Journal for Geometry and Graphics
Volume 12 (2008), No. 2, 215–228.

Using PostScript Programming Language in an Undergraduate Computer Graphics Course

Thomas Walter Rauber[1], **Aura Conci**[2]

[1] *Dept. de Informática, Centro Tecnológico, Universidade Federal do Espírito Santo*
Av. Fernando Ferrari s/n, Goiabeiras, 29060-970 Vitória, ES, Brazil
email: thomas@inf.ufes.br

[2] *Instituto de Computação, Universidade Federal Fluminense*
Rua Passo da Pátria 156, 24210-240 Niterói, RJ, Brazil
email: aconci@ic.uff.br

Abstract. We report about the experiences of using the PostScript programming language in an undergraduate computer science and computer engineering course as a complementary tool besides OpenGL to teach basic concepts of computer graphics, especially affine transformations and hierarchical modeling using a transformation matrix stack mechanism. We can conclude that once the somewhat cryptic syntax of this stack-oriented language has been overcome, a natural computer graphics programming interface is available which permits a rapid understanding of essential concepts in graphics which can then easily be extrapolated to a 3-D interface like OpenGL. We would like to emphasize that the use of PostScript is not intended as an alternative to the standard graphics programming languages, but as an enrichment of the students programming skills in a completely distinct programming paradigm.

Key Words: PostScript, undergraduate teaching, computer graphics
MSC 2000: 68U05

1. Introduction

It is not common sense that PostScript® [1, 2, 3] (from now on sometimes abbreviated as 'PS') is a fully featured programming language which can be used to implement single-task non event-oriented applications. Theoretically, one could teach an introductory course to computer programming by it, topics like searching and sorting. Obviously no instructor worldwide does this because other imperative programming languages, like for instance C++ or Java are much more appropriate. PostScript is an interpreter that is controlled by a stack onto which all commands must be placed. So why should someone use this language for

ISSN 1433-8157/$ 2.50 © 2008 Heldermann Verlag

general purpose programming when the syntax is very uncommon and eventually even the abstract data type 'stack' is unknown to the audience?

When it comes to programming where graphical output is needed, the general purpose languages must have an interface to set up the graphical environment and manipulate the graphic objects. It is sometimes quite laborious to create this graphical programming environment. There are different operating systems and graphical capabilities of the specific interfaces.

PostScript was created to provide a device independent programming language especially to produce high quality graphical output, usually on hard copy devices like printers. Its conception does not aim at using it as a general purpose programming language, but rather develop software in a highly specialized community of printer driver programmers. There exist PS emulators that produce the graphical output on the monitor, interpreting the language in batch mode. The intention behind the emulator is to visualize the page before eventually printing it out. The most commonly used emulator is Ghostscript [4]. Since PS documents are ubiquitous on digital computers, an emulator is usually available on the machine. On Linux OS, Ghostscript is generally pre-installed and on Windows the need to visualize PS documents leads also to an installation sooner or later. The PS emulator is also an interactive command interpreter when the state of the stack is considered as the output. We can therefore state that a PostScript programming environment is available on practically every computer and hence can be used to implement graphical applications. A strong motivation for using PostScript in graphics programming for tutorial purposes is the build-in availability of generic computer graphics features like the matrix stack mechanism and affine transformations, just like in the 3-D OpenGL programming interface [9]. The effects of sequences of affine transformation can easily be visualized and the transformation matrix stack mechanism permits to illustrate the modeling of hierarchical objects like a robot for instance. These concepts learned in a 2-D environment can immediately be applied in a 3-D programming interface. The semantics of the complementary Postscript operators `gsave` and `grestore` with respect to the transformation matrix stack is identical to the OpenGL counterparts `glPushMatrix()` and `glPopMatrix()`.

Another reason to use a programming language with a natural graphical interface is the visualization of concepts. An example is recursion. Standard problems like Fibonacci numbers or factorials can be enriched by graphical recursions (e.g. self-similar fractals like Sierpinski triangles or Koch snowflakes). Since the various levels of recursion are usually structured on a stack, the programming in PostScript becomes very compact and elegant.

