PDF on the Fly

Version 1.0a24

1.0 Introduction

This is documentation for the Portable Document Format Perl5 Library (PDF-PL), a Perl5 Library designed to assist in writing Perl5 programs that create Portable Document Format (PDF) files as used by Adobe's® Acrobat® products. The current distribution can be found at http://www.ep.cs.nott.ac.uk/pdf-pl/. For a complete specification of PDF see *Portable Document Format Reference Manual* by Tim Bienz and Richard Cohn, published by Addison-Wesley Publishing Company. Also check Adobe's World Wide Web (WWW) site for possible updates via: www.adobe.com.

Of course, a simple way to generate PDF files is by using almost any application program and either printing through the print driver PDFWriter®, or by using the printer driver to make a PostScript® file and then use the Acrobat Distiller® to create the PDF file. Making a PDF file using a Perl5 program might be considered doing it the hard way, but if the results of a WWW query are most suitably expressed as a PDF file generated on a WWW server, then this library provides one way to do it "on-the-fly" today. Hopefully more high level techniques will be forthcoming.

PDF was designed as an electronic document format that allows fast and efficient random access to any page of the document. For example, when a document is displayed on a screen, the user desires fast access to any page even if the document consists of hundreds or even thousands of pages. PDF files are a little tricky to create because they contain "objects" that are addressed by their byte offset within the file and which can be accessed randomly, that is, without reading the file serially from front to back. This is accomplished using a set of (fixed format) object reference tables at the end of the PDF document and by packaging the entire document information into objects of various kinds: pages, dictionaries, text streams, resources, and more (see the above referenced PDF Manual).

The PDF-PL is a library of functions for the generation of a PDF file and should suffice to build an error free PDF file. The functions are all written in Perl5. Since it is intended that the PDF builder should use these functions within his own Perl5 program, familiarity with Perl5 is assumed. Familiarity with PDF is also assumed. The user of PDF-PL does not need to know all the details of generating a complete and correct PDF file as the library handles most of that. However, the user will have to understand the details of "page content" operators in PDF in order to put text and graphics onto the pages of the files.

This is a first version of this library and as such has limited function. Primarily it makes it easy to make multiple page documents containing text and line art. Some elementary text formatting features are provided by the library but for the most part the user has to write Perl5 code to handle the detailed placement of text and graphics on the page. So far, this PDF-PL does not support image placement on pages or functions that go beyond page content like "links", "bookmarks", or "annotations." Our informal plans call for support for these PDF features to be added in a future release of this PDF-PL.

2.0 Building the PDF

We will simply explain how to generate correct PDF files by using appropriate examples, simple ones first and then progressing to more complicated ones. Together with the list of functions in the *Function Reference List* given later in these notes, the user should be able to learn how to generate correct, i.e., error free PDF files. Incorrect use of the PDF-PL provided functions may result in an erroneous PDF without the builder being informed by the PDF-PL of these errors. We recommend that the user test the results of Perl5 programs by attempting to view the PDF files with, for example, the Acrobat Reader or Exchange. A typical user might have this document, the PDF manual and a Perl5 manual all available on the desktop.

2.1 PDF Top Level Structure

The PDF file consists of four basic consecutive building blocks:

- The header
- The body: a collection of PDF objects
- The cross reference section
- The trailer

If a PDF has been modified (for example by generating annotations by using the Adobe Acrobat product) these four original blocks are followed by one or more triplets, one for each updating session. Each triplet consists of the following three building blocks:

- The updated body portion
- The updated cross reference section
- The updated trailer

Note that the original portion of a PDF file (the first four blocks) will never actually be changed. Only those sections which by the update have been modified are recreated and saved in the triplet blocks following the original blocks. This is another of the typical properties of the PDF file.

The current version of these notes describes only building an original PDF file, that is one that has not been updated yet.

The body of a PDF file consists of a sequence of PDF objects. These objects are numbered by the internal routines as they are created, but the sequence in which they are written to the output file need not be in ascending order of the numbers or, for that matter, in any order. The objects contain everything there is to know about the document. Header, cross reference section and trailer are constructed by the library routines internally and written to the output file. For this construction process the library routines make use of information collected during the time the user generates the content of the objects in the body.

Using PDF-PL, a user written Perl5 program for generating a PDF file has the following general form:

- Open the output file
- Generate the pages and their content
- Close the output file

Since PDF files are accessed randomly by objects, as one of the later sample Perl5 programs will illustrate, we can build the pages in any order. Generally the pages are written to the file in the same order in which they are created by the Perl5 calls.

The *Function Reference List* later in these notes lists the functions the user needs for building a simple PDF file. The functions are listed in alphabetical order.

2.2 The Marking of the Page

The purpose of page contents objects of a PDF file is to put marks (graphic or text) on pages. This is done by suitable PDF-PL functions invoked between a **beginGraphic** and **endGraphic** invocation. For lack of a better name we call the program area between these begin/end functions the graphic marking area or *GMA*. The PDF-PL library functions match the basic marking operators of PDF and this document gives the simple mapping between the descriptions found in the PDF Manual and the appropriate Perl5 calls.

One significant difference should be noted. There is an optimization done for certain state setting calls and redundant ones may be ignored by the PDF-PL and will not result in any output in the PDF file. This could be quite confusing when the results of a particular Perl5 call don't show up in the file. This will have been because the results would have been redundant and leaving those operations out will not change the behavior of the resulting PDF file. If the users would like to force the library to output the operators in any case, certain functions that begin with "force" rather than "set" can be used.

While in the GMA (i.e. between corresponding beginGraphic and endGraphic function invocations) the PDF builder invokes, on the one hand, environment changing functions and, on the other hand, also uses functions which actually put marks on the page, i. e. position text, draw lines and curves, etc. In the *Function Reference List* a brief description is given for all functions and their parameters are defined.

It must be noted here that invocation of a PDF-PL function never puts any marks anywhere on a page. The more correct phrasing is: it puts the appropriate information into the PDF so that whenever this portion of the PDF is rendered on an output medium (say a screen or paper) the intended marking occurs. Throughout these notes we will informally use the shorthand wording *"is printed," "is drawn" or "is rendered"* instead of always saying *"appropriate information is put into the PDF so that when rendered on an output medium the intended effect is achieved."*

While in the GMA the PDF builder invokes both graphic and text *environment setting* functions and the graphic and text *marking* operations. However, the text marking operations must occur while in the text marking area (TMA) of the program. A TMA is defined as the area in the program starting with the **beginText** function and ending with the **endText** function. Since **beginText** and **endText** must be invoked within a GMA, the TMA is

a subarea of the GMA, i.e. entirely contained within it. It follows that graphic marking function invocations must occur within the GMA but not within a TMA. Text marking function invocations must occur only within a TMA. Graphic environment setting functions must occur within the GMA. And they may also occur within a TMA (and propagate outside of it when the TMA ends by an **endText** invocation). The same is true for text environment setting functions.

Note that there are text marking functions in the PDF-PL which, in a matter of speaking, contain an entire TMA within themselves. These are the text block printing functions **textBlockLeft**, **textBlockRight**, **textBlockCenter** and **textBlockOffset**.

Many of the PDF-PL functions are primitives, i.e., they are implementations of one single PDF operator. But these text block functions (as well as a few a others to be introduced later) output a set of such primitive functions. When they are invoked, the PDF builder must specify font name, font size and text leading as parameters. After their invocation the specified text parameters remain active just as they would if set within a TMA. Other currently still active text environment setting functions apply, of course, also to the text rendered by the text block functions.

