.
Im Script-Teil finden sich PostScript Operatoren und Prozeduraufrufe, deren Argumente, sowie Daten, z.B. Text. Im Gegensatz zum Prolog ist der Inhalt des Script-Abschnittes eher einfach und er wird gekennzeichnet durch viele wiederholende Aufrufe. In diesem Abschnitt werden die Seiten eines Dokuments beschrieben.

Es ist für Programme die PostScript-Dateien verarbeiten nützlich, wenn die Struktur eines Dokuments aus dem PS-Programm hervorgeht, beispielsweise wenn das Ende einer Seite gekennzeichnet ist. Andernfalls wäre es unmöglich für eine Anwendung bestimmte Seiten eines Dokuments zu drucken oder Seiten in einem Dokument zu verschieben.
Damit einzelne Seiten verarbeitet werden können, ist es notwendig, dass eine Seite nicht von Definitionen einer anderen Seite abhängig ist. Aufgrund dessen darf eine einzelne Seite nur auf Definitionen im Prolog zurückgreifen und niemals etwas verwenden was in anderen vorhergehenden Seiten benutzt wurde.
Diese *Unabhängigkeit zwischen Seiten* kann folgendermaßen erreicht werden:
Meist ist es gewünscht für ein Dokument ein generelles Erscheinungsbild festzulegen. Soll eine einzelne Seite ein anderes Erscheinungsbild erhalten, so wird dieses am Seitenbeginn definiert und am Ende der Seite wieder aufgehoben, sprich die ursprüngliche Einstellung wieder hergestellt.

3.3 Zeichnen auf einer Seite

Nachfolgend werden einige grundlegende Funktionen zum Zeichnen von Grafiken besprochen.

3.4 Ein Quadrat zeichnen

Folgendes Beispiel aus dem PostScript Tutorial and Cookbook [2] zeichnet die Außenlinien eines Quadrates.

```
newpath
270 360 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
closepath
4 setlinewidth
stroke
showpage
```

newpath leert den aktuellen Pfad (sofern vorhanden) und erstellt einen neuen Zeichenpfad. Ein alter Pfad und der neu erstellte sind also getrennt. *moveto* setzt den aktuellen Punkt im Koordinatensystem auf (x:270, y:360). Die Anweisung *0 72 rlineto* erstellt eine neue Linie, relativ zum aktuellen Punkt. Die Argumente 0 und 72 bedeuten also, dass kein Versatz in x-Richtung stattfindet und dass man sich um 72 Einheiten (ein Zoll) nach oben (in positiver y-Richtung) bewegt. Das *r* in *rlineto* weist auf darauf hin, dass die angegeben Koordinaten relativ zum aktuellen Punkt sind. Die zwei weiteren *rlineto* Anweisungen zeichnen jeweils noch eine weitere Linie des Quadrats. Mit *closepath* wird vom letzten Punkt zum Ursprung (genauer: zum letzten Punkt eines moveto Befehls) des aktuellen Pfades eine Linie und schließt den Pfad. Dadurch werden unschöne Artefakte zusammenstoßender Linien mit verschiedenen Winkeln vermieden. *setlinewidth* setzt die Linienstärke auf 4/72 Zoll; *stroke* zeichnet die Pfadlinien und löscht den Pfad, *showpage* gibt die Seite aus.

3.5 Ein Quadrat mit Außenlinien zeichnen

Da die *stroke* Anweisung das Pfadobjekt löscht, bedarf es der Einführung weiterer Operatoren um den Pfad für weitere Zwecke zu sichern:

```
newpath
270 360 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
closepath
gsave             % neu
4 setlinewidth
stroke
grestore          % neu
1 0 0 setrgbcolor
fill
showpage
```

gsave legt eine Kopie des aktuellen Grafikstatus auf den *graphics state stack*. Nach diesem Aufruf wird wie im Beispiel zuvor die Außenlinie des Quadrats gezeichnet, was zugleich den zugrunde liegenden Pfad löscht. Der zuvor gesicherte Pfad wird jedoch mit *grestore* wieder vom Stack geholt und mit den Befehl *fill* gefüllt. Zuvor wird noch die RGB-Füllfarbe mittels *setrgbcolor* festgelegt; in diesem Fall mit 100% Rot.

3.6 „Hallo Welt“ in PostScript

Für ein „Hallo Welt“ PS Programm benötigen wir einige neue Operatoren.

```
/Monospace findfont
20 scalefont
setfont
200 200 moveto
(Hallo Welt!) show
showpage
```

Der Operator *findfont* lädt die Informationen der Schriftart Monospace, *scalefont* skaliert die Zeichen auf eine Größe von 20 Punkt. *setfont* setzt den geladenen Zeichensatz zum aktuell verwendeten. Die Anweisung *moveto* bewegt den aktuellen Punkt zur Position 200, 200 im Koordinatensystem. Beginnend an diesem Punkt wird der String „Hallo Welt“ mittel *show* in die Seite geschrieben.