We propose the use of PS in an introductory computer graphics course as a small part (six hours of a 60 hours course) of the curriculum, parallel to the standard topics, usually implemented in the OpenGL API. The main motivation is to illustrate some of the fundamental concepts of graphics from a different viewpoint which facilitates their understanding (such as in affine transformations). Besides, the student learns to program in a programming language which is uncommon in its conception and academically is as least as interesting as Lisp in the field of Artificial Intelligence.

2. The PostScript programming language

It is outside the scope of this text to give a complete introduction to PostScript. There are several freely available books [1–3, 5, 6, 8] which can be consulted. Besides, the language has become more sophisticated, expressed in various levels of the syntax. In 1997, PostScript 3

was defined with more levels of gray, manipulation of fonts and color processing. In this exposition we concentrate on the essential parts of the language needed in teaching affine transformations and hierarchical modeling.

2.1. The stack

The PostScript interpreter processes the elements that are put on a single stack. Operands are pushed until a keyword is found which spawns a user-transparent processing that modifies the stack, internal data structures or the graphical output. Let us assume that we are in an interactive mode where we communicate with the interpreter. This can be done with Ghostscript typing `gs` in a terminal window shell which passes the control to an interactive PS interpreter displaying the prompt `GS>`. The sequence

```
2 3 add
```

first pushes `2`, then `3` on top of `2` and finally `add` on top of `3`. When the interpreter finds the keyword `add` on top of the stack, it takes the previous two objects from the stack, sums them up and pushes the result `5` back on the stack. It is important for a programmer to be always aware of what is on the stack at a certain point of the program. At any time the stack can be inspected by `pstack`. Small debugging commands can be written that facilitate the visualization of the contents together with explanatory comments.

The rigid sequential nature of the stack abstract data type is a drawback and seems to be a main reason to discourage the use of PS as a general purpose programming language even for computer graphics purposes. Operands that are needed several times in an expression and/or position in a program must be replicated by the `dup` (duplicate the topmost stack element) or *n* `copy` (replicate the last n elements on the stack) operators. Often the order of two subsequent elements must be swapped by `exch`, superfluous elements must be eliminated by `pop` or circular shifts are done by `roll`. As an example consider the calculus of the expression $-c + (a + b)\ln(a - b)$, assuming that the current elements on the stack are `a b c`. For the usual infix notation of a general purpose language like C++ the command would simply be `result = -c +(a+b)*log(a-b)`. In PostScript one possible command sequence is

```
3 1 roll 2 copy sub ln 3 1 roll
    add mul exch sub
```

which leaves the result on the stack. In Table 1 the effect of each of the commands on the stack contents is explicitly illustrated. In this table elements enclosed by curly brackets means that this expression has already been evaluated by the interpreter.

Fortunately there are mechanisms that alleviate the rigidity of the stack element order. Symbolic names can be introduced with the *dictionary* feature of PS. In a hash table like manner values or even complete sequences of commands can be attributed to variables. The `/symbol {...} def` command combination registers `symbol` in the dictionary from where it can be used from now onward at any position on the stack. Usually when an element is on the stack, an `exch def` combination reads it off the stack and attributes it to a variable. The above problem to calculate the expression simplifies to

```
/c exch def
/b exch def
/a exch def
a b add
a b sub
ln
mul
c sub
```

Command	*Stack*
	`a b c`
`3 1 roll`	`c a b`
`2 copy`	`c a b a b`
`sub`	`c a b {a-b}`
`ln`	`c a b {ln(a-b)}`
`3 1 roll`	`c {ln(a-b)} a b`
`add`	`c {ln(a-b)} {a+b}`
`mul`	`c {ln(a-b)(a+b)}`
`exch`	`{ln(a-b)(a+b)} c`
`sub`	`{ln(a-b)(a+b)-c}`

Table 1: Evolution of stack in the calculus of expression $-c + (a + b)\ln(a - b)$.

which is reasonably more readable, especially if the postfix notation is not unknown to the programmer. For Lisp programmers the change from prefix to postfix notation should also be no problem at all. The dictionary mechanism can also be used to define complete parametrizable sub-programs. If we therefore define `/calcexpr{...} def` with the above sequence in lieu of the `'...'`, a call to the sub-program as `3 1 4 calcexpr` would deliver the result `-1.2274`.