These are the defaults for the environmental graphic and text variables at the start of a new GMA:

- dash pattern — [0], i.e., solid line
- fill color — black
- flatness — 0, i.e., the device default
- font name — UNDEFINED
- font size — UNDEFINED
- line cap style — 0, i.e., butt end
- line width — 1 user unit
- miter limit — 10
- line join style — 0, i.e., mitered
- stroke color — black
- text character spacing — 0
- text horizontal scale — 100, i.e., 100%, no scaling
- text leading — 0
- text rendering — 0, i.e., fill the text with color
- text rise — 0
- text word spacing — 0

These are the defaults for the environmental graphic and text variables as they are valid at the start of a new page:

- clipping path — crop box
- crop box — media box (page size)
- current point — (0, 0), i.e., lower left page corner

The terms “fill color” and “stroke color” refer to the use of color for the operations of filling an area and stroking (i.e., drawing) a line. Filling refers to the filling with color of an area encircled by a defined path. Stroking refers to drawing a line along a defined path. Stroking is a term specifically defined for the PDF to mean drawing a line.

The following graphic and text related functions are grouped into graphic marking functions, text marking functions and environment changing functions. The list also shows the corresponding operators as they are used in the PDF. Functions without an operator (indicated by “--”) are unique to the PDF-PL and hence have no direct equivalent in the PDF.

Graphic environment changing functions are:

- **setColors** --
- **setDashPattern** d
- **setFlatness** i
- **setJoinStyle** j
- **setLineCapStyle** J
- **setLinewidth** w
- **setMiterLimit** M

The corresponding force functions also belong into this list. They are obtained by replacing the prefix “set” with the prefix “force.” All functions in the list have a force function which do exactly the same as the set functions except for the time when the function is actually put into the PDF stream object.

The text environment changing functions are:

- **setCharSpace** Tc
- **setFont** Tf
- **setTextHorizScale** Tz
- **setTextLeading** TL, TD
- **setTextRender** Tr
- **setTextRise** Ts
- **setTextWordSpace** Tw

Again there are corresponding “force” functions for each function in this list. They are obtained by replacing the prefix “set” with the prefix “force.”

Font and font size can also be changed by being specified as parameters of the **beginText** function (see the function description in the *Function Reference List*) and by **setFont**.

Although the PDF-PL is not a formatter there are environment setting functions for line spacing (leading), word spacing, character spacing and super/subscripting (rise with positive or negative parameter value). The function **setTextHorizScale** sets a scale factor which is horizontally applied to the text string before it is actually rendered on a screen or

printer. The function **setTextRender** specifies how text is to be rendered, for example, filled with color, stroked only, or stroked and filled, etc.

Functions for graphically marking the page while in the GMA are not listed here. They are introduced in the next section, *The Current Point and Paths*.

Functions for marking the page with text while in the TMA are:

- **textCenter**
- **textLeft**
- **textRight**

The text block functions also mark the page with text but they are invoked within a GMA (remember, you can look at it as if they carry their own TMAs with them).

- **textBlockCenter**
- **textBlockLeft**
- **textBlockOffset**
- **textBlockRight**

One might consider **endText** to be another function which belongs into this category. Although it in itself is not contributing to marking the page, it gives rise to the actual putting of text into the output file by the PDF-PL internal routines.

2.3 The Current Point and Paths

In the GMA one of the environment variables is the “current point.” The current point is used to define a path. A path is a sequence of line segments (straight or Bezier curves). A path can be stroked (i.e. drawn), filled (when closed) or both. The current point is valid only outside a TMA, since graphic marking functions may not be invoked inside a TMA.

The kind of stroking is determined by the setting of the environment variables: line width, dash pattern, miter limit, line cap style, line join style and stroke color. Stroking means drawing a line along the path, such that the line is centered on the path.

A path is defined by using the current point changing functions (the letters are again the corresponding PDF operator codes):

- **closepath** h
- **curveto** c
- **endpath** n
- **lineto** l
- **moveto** m
- **rectangle** re
- **vCurveto** v
- **yCurveto** y

Nothing is committed yet to the PDF file when they are invoked.

The function **moveto** moves the current point. The functions **lineto**, **curveto**, **vCurveto** and **yCurveto** append new Bezier curves (segments) to the current path, and leave the current point at the end of the last segment. The function **rectangle** adds a rectangle to the current path and **closepath** ends a path by appending a straight line from the current point to the beginning of the path. The function **endpath** just ends a path without closing it.

These are the path committing functions:

- **closepathFillStroke** b
- **evenOddFill** f*
- **evenOddFillStroke** B*
- **fill** f
- **fillStroke** B
- **stroke** S

Now the defined path is committed to the PDF file, either as a filled path (by the **fill** or **evenOddFill** functions), as a stroked path only (by the **stroke** function) or as both filled and stroked (by the **fillStroke** or **evenOddFillStroke** functions). Note that the **closepathFillStroke** function does the same as the **fillStroke** function except it is preceded by closing the path as if the **closepath** function were called first.

The occurrence of any of the stroking and/or filling operators "consumes" the path. The path ceases to live. The only exception is the existence of a clipping function (**clip** or **eoclip**) in the path definition. Even if the path has been consumed its ability to act as a clipping path lives on. This ability disappears only by restoring an environment as it existed before the clipping path was defined (see the **save** and **restore** functions in the section *The Function Reference List*).

Note that the **fill**, **fillStroke** and **closepathFillStroke** functions fill the path using the nonzero winding rule to determine which region to fill. **EvenOddFill** and **evenOddFillStrok**e fill the path using the even-odd rule to determine the filling region. For a detailed discussion of these rules see the section *Path Painting Operators* in the PDF Manual.

2.4 Clipping

In the PDF it is possible to specify more than one separate paths and designate one or more of them to be clipping paths.

When a clipping path has been defined and another path is to be filled or stroked or both, then only the intersection of clipping path and the fill/stroke path are actually filled/stroked. For an sample program see the section *Clipping Path*.

The default clipping path on a page is the crop box (or its default, the media box, which is the page size). Hence nothing need be done by the PDF builder as long as no clipping effects are desired. In fact, this is the most desirable situation. The operation of clipping is

an expensive one. Therefore it is advisable always to deactivate an existing clipping path once it is no longer needed. Even if the current clipping path does not effect any of the paths currently to be filled its mere existence costs. It is good programming practice and economic use of the available resources to use available environment saving and restoring functions for saving and restoring the clipping path (for **gsave** and **grestore** see the section *The Reference Function List*).

A second definition of a clipping path is given by a string of text, or more precisely, the outlines of the characters comprising the string of text. Current font and size parameters etc., of course, apply. The characters of text strings are marked to act as clipping paths by the **setTextRender** function. For a sample program see the section *Text as Clipping Path*.

2.5 User Units and Coordinate System

Almost all parameters used in the functions of the PDF-PL specify linear dimensions, for example, coordinates of a point on the page or width of lines to be drawn. They are to be specified in user units. A user unit is 1/72nd of an inch which is very closely equal to one printer's point. The origin of the user coordinate system on the page is the left lower corner. In the PDF user units are the default units. The actual length of a user unit may change by transformations of the coordinate system.

By using the **ctm** function it is possible to transform the coordinate system to suit a particular application. Possible transformations are the operations of translation, rotation and scaling. After executing a **ctm** function, the length of a unit in either x or y may have changed. The directions into which x and y point might have also.

In other words, if a PDF-PL function is invoked after some coordinate system transformation, which includes scaling in x-direction by a factor of 2, then a specified length of 3 user units will on the page appear as a length of 6 points.

2.6 Error Checking

Error checking of user supplied data (parameters) as well as correct sequencing of user supplied invocations of the PDF-PL functions is not guaranteed. In particular, the PDF builder is responsible that the effect of positioning functions, as, for example, **moveto** and **lineto**, do not exceed the dimensions of the page.

3.0 Sample Programs

In this section sample programs will demonstrate the use of the functions of the PDF-PL. The functions have all been tested and the code was inserted here into these notes directly from the test library. However, adjustments had to be made to the program text, for example, the name of the PDF-PL library was changed to a generic indicator after we used an explicit name necessary to test the programs in our environment. Therefore accidental errors cannot be ruled out.

Note that these sample programs also explicitly augment the definition of the functions in *The Function Reference List*. They are a vital part of this documentation.