3.7 99 bottles of beer

Nun haben wir uns genug Wissen angeeignet um uns den „99 bottles of beer“ zu widmen. Der folgende Code ist eine leichte Abwandlung des Codes von Volker Beyler, verfügbar unter
`http://99-bottles-of-beer.net/language-postscript-1135.html`

```
/Helvetica findfont
6 scalefont
setfont

/tostring { 3 string cvs } def
/bottles 99 def

99 -1 0
{
  72 bottles 7.5 mul 20 add moveto

  bottles 0 gt
  {
    bottles tostring show
    bottles 1 gt
      {( bottles of beer on the wall, ) show}
      {( bottle of beer on the wall, ) show}
    ifelse
```

```
    bottles tostring show
    bottles 1 gt
      {( bottles of beer. ) show}
      {( bottle of beer. ) show}
    ifelse
    (Take one down and pass it around, ) show
    /bottles bottles 1 sub def
  }
  {
    (No more bottles of beer on the wall, ) show
    (no more bottles of beer. ) show
    (Go to the store and buy some more, ) show
    /bottles bottles 99 add def
  }
  ifelse




  bottles 0 gt
  {
    bottles tostring show
    bottles 1 gt
      {( bottles of beer on the wall.) show}
      {( bottle of beer on the wall.) show}
    ifelse
  }
  {
    (no more bottles of beer on the wall.) show
  }
  ifelse
}
for
showpage
```

Die ersten drei Zeilen des Codes legen die Schriftart und Schriftgröße fest. Die Prozedur

```
/tostring { 3 string cvs } def
```

führt die Konvertierung in einen String der Länge 3 durch. Wesentlicher Bestandteil des Programms ist jedoch die for-Schleife, welche 100-mal durchlaufen wird. Dass von 99 bis 0 heruntergezählt wird, tut für die Erstellung der Textzeilen des Liedes nichts zur Sache; für diesen Zweck wurde die Variable *bottles* definiert. Die Zeile

```
72 bottles 7.5 mul 20 add moveto
```

setzt den currentpoint nach jedem Schleifendurchlauf eine Zeile (genauer: 7.5 Punkt) nach unten.

Danach wird, je nachdem ob noch Bierflaschen vorhanden sind oder nicht, der erste Teil der Textzeile ausgegeben, z.B.:
99 bottles of beer on the wall, 99 bottles of beer. Take one down and pass it around,
Anschließend wird der Wert der Variable bottles um 1 verringert:

```
/bottles bottles 1 sub def
```

Schließlich wird noch der Rest der Textzeile ausgegeben:
98 bottles of beer on the wall.

Neu in diesem Programm ist lediglich die Konvertierung eines Integerwertes zu einem String, was für die Ausgabe notwendig ist. Diese Konvertierung erfolgt in der selbst definierten Prozedur *tostring*. Der Befehl *3 string* erzeugt einen mit Nullen initialisierten String der Länge 3 am Stack. *cvs* konvertiert dann ein beliebiges Objekt in einen String der Länge 3.

4 Zukunft von PostScript

Mit Erscheinen von PostScript wurde damals der Weg für das Desktoppublishing geebnet. Lange Zeit wurde PostScript massiv und erfolgreich im Printbereich eingesetzt. PostScript-fähige Drucker mussten aber mit teuren Raster-Image-Prozessoren ausgestattet werden, was den Wunsch nach kostengünstigeren Lösungen weckte.
Deshalb wurden viele günstige Drucker mit PostScript-kompatiblen Interpretern von Drittanbietern ausgestattet, bzw. wurde ganz auf die Unterstützung von PostScript verzichtet. Drucker ohne PostScript-Support rastern das native Grafikformat der jeweiligen Plattform mit Hilfe geeigneter Treiber. Muss dennoch ein PostScript-Dokument auf einem solchen Gerät gedruckt werden, kann auf einen Software-Interpreter wie *GhostScript* zurückgegriffen werden, welcher das Dokument rastert und als Bitmap an den Drucker sendet.
Zudem hat das Portable Document Format, kurz PDF, PostScript als Seitenbeschreibungsformat abgelöst. PDF ist der Nachfolger von PostScript, ebenfalls eine Entwicklung von Adobe Systems. Es verwendet auch noch einen Teil der PS-Funktionalitäten, allerdings wurden die Elemente zur Steuerung des Kontrollflusses nicht übernommen.
Auf lange Sicht dürfte PostScript wohl gänzlich von PDF verdrängt werden.

Literatur

[1] Steve Chernicoff, Ed Taft, and Caroline Rose. *PostScript Language Reference, Third Edition.* Addison-Wesley, 1999.

[2] Linda Gas and John Deubert. *PostScript Language Tutorial and Cookbook.* Addison-Wesley, 1985.