2.2. Data representation and control flow

PostScript possesses all important data types and control mechanisms to implement a sequential structured program that can access files, calculate numerical and logical values, select program branches, do unconditional and condition repetitions. In Table 2 a superficial glimpse of some typical constructs is given. At this point we can be aware that complex programs can be implemented with this language, even when the syntax is usually not familiar to the programmer. In Appendix A some graphical and non-graphical simple programs are listed.

2.3. Graphics

Affine geometric transformations are fundamental concepts in computer graphics. PostScript reveals its potential when graphical objects are produced. For drawing polygons and curves the *path* mechanism is used. For our purpose of familiarizing with affine transformation and hierarchical modeling, simple geometric objects (rectangles, circles) are sufficient to illustrate the effects of the basic 2-D computer graphics mechanisms of the language. The sub-program `unitbox` of Table 3 draws a square of unit side length (comments in PS begin with the symbol '`%`'). From now on it can be used simply by invoking `unitbox` in the PS program, for instance for illustrating the non-commutativity of geometric transformations.

2.3.1. Current transformation matrix

One of the main motivations to use PostScript for an introductory familiarization with graphical programming is the availability of a built-in transformation matrix and its stack. When this mechanism has been learned it becomes easy to extrapolate it to 3-D programming. In OpenGL for instance we find the same concept in form of the *modelview matrix*. PostScript

Construct	*Effect*
`/i 0 def`	initialize variable `i` with value `0`
`/i i 1 add def`	increment variable `i` by `1`
`/b b not def`	negate boolean variable `b`
cond {*seq*} **if**	if condition `cond` is true execute command sequence `seq`
cond {*seq1*}{*seq2*} **ifelse**	if condition `cond` is true execute command sequence `seq1` else execute command sequence `seq2`
start inc final {*seq*} **for**	unconditional repetition of command sequence `seq` from initial value `start` to final value `final` by incremental step `inc`
{*seq cond* {exit} if} **loop**	repeat command sequence `seq` while condition `cond` is false
`/ar` *n* **array** `def`	define an array of `n` elements called `ar` and put it into the dictionary
`0 1 ar length 1 sub {ar exch` *n* `put} for`	set each element of array `ar` to value *n*

Table 2: Some typical PostScript constructs for program flow control

```
% Usage:  unitbox
/unitbox {
   0 0 moveto
   1 0 lineto
   1 1 lineto
   0 1 lineto
   closepath stroke
} def
```

Table 3: Sub-program for drawing a unit outline box

stores its current transformation matrix in a six-dimensional array, since any affine transformation matrix in homogeneous coordinates in 2-D $\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$ has six relevant values [7]. The main commands or combinations of commands for manipulating the current transformation matrix (CTM) are resumed in Table 4.

The affine transformation *shearing* with transformation matrix $\begin{pmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ is not provided a priori, but can easily be implemented by the sub-program

```
% Usage:  shx shy shear
/shear {
   /shy exch def
   /shx exch def
   % construct shearing matrix SH(shx,shy)
   /shmat { 1 shy shx 1 0 0 6 array astore } def
   % multiply current transformation matrix by SH
   shmat concat
} def
```