3.1 The Simplest PDF (The “Hello World!” Case)

Traditionally, generating a salutation to the world is the start for learning a new programming language or, as the case is here, beginning to use a new library of functions for generating documents. Here is a simple Perl5 program printing these classical words in 24 point Helvetica Bold on a default size page six inches up from the bottom and starting one inch over from the left edge.

```
##### Test t1.pl  --  Hello World!  ########################
BEGIN {
    $PDF_LIB = $ENV{“$PDF_LIB”};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;                  ##### USE lines
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();     ##### OPEN FILE lines
$pdf->pdfOpen (’./t1.pdf’);

$pdf->newPage ();               ##### Page line

$g = $pdf->beginGraphic ();     ##### BEGIN GRAPHIC line

                                ##### TEXT lines
$g->beginText (’Helvetica-Bold’, 24);
$g->textLeft (72, 432, “t1.pl:  Hello World!”);
$g->endText ();

$g->endGraphic ();              ##### END GRAPHIC line

$pdf->pdfClose ();              ##### CLOSE FILE line
#####################  E N D  ##############################
```

Note that in this and all following sample programs there is an empty parameter list used for all invocations of subroutines which do not require parameters. Use is made of this Perl5 option in order to improve readability of the program text.

The first executable line in the t1.pl sample program is a compile time statement of Perl5. If the Pdf library is not installed in a standard search place, these lines cause the program to look for the Pdf directory in the directory given by the PDF_LIB environment variable.

The **USE** lines are a notification to the Perl5 interpreter about which packages are going to be needed and in which directory they can be found. In other words, the PDF-PL directory specified in the **BEGIN** statement contains a directory named Pdf, which in turn contains the PDF-PL module Files, Font and Graphic.

In the first of the **OPEN FILE** lines a local Perl5 variable ($pdf) containing a reference to a new File object is defined. In the second line $pdf is used to open a user specified output file.

In the **PAGE** line a new page (here the first) of the output PDF file is started. One such function invocation is required for each new page.

With the **BEGIN GRAPHIC** line begins a new GMA. A reference to the new GMA is returned and saved since it is now needed to qualify the functions to be used for putting marks on the page.

The **TEXT** lines define a TMA within which one line of left adjusted text is put out. The **beginText** function also provides specification of font name and font size.

The **END GRAPHIC** line ends the GMA.

The last line closes the output file.

It should be noted that the functions invoking the end of the GMA and the end of file are required. No PDF-PL diagnostics will indicate their absence yet the generated erroneous PDF will not be processed by Acrobat. In fact, running under Windows Acrobat caused Windows to stall.

Note: There are two Perl5 objects used in this program sequence represented by the variables named $pdf and $g. The “page” created by **newPage** and the output file created by **pdfOpen** are not materialized as Perl5 objects in the example but reside within the “file” Perl5 object. Similarly, the **beginText** does not make a Perl5 object but begins a text state within the graphic object $g. In these notes we call this text state a TMA.

3.2 Two Pages

In the second sample program we will generate a PDF file with two pages and a mix of graphic and text content.

```
##### Test t2.pl  --  Two Pages  ############################
BEGIN {
     $PDF_LIB = $ENV{"$PDF_LIB"};
     unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;
```

```
my $pdf = Pdf::File->new();
$pdf->pdfOpen (’./t2.pdf’);

$pdf->newPage ();            ##### start of first page

$g = $pdf->beginGraphic ();

$g->beginText (’Helvetica’, 18);
$g->textLeft (72, 650,
    “t2.pl:  On this page text and graphics are drawn using the”);
$g->textRight (540, 610,
    “default environment (except for font and fontsize)!”);
$g->endText ();

&strokeLine (120, 200, 370, 450);
&strokeLine (120, 250, 370, 500);

$g->rectangle (400, 300, 100, 100);
$g->fill ();

$g->endGraphic ();

$pdf->newPage ();            ##### start of second page

$g = $pdf->beginGraphic ();

$g->setColors (’f’, ’red’);

$g->beginText (’Helvetica-Bold’, 24);
$g->textLeft (120, 600, “t2.pl: Second Page!  The Fun starts!”);
$g->endText ();

$g->setLinewidth (5);
$g->setColors (’s’, ’blue’);
$g->setColors (’f’, ’yellow’);
$g->rectangle (250, 400, 100, 100);
$g->fillStroke ();

$g->setColors (’s’, ’green’);
&strokeLine (245, 200, 295, 275);

$g->setColors (’s’, ’darkblue’);
&strokeLine (305, 275, 355, 200);

$g->setColors (’s’, ’red’);
&strokeLine (247, 192, 353, 192);
```

```
$g->endGraphic ();

$pdf->pdfClose ();


sub strokeLine {
       $g->moveto ($_[0], $_[1]);
       $g->lineto ($_[2], $_[3]);
       $g->stroke ();
}
#####################  E N D  ##############################
```

Only those features of PDF-PL functions will now be described which were not already mentioned in the previous example.

In the sample program t2.pl more than one line of text is generated within the first TMA (beginText ('Helvetica-Bold', 18). Note that the first line is left adjusted, starting one inch to the right of the page edge. The second line is right adjusted. Its right end is positioned one inch to the left of the right edge of the page.

The drawing parameters default to those initially valid when a new GMA is started.

The lines following the TMA (two strokeLine lines) generate two straight lines having the default width of one user unit and drawn with the default stroke color of black. Note: The strokeLine function is an internal subroutine to this sample porgram. It produces a line on a page by executing the PDF operators moveto, lineto and stroke. Consequences: The current path is undefined i.e. consumed after strokeLine is executed.

The next two lines generate a square filled with the default fill color black. Note that the square is only a consequence of the proper choice of the two defining points of the rectangular box.

On the second page of the PDF after beginning the GMA a new current color (red) is defined for all filling ('f') operations. Filling includes the coloring of text. Hence the following text line is drawn in red.

Next the line width used by the drawing routines is set to five user units. Then the stroking color is set to blue and the filling color changed to yellow. Then a two colored square is drawn. It is to be stroked ('s') and filled ('f'). Hence its five user units wide outline appears in blue and its interior is filled with yellow.

The last figure consists of three differently colored lines.

3.3 Repeated GMAs

In the sample program t3.pl we want to show how the same GMA can be used repeatedly. As an example we want to generate the same heading and footing text on all pages of a

document. Then we use additional functions to generate the variable text on the pages, the page numbers.

```
##### Test t3.pl  --  Repeated GMAs  ######################
BEGIN {
     $PDF_LIB = $ENV{"$PDF_LIB"};
     unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new(1);
$pdf->pdfOpen (’./t3.pdf’);

$pdf->newPage (3);            ##### init. of third page
$pdf->newPage (1);            ##### init. of first page
$pdf->newPage (2);            ##### init. of second page

$g = $pdf->beginGraphic (1);  ##### goes to page 1

$xc = 4.25 * 72;
$yc = 10 * 72;

$g->beginText (’Helvetica’, 18);
$g->textCenter ($xc, $yc, “t3.pl:  Sample H e a d i n g”);

$yc = 72;

$g->textCenter ($xc, $yc, “Sample F o o t i n g”);
$g->endText ();

$g->endGraphic ();

$g->addGraphic (2);          ##### now added to page 2
$g->addGraphic (3);          ##### now added to page 3

foreach $pn (1, 2, 3) {      ##### add the page numbers
     $g = $pdf->beginGraphic ($pn);
     $g->beginText (’Times-Roman’, 18);
     $g->textCenter ($xc, 36, $pn);
     $g->endText ();
     $g->endGraphic ();
}

$pdf->pdfClose ();
```

```
#####################  E N D  ##############################
```

In the sample program t3.pl we first start three pages by three repeated **newPage** function invocations. Just for demonstration we use explicit page numbers and initialize these pages not even in sequence. Between these page functions we would ordinarily find all the content generating functions for the current page. If the page numbers were not specified the pages would be automatically numbered sequentially starting with 1.

Then text strings for heading and footing of a page are generated within a GMA. We pass the parameter 1 to the **beginGraphic** function. This attaches the generated text to page 1. In the previous sample programs we used the **beginGraphic** function always without a parameter. Then the contents generated were automatically attached to the most recently initialized page.

Now we use two further **addGraphic** functions with page numbers 1 and 2 respectively, as parameters for attaching the same text strings (the most recently ended GMA) to the specified pages.

We now want to print the page numbers correctly on each of the pages. We use a foreach statement for the loop. Within each pass through the loop another GMA is begun and ended. By using the page number as parameter on the **beginGraphic** function the generated content is attached to the correct page.

3.4 Save and Restore the GMA Environment

In the sample program t4.pl we show the usage of the **gsave** and **grestore** functions.

```
##### Test t4.pl  --  Save/Restore Environment  ############
BEGIN {
      $PDF_LIB = $ENV{"$PDF_LIB"};
      unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t4.pdf');

$pdf->newPage ();            ##### init. page

$g = $pdf->beginGraphic ();

$x = 1.5 * 72;
$y = 9 * 72;
```

```
$g->beginText (’Helvetica’, 12);
$g->setColors (’f’, ’red’);
$g->textLeft ($x, $y, “t4.pl:  First Environment (Helv. 12pt,
     red)”);
$g->endText ();

$g->gsave ();

$y -= 50;

$g->beginText (’Times-Roman’, 24);
$g->setColors (’f’, ’blue’);
$g->textLeft ($x, $y, “Second Environment (Times 24pt, blue)”);
$g->endText ();

$y -= 50;

$g->grestore ();

$g->beginText ();
$g->textLeft ($x, $y, “First Environment again”);
$g->endText ();

$g->endGraphic ();

$pdf->pdfClose ();
#####################  E N D  ##############################
```

In the t4.pl sample program we demonstrate saving and restoring of the text and graphic environment. In each of three successive TMAs a line of text is generated. For the first line 18 point Helvetica and red fill color was chosen. The second line is printed in blue 24 point Times-Roman. After the first TMA the environment has been saved (**gsave**). After completing the second text environment the first environment is restored again (**grestore**). Then a third text line is printed but without setting font or color. As the example demonstrates the environment has been saved and the third line appears again in the color and font of the first line.

3.5 Line Samples

In this example different line samples are demonstrated. Also the size of the page is changed from the default size (which is 8 and a half by 11 inches) to 6 by 8 inches. On each of the four pages another property of lines is demonstrated: line width, dash patterns, line cap styles and miter styles.

```
##### t5.pl  --  Line Samples on smaller page size  ########
BEGIN {
     $PDF_LIB = $ENV{"$PDF_LIB"};
```

```
     unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t5.pdf');

                              ##### change of default page size
$w = 6 * 72;
$h = 8 * 72;
$pdf->setPageSize ($w, $h);

$pdf->newPage ();         ##### start of first page: varying lines

$g = $pdf->beginGraphic ();

$x = 120;
$y = 470;

$g->beginText ('Helvetica-Bold', 18);
$g->textLeft ($x, $y, 't5.pl:  Line Width examples');
$g->endText ();

$x += 35;
$y -= 55;
$ll = 100;
$ld = 50;
$td = 115;
@lwa = ('1', '3', '5', '10', '15', '20', '25');

foreach $lw (@lwa) {
     $g->setLinewidth (0+$lw);
     strokeLine ($x, $y, $x+$ll, $y);

     $g->beginText (12);
     $g->textLeft ($x+$td, $y-5, $lw);
     $g->endText ();

     $y -= $ld;
}

$g->endGraphic ();

$pdf->newPage ();          ##### start new page: dash patterns
```

```
$g = $pdf->beginGraphic ();

$y = 470;
$x = 120;

$g->beginText (’Helvetica-Bold’, 18);
$g->textLeft ($x, $y, ’t5.pl:  Dashed Line Examples’);
$g->endText ();

$x -= 5;
$y -= 55;
$ll = 200;
$lw = 4;
$ld = 50;
$td = 215;

$g->setLinewidth ($lw);

@da = ( [4], [4], [4, 2], [3, 5], [2, 3], [6, 3, 2, 3]);
@pa = (0, 4, 0, 3, 0, 9);

foreach $i (0..5) {
      @a = @{$da[$i]};
      $p = $pa[$i];
      $g->setDashPattern (\@a, $p);
      strokeLine ($x, $y, $x+$ll, $y);

      $g->beginText (12);
      $g->textLeft ($x+$td, $y-$lw/2, $p);
      $g->endText ();

      $y -= $ld;
};

$g->endGraphic ();

$pdf->newPage ();          ##### start new page: line cap styles

$g = $pdf->beginGraphic ();

$y = 470;
$x = 120;

$g->beginText (’Helvetica-Bold’, 18);
$g->textLeft ($x, $y-5, ’t5.pl:  Line Cap Examples’);
$g->endText ();
```

```
$x += 35;
$y -= 55;
$ll = 100;
$ld = 75;
$td = 115;

$g->setLinewidth (15);

foreach $cap (0, 1, 2) {
      $g->setLineCapStyle ($cap);
      strokeLine ($x, $y, $x+$ll, $y);

      $g->beginText (12);
      $g->textLeft ($x+$td, $y-5, $cap);
      $g->endText ();

      $y -= $ld;
};

$g->endGraphic ();

$pdf->newPage ();          ##### start new page: miter styles

$g = $pdf->beginGraphic ();

$x = $w/2;
$y = $h - 60;

$g->beginText ('Helvetica-Bold', 18);
$g->textCenter ($x, $y, 't5.pl:  Miter Style Examples');
$g->endText ();

$y -= 150;
$w = 50;
$h = 100;
$x -= $w;
$ld = 150;
$td = 150;

$g->setLinewidth (15);

foreach $m (0, 1, 2) {

      $g->setJoinStyle ($m);
      $g->moveto ($x, $y);
      $g->lineto ($x+$w, $y+$h);
```

```
      $g->lineto ($x+2*$w, $y);
      $g->stroke ();

      $g->beginText (12);
      $g->textLeft ($x+$td, $y+$h/2, $m);
      $g->endText ();

      $y -= $ld;
};

$g->endGraphic ();

$pdf->pdfClose ();


sub  strokeLine  {
        $g->moveto ($_[0], $_[1]);
        $g->lineto ($_[2], $_[3]);
      $g->stroke ();
}
#####################  E N D  ##############################
```

Before anything is done in the fifth sample program, t5.pl, the default page size is changed to six inches in width and 8 inches in height using the **setPageSize** function.

Text font name and font size must be defined before any text can be put on the page within a new GMA. This is done with the **beginText** function at the beginning of the first TMA of this page.

Within a GMA, once defined, the font parameters propagate into each new TMA as long as it is contained within the same GMA. Hence in our sample program only the first occurrence of a **beginText** function within a GMA shows two parameters: here Helvetica-Bold and 18 points. Repeated use of a **beginText** function within the same GMA shows only the changed parameter: here 12 points.

For Perl5 novices: here is a common trick to overcome problems often arising from the easy (and therefore often careless) use of different data types in Perl5. The program shows $lw to be of string type because of the string assignment from the string array (@lwa) in the foreach statement. So it is used and needed in the **textLeft** function of the fourth statement in the loop body. In the first loop statement, however, **setLinewidth** calls for a numerical parameter. If we used $lw only, the **setLinewidth** function answers with an error message, saying that it expects a number and not a string. Using 0+$lw lets Perl5 convert the string into a number without changing its value and the **setLinewidth** routine is happy.

The dashed line example on the second page shall serve to explain the details of the usage of the two parameters of the **setDashPattern** function. The first parameter must be an array (actually the reference to an array, denoted here by the reverse slash, ‘\’).

For the first line the array consists only of one element: the number four. In the definition of the array of arrays, @pa, just before the foreach loop, the first element is a one-element array). The single element means that the dashed line consists of equal long dashes and gaps, each of them being four user units long. The second line is similar to the first except that the line does not start with a dash but with a gap (see below).