Command	*Effect*
`matrix setmatrix`	set CTM to identity
`6 array currentmatrix`	retrieve CTM and put its relevant values into a six-dimensional array
matrix `setmatrix`	set CTM to *matrix*
matrix `concat`	multiply CTM by *matrix*
trx try `translate`	multiply CTM by translation matrix $\begin{pmatrix} 1 & 0 & tr_x \\ 0 & 1 & tr_y \\ 0 & 0 & 1 \end{pmatrix}$
θ `rotate`	multiply CTM by rotation matrix $\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$
sx sy `scale`	multiply CTM by scaling matrix $\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Table 4: Command sequences for manipulating the current transformation matrix

2.3.2. Current transformation matrix stack

Another fundamental concept in computer graphics is hierarchical modeling where in a tree-like organization the dependent subparts are subjected to all geometric transformations of the superior nodes of the tree. Usually each node of the tree has one degree of freedom with an associated parameter. The visualization of the complete object reads the parameters along the path down the tree and concatenates the respective transformation matrices in order to have different transformations at the tree leaves. The matrix stack mechanism is the ideal tool to implement this hierarchy. In OpenGL we have the `glPushMatrix()` and `glPopMatrix()` commands that control the transformation matrix stack. The concept of the OpenGL *modelview matrix* is the CTM in PostScript. The PS commands `gsave` and `grestore` push and pop the *graphics state* which includes the CTM, besides other attributes that are not considered further here (color state, positions, paths, fonts, line width, among others).

3. Course topics

The necessary knowledge for understanding affine transformations and hierarchical modeling and to implement an application program in PostScript defined the topics of the course, compiled in a sequence of lessons to teach PS programming. Usually three 90 minutes class hours are sufficient to present the contents. In Table 5 the modules of the introductory course to computer graphics programming in PostScript are resumed. The concepts are presented by examples. At the end of each module, exercises have to be solved to solidify the acquired knowledge. Module 1 familiarizes with the basic philosophy of the stack oriented programming, module 2 shows how a polygonal path is constructed and outlined or filled, module 3 presents the most important data types of the language, together with their operators, for instance the boolean type or the relational operators `gt`, `ge`, `lt`, `le` and the most important control constructs of the language.

Module 4 shows how constants and variables can be created and modularization of programming be achieved by the definition of sub-programs. Module 5 is not the focus of the

Module	*Name*	*Contents*
1	Introduction and Stack	Programming paradigm, Stack orientation of PS
2	Introduction to Graphics	Path operators, color models
3	Types, Operators	Integer, boolean, string, operators of each type
4	Dictionaries	Variables, Sub-programs, parameter reading with `exch`
5	Text & Fonts	Font selection, text in graphics
6	Affine Transformations	Affine operators of PS, transformation matrices, concatenation, importance of order
7	Hierarchical modeling	Graphics context, matrix stack, parameter modification
8	Interactivity	Shell based interaction with PS interpreter, interactive application
9	Animation	Simple graphics sequences

Table 5: Topics of the course
"Introduction to Computer Graphics Programming using PostScript"

course, only a small introduction on how to choose fonts or scale symbol sizes. The extremely important topic of affine transformations is outlined in module 6. How can transformations be applied directly (translation, rotation, scaling), or how can the current transformation matrix be queried or manipulated? The composition of basic transforms is shown, for instance the rotation about an axis through a pivot point, or the effect of swapping the order of transformations (e.g. first translation, then rotation or vice versa), besides a non-composed shear, see Fig. 1.

Another important topic is hierarchical modeling of module 7. All non-trivial computer graphics applications must organize the design of dependent object hierarchies. Some of the branches of the modeling tree have degrees of freedom for affine transformations, others have constant values for the transformation. A classical example is a robot which has for

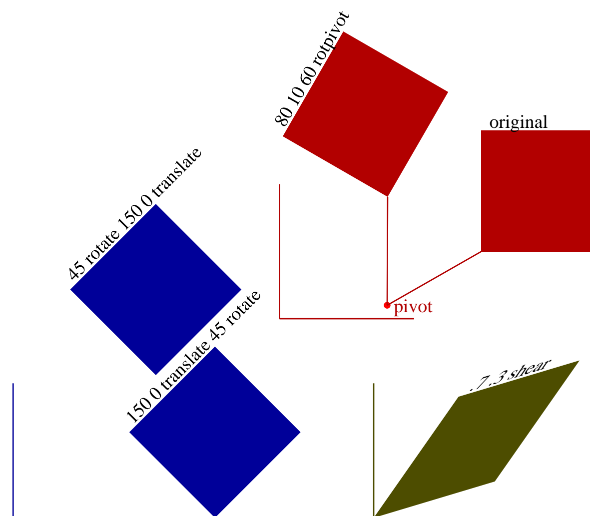

Figure 1: Examples of the affine transformation module of Table 5. A shearing, a rotation about an axis through a pivot point and the effect of interchanging the order of the application of transformations are shown (groups have the same color).

instance at least one rotational degree of freedom in his shoulder. Once a rotation is applied it is propagated to all subsequent parts, like arm, forearm, hand and fingers. The objective of this module is to transmit the basic philosophy of the tree-like structures. Usually the transformations are not concatenated, but a parameter is modified which represents the degree of freedom of the movement. For instance the continuous rotation of the robot arm is not achieved by subsequent `rotate` commands, but by setting a single rotation parameter which is read by the rotation procedure. For instance an initial value of the rotation angle of the arm can be set by

`/angle_arm 45 def`

and a 20 degree clockwise rotation is achieved by

`/angle_arm angle_arm 20 sub def.`

In Fig. 2 one of the student projects is shown in two scenarios. The robot has some typical degrees of freedom and can be manipulated in a command oriented interface. The interactivity of a PS application treated in module 8 is achieved by implementing a command interface to the interpreter which changes graphical parameters of the system and refreshes the produced image. The robot of Fig. 2 does work exactly in this way. The file "robot.ps" with the definition of the system is loaded by `(robot.ps) run` in the PS interpreter shell. After parsing the file, the dictionary has now the possible commands that can be executed. The following code shows the definition of the rotation angle and the interactive user command that changes it.

```
% degree of freedom of right arm:  rotation
/rightArmAngle 0 def
% ...  other parameters
/RotRightArm { % Usage:  Theta RotRightArm
   /rightArmAngle exch rightArmAngle add def
   erasepage Robot
} def
```

The right arm (with the gripper) has been rotated by $-30°$ by the command `-30 RotRightArm`. The situation before and after can be observed in Fig. 2.

PostScript was not conceived for an event-driven environment like for instance OpenGL. Nevertheless with a few tricks an animated behavior can be achieved. This is presented in

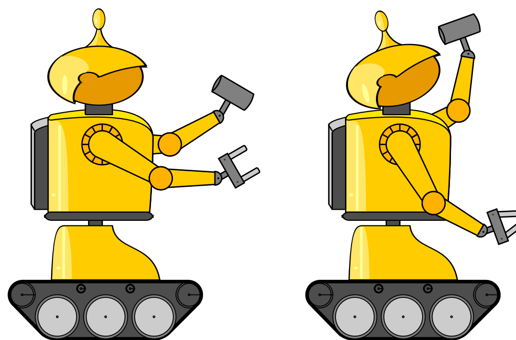

Figure 2: Hierarchical modeling of a robot. Left image shows the object in its initial position. Right image shows the robot after a sequence of parameter modification commands has been executed, in this case a 15° rotation of the head, a $-30°$ rotation of the right arm, a 30° rotation of the left arm, a 20° rotation of the left forearm and hand, and a 70% closing of the right hand.