The third element of the @da array is an array with two elements: [4, 2]. Now, the dashes are four user units long and the gaps two. The fourth and fifth lines have similar dash patterns, only the lengths are different.

The sixth array, [6, 3, 2, 3], defines a dash pattern of a six unit dash, followed by a three unit gap, followed by a two unit dash, followed by a three unit gap, before it repeats itself. As you might guess: an odd number of elements defines a pattern where the last gap has the same length as the last defined dash.

It remains to explain the second parameter, also called phase, of the **setDashPattern** function. This parameter is a length value measured in user units. It defines where in the pattern the drawing shall be started, before the repetition sets in.

In the first line the phase is zero. The pattern starts at the beginning of the path with the dash defined by the first element of the defining array: four user units long. Now you see the difference between the first and the second lines. In the first line the drawing starts with the dash, in the second line the pattern starts as far advanced in the defined pattern as the phase specifies, here exactly with a gap. (NOTE: This is probably not what you see when you try this example. You have run across an Acrobat bug. You see the line identical to the first one, i.e. as if the phase were not 4 but 0. But the PDF-PL generates a correct PDF operator and parameters.) The last line with a phase of nine starts with the second dash in the pattern, a dash with a length of two user units.

The third page shows nothing new, but serves to demonstrate the line cap style function.

On the fourth page we use the defining of a path for the first time. We draw two lines in each loop and demonstrate the available styles of joining them. There are three functions defining the path: **moveto** followed by two **lineto** functions. This defines a path starting with the parameter point of the **moveto** function and ends with the parameter point of the second **lineto** function. It ends simply by the **stroke** function which actually does the drawing (stroking) of the path according to the current environment. This environment consists mostly of the defaults: 15 user units line with, default cap style (butt end), default join style mitered), default dash pattern (solid line) and default color (black).

3.6 Bezier Curves

In the sample program t6.pl we demonstrate the use of the curveto function which generates a Bezier curve. This is the way to draw any shape curve in PDF.

```
##### t6.pl  --  Bezier Curves  ############################
BEGIN {
     $PDF_LIB = $ENV{"$PDF_LIB"};
```

```
     unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t6.pdf');

$pdf->newPage ();

$g = $pdf->beginGraphic ();

$xc = 4.25 * 72;
$yc = 9 * 72;
$g->beginText ('Helvetica-Bold', 18);
$g->textCenter ($xc, $yc, 't6.pl:  Bezier Curve Examples');
$g->endText ();

$D = 250;                 ## width of curve sample
$dx = 50;                 ## init. x-diff of Curr.Pt. and P1
                          ## and of P2 and P3
$d = $D - 2*$dx;          ## init. x-diff. of P1 and P2
$dy = 100;                ## height of curve
$tdu = 12;                ## upper vertical text adjustment
$tdl = 16;                ## lower vertical text adjustment

$x = $xc - $dx - $d/2;    ## x-coord. of current point
$y = $yc - 60 - $dy;      ## y-coord. of init. current point
$x1 = $x + $dx;           ## x-coord. of 1. control point
$x2 = $x1 + $d;           ## x-coord. of 2. control point
$x3 = $x2 + $dx;          ## x-coord. of end point
$y1 = $y + $dy;           ## init. y-coord. of both ctrl pts

foreach $s (0, 25, 25) {

     $x1 -= $s;
     $x2 -= $s;

     $g->setLinewidth (3);
     $g->setColors ('s', 'black');

     $g->moveto ($x, $y);
     $g->curveto ($x1, $y1, $x2, $y1, $x3, $y);
     $g->stroke ();
```

```
        $g->setLinewidth (1);
        $g->setColors ('s', 'red');

        $g->moveto ($x, $y);
        $g->lineto ($x1, $y1);
        $g->moveto ($x3, $y);
        $g->lineto ($x2, $y1);
        $g->stroke;

        $g->beginText ('Helvetica-Bold', 12);
        $g->textCenter ($x, $y-$tdl, 'Curr. Pt.');
        $g->textCenter ($x1, $y1+$tdu, 'P1');
        $g->textCenter ($x2, $y1+$tdu, 'P2');
        $g->textCenter ($x3, $y-$tdl, 'P3');
        $g->endText ();

        $y  -= $d + 40;
        $y1 -= $d + 40;
        $y2 -= $d + 40;
        $y3 -= $d + 40;

};

$g->endGraphic ();

$pdf->pdfClose ();
#####################  E N D  ##############################
```

In the t6.pl sample program the foreach statement controls the loop. Three curves are to be drawn. The first is a symmetric curve. The shift parameter, $s, has the value zero. It is drawn with the x-coordinates as they are set up before the loop statement is ever entered. In the second and third loops the x-values of the control points P1 and P2 are each shifted by 25 user units to the left. The variable $s, set by the foreach statement, specifies the amount of the shift.

The program clearly demonstrates the setting of the environment (here the stroking color and line width) before each section is drawn by a **stroke** function and thus being put into the PDF. It also demonstrates that the stroking color (set by the **setColors** function with the “s” parameter) does not influence the drawing of text. Its color is determined by the filling color (set with the “f” parameter). Here it remains the default: black.

3.7 Scaled and Boxed Text String

This sample program demonstrates the use of a measuring function for obtaining the length of a text string when rendered in a particular environment, i.e. font and horizontal scale factor.

```
##### Test t7.pl  --  Scaled and Boxed Text String  ########
BEGIN {
     $PDF_LIB = $ENV{"$PDF_LIB"};
     unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t7.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 24);
$g->textCenter ($x, $y, 't7.pl:  Scaled and Boxed Text');
$g->endText ();

$p = 18;
$d = 80;
$y -= 100;

$s = 'Text horizontally normal';
$g->beginText ($p);
$g->textCenter ($x, $y, $s);
$g->endText ();

$y -= $d;

$s = 'Text horizontally compressed';

$g->beginText ();
$g->setTextHorizScale (60);
$g->textCenter ($x, $y, $s);
$g->endText ();
$w = $g->textWidth ($s);
$llx = $x - $w/2;
$lly = $y - 0.2*$p;
$urx = $llx + $w;
$ury = $lly + $p;
```

```
$g->rectangle ($llx, $lly, $urx-$llx, $ury-$lly);
$g->stroke ();

$g->setColors ('f', 'yellow');

$d = 100;
$lly -= $d;
$ury -= $d;

$s = 'Text horizontally expanded';
$g->setTextHorizScale (130);
$w = $g->textWidth ($s);
$llx = $x - $w/2;
$urx = $llx + $w;

$g->rectangle ($llx, $lly, $urx-$llx, $ury-$lly);
$g->fill ();

$g->setColors ('f', 'black');

$y -= $d;

$g->beginText ();
$g->textCenter ($x, $y, $s);
$g->endText ();

$g->endGraphic ();

$pdf->pdfClose ();
#####################  E N D  ##############################
```

In this sample program, t7.pl, we show how a text string is enclosed by a drawn box and how it is rendered on top of a colored background. The heart of the operation is the use of the **textWidth** function. It returns the horizontal extent of the text string in the currently active environment.

The execution of the two tasks is straight forward. For convenience, font point size and text string are put into variables ($p and $s). Both are needed more than once.

Note that the parameter of the **setTextHorizScale** function is not the true scale factor (for example, 1.3 if 30% expansion is desired) but it is the percentage of the expansion (in the example, 130). Unfortunately, 1.3 is also a correct parameter value, resulting in a compression of 0.013%.

The point size is used for the vertical extension of the box. But what remains to be determined is the vertical position of the box in relation to the printed text string. Because of the lack of additional font parameters, we use trial and error for the fraction of the point

size we want to see below the baseline of the printed string. In this case it suffices to position the lower edge of the box 20% of the point size below the center of the text (0.2*$p) and the lower extender of the lowercase p falls still within the box.

3.8 Circles and Circular Arcs

In this sample program we use a half circle to generate a circle. In the first part of the program, circular arcs of various angles are generated. Each is the result of only one curveto invocation, i.e., of one Bezier segment. Note that this suffices for a half circle. In the second part we demonstrate the generation of half circles of varying radius.

```
##### Test t8.pl  --  Circular Arcs and Circles  ###########
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t8.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y,
               't8.pl:  Circular Arcs (one Bezier segment each)');
$g->endText ();

$ax = 4.25 * 72;
$ay = $y - 120;

foreach $w (25, 35, 45, 90) {

     @ar = &arc (100, $w);

     $g->setColors ('s', 'black');
     $g->setLinewidth (2);
     $g->moveto ($ax+$ar[0], $ay+$ar[1]);
     $g->curveto ($ax+$ar[2], $ay+$ar[3], $ax+$ar[4],
```

```
            $ay+$ar[5], $ax+$ar[6], $ay+$ar[7]);
      $g->stroke ();

      $g->setColors ('s', 'red');
      $g->setLinewidth (0.5);
      $g->moveto ($ax+$ar[0], $ay+$ar[1]);
      $g->lineto ($ax, $ay);
      $g->lineto ($ax+$ar[6], $ay+$ar[7]);
      $g->stroke ();

      $ay -= 130;
};

$g->endGraphic ();

$pdf->newPage ();            ##### init. page

$ax = 4.25 * 72;
$ay = $y - 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y,
                't8.pl:  Circles (from two Bezier half circles)');
$g->endText ();

$d = 20;

foreach $r (25, 50, 100) {

      $ay -= $r;

      @ci = &arc ($r, 90);     ##### half circle by Bezier

      $c0 = $ax+$ci[0];        ##### adjust coordinates
      $c1 = $ay+$ci[1];
      $c2 = $ax+$ci[2];
      $c3 = $ay+$ci[3];
      $c4 = $ax+$ci[4];
      $c5 = $ay+$ci[5];
      $c6 = $ax+$ci[6];
      $c7 = $ay+$ci[7];

      $g->setLinewidth (2);
      $g->setColors ('s', 'black');
```

```
                        ##### move to and draw curve
      $g->moveto ($c0, $c1);
      $g->curveto ($c2, $c3, $c4, $c5, $c6, $c7);
      $g->stroke ();

      $c0 = $ax-$ci[0];        ##### adjust coordinates
      $c1 = $ay-$ci[1];
      $c2 = $ax-$ci[2];
      $c3 = $ay-$ci[3];
      $c4 = $ax-$ci[4];
      $c5 = $ay-$ci[5];
      $c6 = $ax-$ci[6];
      $c7 = $ay-$ci[7];

                              ##### move to and draw curve
      $g->moveto ($c0, $c1);
      $g->curveto ($c2, $c3, $c4, $c5, $c6, $c7);
      $g->stroke ();

      $y1 = $c1;               ##### set for red line
      $x1 = $ax - $r - $d;
      $x2 = $x1 + 2*($r+$d);

      $g->setLinewidth (0.5);
      $g->setColors (’s’, ’red’);

      $g->moveto ($x1, $y1);   ##### draw red line
      $g->lineto ($x2, $y1);
      $g->stroke ();

      $ay -= $r + 50;          ##### adjust vert. pos.
};

$g->endGraphic ();

$pdf->pdfClose ();

sub arc {
      $r = $_[0];
      $a = (3.141593 / 180) * $_[1];

      $l = ( $r * (4/3) * (1 - cos ($a)) ) / sin ($a);
      $x3 = $r * cos (2*$a);
      $y3 = $r * sin (2*$a);
      $x2 = &chop ($x3 + $l * sin (2*$a));
      $y2 = &chop ($y3 - $l * cos (2*$a));
      $x3 = &chop ($x3);
```

```
      $y3 = &chop ($y3);
      $x1 = &chop ($r);
      $y1 = &chop ($l);
      $x0 = $r;
      $y0 = 0;
      @ret = ($x0, $y0, $x1, $y1, $x2, $y2, $x3, $y3);
      return @ret;
}

sub  chop {
      return  ( int ($_[0] * 1000) ) / 1000;
}
####################  E N D  ##############################
```

The internal subroutine arc is the basis of all Bezier curves in the sample program. It generates part of a circular arc of given radius and given semi-angle. The arc always starts from the x-axis upwards, with the tangent being perpendicular to the x-axis. It returns an array of eight numbers (four points), which are the origin, the first and second control points and the end point of the curve.

As the results show even a half circle is still reasonably well approximated by a single segment of a Bezier curve.

On the second page we use the arc routine for obtaining a half circle, or more precisely, for obtaining the four points defining a Bezier curve approximating a half circle. The curve itself is then drawn by the moveto and curveto invocations with these four points as parameters. The second half circle we generate by using the obvious symmetries for the control points.

During our computations it became apparent that some routines of the PDF-PL expect integers or decimal numbers as input parameters. Scientific notation (as needed for sufficiently small or large numbers) is not accepted. In order to avoid this problem, we use the internal routine chop. It chops our computational results to three decimals on the right side of the decimal point, before they are submitted as parameters to the PDF-PL routines.

3.9 The Transformation Matrix

In this sample program it is demonstrated that generally the sequence of two transformations cannot be reversed without changing the result.

```
##### Test t9.pl  --  The Transformation Matrix  ############
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;
```

```
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t9.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y, 't9.pl:  The Transformation Matrix');
$g->endText ();

$x = 72;
$y -= 100;

$g->setColors ('f', 'red');
$g->beginText ('Times-Roman', 24);
$g->textLeft ($x, $y, 'Sample Text BEFORE transformation!');
$g->endText ();

$g->gsave ();

$g->ctm (1, 0, 0, 1, 180, 0);
$g->ctm (0.7, 0, 0, 1, 0, 0);

$y -= 100;

$g->setColors ('f', 'blue');
$g->beginText ('Times-Roman', 24);
$g->textLeft ($x, $y,
              'Sample Text AFTER translation and scaling!');
$g->endText ();

$g->grestore ();

$g->ctm (0.7, 0, 0, 1, 0, 0);
$g->ctm (1, 0, 0, 1, 180, 0);

$y -= 100;

$g->beginText ();
$g->textLeft ($x, $y,
```

```
              'Sample Text AFTER scaling and translation!');
$g->endText ();

$g->endGraphic ();

$pdf->pdfClose ();
####################  E N D  ##############################
```

In the sample program, t9.pl, we print three lines of text. The first is printed in the original environment: red fill color and no coordinate transformations yet. We save this environment (gsave).

Now we change to the fill color blue and transform the coordinate system by two successive transformations: first a translation along the x-axis by 250 user units, then a scaling in x-direction by a factor of 0.7. Remember the original coordinate origin is the left lower corner of the page!

After the transformation the blue text is right shifted (the translation) by 250 user units and somewhat compressed (the scaling). (We changed the y-coordinate so that the text strings are not printed on top of each other. But this coordinate is not influenced at all by the transformations.)

Now we restore the saved environment (**grestore**). Then we apply the same two transformations again, but in reverse order: first the translation and then the scaling. The printed text appears in blue and is compressed similarly in x-direction as before, but it is shifted less to the right than before.

The reason for the difference lies in the fact that the translation in the first case (translation followed by scaling) remains 250 user units. Together with the x- coordinate of the text positioning this results in a given starting point for the string. But in the second case the x-axis is first scaled by 0.7, i.e. each coordinate unit is 30% shorter. The translation by 250 units is now worth only 70% of the previous translation, i.e. 175 user units. Remember, user units remain the units of measure in the original coordinate system. Each x-unit in the coordinate system after the transformation is now worth only 70% of a user unit. This is a consequence of the philosophy of transformations in the PDF-PL, caused--of course--by the philosophy in the PDF: We transform the coordinate system to the new environment (we translate it, rotate it, scale it, skew it). But the object remains the same in terms of the number of units it is specified in. Of course, the units may have changed their size by the transformation.