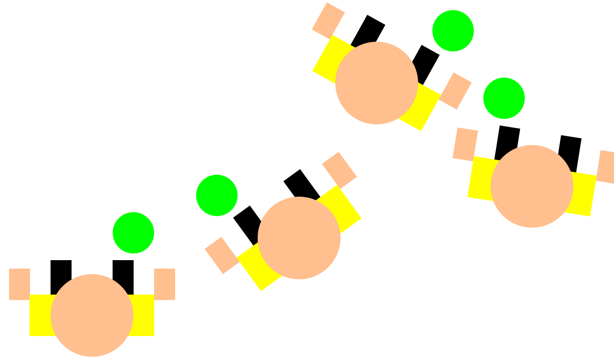

Figure 3: A sequence of four snapshots of an animation of a dribbling football player. In the animation only one player is shown at a time.

module 9. An emulated sleeping of the main process can for instance be done by the the following sub-program.

```
/sleep { %Usage:  millisec sleep
   /duration exch def % get duration of pause
   /start realtime def
   {
         % continue if time limit not yet reached
         realtime start sub duration ge
         { exit } if
   } loop
} def
```

In Fig. 3 four frames of a player are shown where the position and orientation is randomly modified and the new image is shown after a short period of time to suggest a movement.

Fig. 4 shows different scenarios of a complex interactive application solely written in PostScript, originating from a student project. It is a football game where a series of sophisticated commands can be executed, like fouling, trying to dribble the adversary, expulsion of the player, among others. The implementation of the game shows the capability of the PS language on modeling and visualization and that it even is possibly to create interactivity.

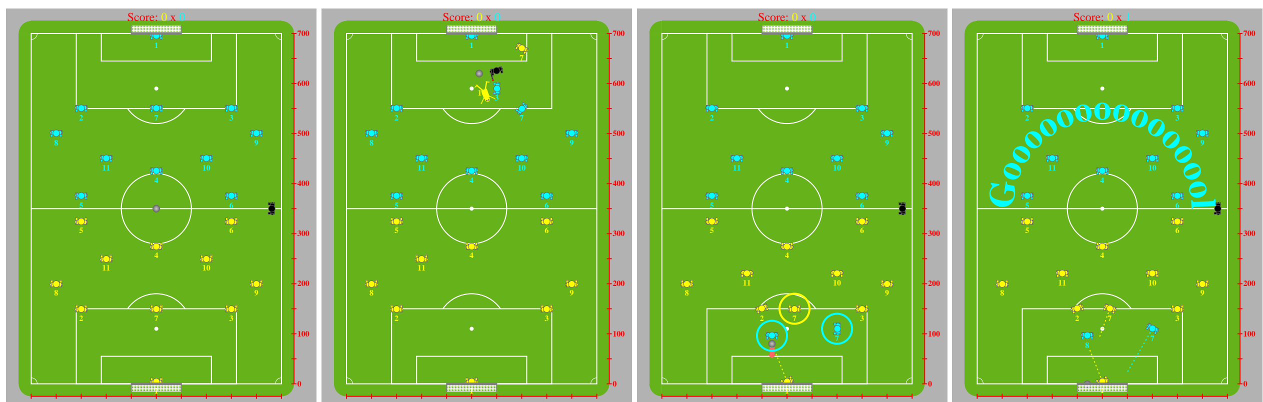

Figure 4: Four scenarios of a complex interactive computer graphics application implemented in PostScript

4. Evaluation

We teach “Introduction to Computer Graphics” at the Federal University of the state of Espírito Santo in Vitória, Brazil for undergraduate students in their eight semester for the courses of Computer Engineering and Computer Science. Our curriculum does not differ from standard introductory computer graphics courses. Hardware, color models, geometric transformations, viewing and projection, shading and texture are some of the topics. The proposed use of PostScript is intended only for the 2-D part when geometric transformations and hierarchical modeling is presented. Usually two programming projects are demanded in the course, one should be programmed in PS and the second one in OpenGL. When the concepts of affine transformations and the matrix stack mechanism have been learned in PS the students do not have any problems to extrapolate this knowledge to the 3-D OpenGL environment. It is therefore useful to learn these concepts in one language and after apply them in another (without having to repeat the theory). The second project in OpenGL is usually something in the complexity of a flight simulator or interactive game.