3.10 Clipping Path

This program demonstrates how a path, here a rotated square, clips a pattern of lines.

```
##### Test t10.pl  --  Clipping  ############################
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
```

```
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t10.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y, 't10.pl:  Clipping');
$g->endText ();

$g->ctm (1, 0, 0, 1, $x, 6*72); #### shift to center of page

$d = 100;
$g->setColors ('s', 'red');
$g->setLinewidth (8);

$g->moveto (0, $d);            ##### start clipping path
$g->lineto (-$d, 0);
$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();

$g->clip ();

$g->stroke ();

$g->setLinewidth (6);
$g->setColors ('s', 'blue');
$x = -$d;
$sl = $d/5;
$dy = $d/5;

for ($y = -$d-$sl; $y<$d+$sl; $y +=$dy) {
     $g->moveto ($x, $y);
     $g->lineto ($x+2*$d, $y+$sl);
     $g->stroke ();
};
```

```
$g->endGraphic ();

$pdf->pdfClose ();
#####################  E N D  ##############################
```

In the sample program t10.pl some aspects of clipping by a user specified path are demonstrated.

First, i.e., after printing the title of the sample page, the coordinate system is shifted to the desired center of the clipping path, approximately to the center of the page. This is done for user convenience: the involved coordinates become easier to compute.

Then the clipping path is defined. It starts with setting up the environment: line width of 8 points and stroking color red. Then the path along the rotated square is defined by the moveto and lineto functions, using $d as the length of the half diagonal. Note that only three lines are explicitly defined while the last is defined by the closepath function. Now, before the stroking occurs the clip function is invoked, defining the path as a clipping path. Up to this time, a square is drawn in red color with a line width of 8 point and standing on one corner.

Now we must generate the pattern which is to be clipped by the currently active clipping path.

We use a for loop to draw slightly skewed horizontal lines of linewidth 6 points and yellow color. Care is taken to make sure that the lines cover the entire square.

Note that the yellow lines partially overlap the red square line. The clipping takes place exactly at the clipping path which by definition is the mathematical line running along the center of the red lines. The exact order of the rendering of color onto the page is this:

1. Fill color of the clipping path

2. Stroked lines color of the clipping path

3. Fill color of the clipped pattern

4. Stroked lines color of the clipped pattern

Not all four of these cases may actually exist. In our case only numbers 2 and 4 do.

It should be noted that all that is necessary is to remove the **clip** function from the program in order to obtain the successive drawing of the clip pattern followed by the clipped pattern **without** any clipping taking place. Of course, the above stated order of color rendering on the output medium is unchanged.

3.11 Text as Clipping Path

This sample program demonstrates how some text characters clip a pattern of lines.

```
##### Test t11.pl  --  Text as Clipping Path  ##############
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t11.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y, 't11.pl:  Text as Clipping Path');
$g->endText ();

$y = 5*72;

$g->setTextRender (5);

$g->beginText ('Times-Roman', $p=216);
$g->textCenter ($x, $y, $s='Clip');
$g->endText ();

$w = $g->textWidth ($s);

$g->setLinewidth (1);
$x = $x - $w/2;
$y0 = $y;
$dy = 5;

for ($y=$y0-0.3*$p; $y<$y0+1.1*$p;  $y+=$dy ) {
     $g->moveto ($x, $y);
     $g->lineto ($x+$w, $y);
     $g->stroke ();
```

```
};

$g->endGraphic ();

$pdf->pdfClose ();

#####################  E N D  ##############################
```

In sample program t11.pl we show the word Clip drawn in outline and filled with a horizontal line pattern. Again, first the clipping path (which here is the succession of all character outlines of the word Clip) is defined and drawn. This happens in the first part of the program (the second TMA!). The rendering mode 5 (outline the characters and define them as clipping paths) is applied to the text string Clip.

Now the lines to be clipped are drawn. For this purpose we imagine a text string box similar to the one in sample program t7.pl. We compute the width (length) of the text string in 144 point Times-Roman font. Again we estimate that 20% of the point size is enough to cover the descenders of the font. Thus we assure that we cover the text characters completely. The horizontal lines are drawn in a straight forward manner by the triplets (moveto lineto stroke).

3.12 Text Blocks and Text Parameters

In this sample program we demonstrate some functions which permit printing on the output medium not just one text line at a time but an entire set of lines. Furthermore, calling these functions also requires specification of some of the text environment variables. The influence of some of the other major text variables is also shown.

```
##### Test t12.pl  --  Text Blocks and Text Parameters  ####
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t12.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 10 * 72;

$g = $pdf->beginGraphic ();
```

```
$g->beginText (’Helvetica’, 18);

                          $g->textCenter ($x, $y, ’Text Block and
     Parameters’);
$g->endText ();

$y = 9*72;

@tb = (’We use this same block for testing’,
       ’the different text parameters:’,
       ’character spacing, word spacing,’,
       ’leading,.....’);

$dy = 110;
$xleft = 72;
$xcenter = 4.25 * 72;
$xright = 7.25 * 72;
$fn = ’Helvetica’;
$fs = 12;
$l = 14;

$g->textBlockLeft ($xleft, $y, $fn, $fs, $l, @tb);

$y -= $dy;
$g->setCharSpace (8);

$g->textBlockRight ($xright, $y, $fn, $fs, $l, @tb);

$y -= $dy;
$g->setCharSpace (0);
$g->setTextWordSpace (8);

$g->textBlockCenter ($xcenter, $y, $fn, $fs, $l, @tb);

$y -= $dy;
$g->setTextWordSpace (0);
$l = 20;

$g->textBlockLeft ($xleft, $y, $fn, $fs, $l, @tb);

$y -= $dy;
$l = 14;
$g->setTextHorizScale (70);

$g->textBlockLeft ($xleft, $y, $fn, $fs, $l, @tb);
```

```
$y -= $dy;
$g->setTextHorizScale (130);

$g->textBlockLeft ($xleft, $y, $fn, $fs, $l, @tb);

$g->setTextHorizScale (100);

$g->endGraphic ();

$pdf->newPage ();

$g = $pdf->beginGraphic ();

@offset = (1.0, 0.75, 0.5, 0.25, 0);

$g->beginText ('Helvetica', 18);
          $g->textCenter (4.25*72, 10*72,
                    'textBlockOffset ({@offset}, ....)');
$g->endText ();

$y = 8.5 * 72;

foreach $offs (@offset) {

     $g->textBlockOffset ($offs, $xcenter, $y, $fn, $fs, $l, @tb);
     $y -= $dy;
}

$g->endGraphic ();

$pdf->pdfClose ();
#####################  E N D  ##############################
```

In the sample program t12.pl the same array of text lines is used to demonstrate several text block rendering functions and several text parameter setting functions. We use a standard value of 14 point for the leading parameter. The PDF default is 0, which means no leading at all. All consecutive lines are printed at the same vertical position.

First the text block is printed using the **textBlockLeft** function. This function prints the lines left adjusted starting with the first line at the specified (x, y) position. Except for font name and font size, all text parameters are at their default values.

Then character spacing is set to 10 points and the text block is printed right adjusted, i.e. using the **textBlockRight** function.

After resetting the character space to its default value, word spacing is set to 10 points. Now the text block is to be centered, which is done by using the **textBlockCenter** function.

After resetting the word space to its default value, the leading is set to 20 points. The lines are now vertically space by 6 points more than in our normal case (14 points).

After resetting the leading to our standard value of 14 points the text block is printed left adjusted first with a horizontal compression of 30% and then with a horizontal expansion of 30%.

After resetting the horizontal scaling the text block is printed five times on the next page demonstrating the **textBlockOffset** function. As horizontal reference point the center of each line is used. The offset varies from 1 over .75, .50, and .25 to 0. It becomes apparent, that the use of the first, third and fifth of these values is equivalent to invocations of the functions **textBlockLeft**, **textBlockCenter** and **textBlockRight** respectively.