The students usually have not much difficulty to program in PS since they have a solid base in programming and usually are familiar with the abstract data type “stack”. They recognize the special challenging nature of implementing a computer graphics project in this programming paradigm and usually show a high degree of satisfaction after the successful completion of the project. Besides, the knowledge of pure PS programming is sometimes the less complicated manner to produce a complex figure, especially when the graphical elements are mathematically defined. Instead of writing the drawing commands in an external software and then producing an image file which is eventually included in a document, if the author dominates PS he can produce the vector graphics directly.

5. Conclusions and future work

In this paper we have illustrated the potential of the programming language PostScript to act as a complementary didactic tool for teaching basics in computer graphics. We consider our work as original since to the best of our knowledge the use of PostScript to teach an introductory computer graphics course is rare, if not unique.

Obviously for more demanding applications outside its specialized world of hard copy devices, PS is not the right tool, but for teaching the basic concepts like affine transformations and hierarchical modeling it is a choice. The concepts can later be reused in OpenGL programming.

Besides, usually students do learn an interactive programming language like C++ or Java during their studies. The knowledge of a completely different programming paradigm, as in the case of PS, the stack oriented processing clearly enriches the scope of the student’s capabilities.

A final observation can be made to this paper in itself. All figures in it are not drawn by an external graphics software, but rather are programs of the programming language PostScript. The compressed LaTeX source together with the encapsulated PostScript images has only 44 Kilobytes in size, showing the expressive power of the language.

Acknowledgments

We would like to thank

- Tobias Colombo for the preparation of the figures of the robot soccer interactive application,
- the CNPq projects 305128/2007-8 and Casadinho 620165/2006-5 for financial support.

References

[1] Adobe: *PostScript language tutorial and cookbook*. Adobe Systems Inc., 1985.

[2] Adobe: *PostScript language program design*. Adobe Systems Inc., 1988.

[3] Adobe: *PostScript language reference manual*. Addison-Wesley, 3nd ed., 1999.

[4] Artifex_Software_Inc: *Ghostscript, 2006*. `http://www.ghostscript.com`.

[5] D. Byram-Wigfield: *Practical PostScript – A Guide to Digital Typesetting*. Cappella Archive Book on Demand Limited Editions, The Steps : Foley Terrace : Great Malvern : England WR14 4RQ, 2000.

[6] B. Casselman: *Mathematical Illustrations, A Manual of Geometry and PostScript*. Cambridge University Press, 2005.

[7] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes: *Computer Graphics, Principles and Practice*. Addison-Wesley, 2nd ed., 1996.

[8] G.C. Reid: *Thinking in PostScript*. Addison-Wesley, 1990.

[9] M. Segal, K. Akeley: *The OpenGL© graphics system: A specification (Version 1.5)*. Technical report, Silicon Graphics, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, 2003.

Appendix A: PostScript example programs

In order to familiarize the reader with the syntax of the PostScript programming language, some non-graphical and graphical examples are given. The presentation of the non-graphical code permits the reader to compare it to some standard tasks in general purpose programming languages. The sub-program in Table 6 calculates the maximum of three numbers. In Table 7 a boolean answer is given to the question if a certain element is contained in an array. The search stops if the element has been found or the end of the array has been reached. A simple 2-D iterative figure is the chessboard of Table 9.

Recursion is a very interesting concept in programming, see for instance the calculus of the factorial in Table 8. If the concept of recursion is of graphical nature then it becomes easier for a student to understand it. PostScript by its stack oriented nature is ideal for the implementation since the various levels are memorized on the stack. The Sierpinski triangle

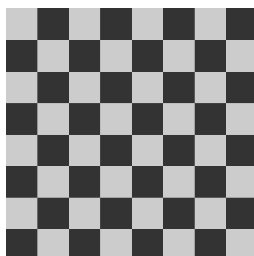

Figure 5: Chessboard produced by sub-program of Table 9

is an example of such a graphical recursive concept. It is a self-similar fractal that can be visualized easily by the program in Table 10. Observe that in the recursion no dictionary variable constructs `/var ... def` are used, since the symbols in the dictionary are unique variables and their value would be destroyed in other levels of recursion because the same symbol would be modified.

```
% Usage: num1 num2 num3 max3
/max3 {
  /c exch def
  /b exch def
  /a exch def
  a b gt
  {
    a c gt {a} {c} ifelse
  }{
    b c gt {b} {c} ifelse
  } ifelse
} def
```

Table 6: Sub-program for calculating the maximum of three numbers

```
% Usage: num array contains, e.g. 3 [4 3 1] contains ==
/contains {
  /ar exch def % get the array
  /elem exch def % get the element
  /found false def % Initialize boolean variable ''found''
  /pos 0 def % Initialize position index of array
  /arlen ar length def % get length of array
  {
    % check if element matches at position
    /found ar pos get elem eq def
    % check if array range is valid
    /endarray pos arlen 1 sub ge def
    % abort search if found or outside range
    found endarray or exit if
    /pos pos 1 add def % increment position
  } loop
  found % leave result on stack
} def
```

Table 7: Sub-program for checking if an array contains an element

```
% Usage: n fac
/fac {
  dup 0 eq
  {
    pop 1
  }{
    dup 1 sub fac mul
  } ifelse
} def
```

Table 8: Sub-program for calculating the factorial $n!$ of a number n

```
/is_dark true def
/side 20 def
/square { % Usage:  is_dark square
  { 0.2 setgray } { 0.8 setgray } ifelse
  gsave side side scale unitbox grestore
} def
/chessboard { % Usage:  chessboard
  0 side side 7 mul
  {
    /y exch def
    0 side side 7 mul
    {
      /x exch def
      gsave
        x y translate
        is_dark square
        /is_dark is_dark not def % toggle flag
      grestore
    } for
    % one extra toggle at last column
    /is_dark is_dark not def
  } for
} def
```

Table 9: Sub-program for drawing a chessboard

```
% Usage:  level length Sierpinski_triangle
/Sierpinski_triangle {
  exch % swap the two arguments
  dup % get the level to test it against 0
  0 eq
  {
    pop % destroy 'level'
    gsave
      newpath
      0 0 moveto
      dup 0 lineto
      dup 0 translate
      120 rotate
      dup 0 lineto
      dup 0 translate
      120 rotate
      0 lineto
      closepath fill
    grestore
  }
  {
    1 sub
    exch
    2.0 div
    % now the stack is length/2 , level-1
    gsave
```

```
          2 copy Sierpinski_triangle
          dup 0 translate
          dup 0 translate
          120 rotate
          2 copy Sierpinski_triangle
          dup 0 translate
          dup 0 translate
          120 rotate
          Sierpinski_triangle
        grestore
      }
      ifelse
    } def
```

Table 10: Sub-program for drawing the Sierpinski triangle fractal

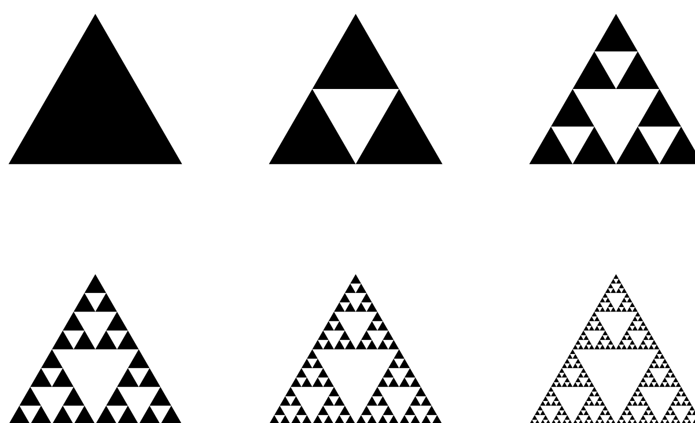

Figure 6: Sierpinski triangle fractals of levels 0 − 5 produced by sub-program of Table 10

Received August 4, 2008; final form December 29, 2008