3.13 Colorspaces

This sample program demonstrates the use of some of the color spaces that are available in the PDF.

```
##### Test t13.pl  --  Set Color Spaces  ###################
BEGIN {
    $PDF_LIB = $ENV{"$PDF_LIB"};
    unshift (@INC, $PDF_LIB);
}

use Pdf::File;
use Pdf::Font;
use Pdf::Graphic;

my $pdf = Pdf::File->new();
$pdf->pdfOpen ('./t13.pdf');

$pdf->newPage ();            ##### init. page

$x = 4.25 * 72;
$y = 9 * 72;

$g = $pdf->beginGraphic ();

$g->beginText ('Helvetica', 18);
$g->textCenter ($x, $y, 't13.pl:  Set Color Spaces');
$g->endText ();

$tx = 50;
```

```
$d = 30;

$g->ctm (1, 0, 0, 1, 3*72, 8*72); #### shift to center page

$g->setColors ('f', 'DeviceRGB', 0.2, 0.3, 0.7);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);
$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText (15);
$g->textLeft ($tx, 0, 'DeviceRGB  0.2  0.3  0.7');
$g->endText ();

$g->ctm (1, 0, 0, 1, 0, -72);  ##### shift down one inch

$g->setColors ('f', 0.7, 0.2, 0.2);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);
$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText ();
$g->textLeft ($tx, 0, 'DeviceRGB  0.7  0.6  0.2');
$g->endText ();

$g->ctm (1, 0, 0, 1, 0, -72);  ##### shift down one inch

$g->setColors ('f', 'DeviceCMYK', 0.8, 0.3, 0.2, 0.1);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);

$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText ();
$g->textLeft ($tx, 0, 'DeviceCMYK  0.8  0.3  0.2  0.1');
$g->endText ();
```

```
$g->ctm (1, 0, 0, 1, 0, -72);  ##### shift down one inch

$g->setColors ('f', 0.2, 0.3, 0.9, 0.1);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);

$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText ();
$g->textLeft ($tx, 0, 'DeviceCMYK  0.2  0.3  0.9  0.1');
$g->endText ();

$g->ctm (1, 0, 0, 1, 0, -72);  ##### shift down one inch

$g->setColors ('f', 'DeviceGray',  0.8);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);
$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText ();
$g->textLeft ($tx, 0, 'DeviceGRAY  0.8');
$g->endText ();

$g->ctm (1, 0, 0, 1, 0, -72);  ##### shift down one inch

$g->setColors ('f', 0.3);

$g->moveto (0, $d);            ##### start path
$g->lineto (-$d, 0);
$g->lineto (0, -$d);
$g->lineto ($d, 0);
$g->closepath ();
$g->fill ();

$g->beginText ();
$g->textLeft ($tx, 0, 'DeviceGRAY  0.3');
$g->endText ();
```

```
$g->endGraphic ();

$pdf->pdfClose ();
####################  E N D  ##############################
```

The sample program t13.pl shows two examples each of the three color spaces **DeviceRGB**, **DeviceCYMK** and **DeviceGray**. For each of the six examples the ctm function is used to transform the coordinate system into the center of a square filled with color. Then the rotated square is drawn easily.

The name of the color space need be specified only once. Then it is enough to specify only another set of color components in order to change the color. Note, that, aside from the first parameter (fill or stroke), a new color (set of color components between 0 and 1) must always be specified, regardless whether a new color space has been specified or not.

The single number required for the **DeviceGray** color space is 1 for white and 0 for black. This to me is counter intuitive!

Note that in the program currently several replacement of the **setColors** function by the **forceColors** function have been done in order to overcome a current bug in the PDF-PL.

4.0 The Function Reference List

This list contains those functions of the PDF-PL that I currently understand. One can open the PDF-PL to the PDF builder to a deeper or less deep level. This current list is intended for a top level user, the typical application oriented person who is not interested in the intricacies of internal implementation but only in correct and successful use of the PDF-PL to build error free PDFs.

It should be noted that most of the environment setting functions, set..., have a corresponding force... function. Such force... function does exactly the same as the set... function except that it forces immediate committing of the corresponding PDF operator (or operators) into the PDF stream. In the case of a set... function the PDF operator is not committed to the stream until it is necessary: it is *cached*.

The PDF-PL library is implemented using Perl5 which provides the concepts of Object Oriented Programming. Therefore we list the user PDF-PL functions (methods) separated into groups for each of the applicable object classes: File and Graphic.

4.1 “File” Methods

These are the methods of the object class File which are visible to the user (here the PDF builder).

info (“title”, t, “author”, a, “creator”, c, “subject”, s)

Supplies values for several keywords to be saved by the internal routines in the new PDF.

May be invoked anywhere between the **pdfOpen** and **pdfClose** functions, but prudent programming practice suggests that it be placed at the start or the end of the PDF generating code.

If there are more than one occurrences of **info** invocation then only the last one for any keyword will be properly saved by the internal routines.

Parameters:

Must be an even number of character strings. Each pair represents a key followed by a key value. The parameter pairs may appear in any order and any of the pairs may be omitted.

newPage (*pn***)**

Creates a new page.

Parameters:

pn is the page number of the new page. This page must not yet have been generated.

The parameter is optional.

Default:

If there is no page number specified the new page number is the next after the most recently created one.

Note that there must be **no** gaps in the sequence of page numbers starting with one.

pdfClose ()

Does whatever is necessary to end the PDF body, mainly generating those PDF objects that are still missing in the output file. Then generates the PDF Trailer and XREF Table and writes them to the output file and closes it.

pdfOpen (*fn***)**

Opens file fileName as the output file, generates the PDF Header, and does whatever is necessary to start the body of the PDF.

Parameters:

fn is either a filename string or a reference to a filehandle for which the file is already open. In the latter case opening of the file does not occur.

The parameter is required.

setPageSize (*w, h***)**

Sets the default size of the pages. This invocation may appear almost anywhere in the PDF builder's program, but prudent programming practices suggest that it be placed at the start.

Parameters:

w is the page width, *h* the page height. They are to be specified in user units.

The parameters are required.

Default:

8.5 inch by 11 inches. This is the default page size if there is no invocation of **setPageSize** at all in the user's PDF building Perl5 program.

4.2 "Graphic" Methods

These are the methods of the object class Graphic which are visible to the user (here the PDF builder).

addGraphic (*pn***)**

Adds everything that is marked by that GMA (or more precisely, that PDF stream) to a specified page. That GMA was specified by the Graphic object instance on which the method, addGraphic, is invoked.

Parameters:

pn is the number of the affected page (see above).

The parameter is required.

beginGraphic (*pn***)**

Initializes the beginning of a new GMA. The PDF builder is expected to begin now to invoke functions which set the environmental variables for the GMA in the way he/she desires as well as to draw graphic entities (lines, curves, etc.) he/she wants to render on the current page. A list of the available functions can be found in the sections *The Marking of the Page* and *The Current Point and Paths*.

A GMA must be ended by invoking the **endGraphic** function.

More than one GMA may be started and ended any page.

If the page number parameter is specified it designates the page to which this new GMA is to be appended.

If a new GMA is started the complete set of environmental variables is reset to the default values. Exceptions are the font name and font size. The PDF-PL routines expect that at the beginning of a TMA these two text environment variables are defined. However, once

defined within a GMA, they propagate up to the **endGraphic** function invocation. Hence, it is necessary to specify font name and font size only at the beginning of the first TMA within the current GMA. Later TMAs may change font name and/or font size if the builder so desires.

Parameters:

pn is optional. If it is specified it is the number of the page to which the newly started GMA is appended. This page must already exist.

Returns:

A reference to the new GMA. This reference is used to qualify the invocation of all graphic marking, text marking and environment changing functions within this GMA. This is demonstrated by all sample programs.

beginText (*fn***,** *fs***)**

Signals the beginning of text to be put on the page and may specify the name and size of the font to be used.

Parameters:

fn is the name of the font, for example ‘Helvetica’ or ‘Helvetica-Bold’ and *fs* is the size of the font in printers points. Other than that it must be a positive number there is no restriction on the size of the font.

Both parameters are optional. However, if one or both are not specified, the PDF builder must make sure that at the point of this function invocation both are defined in the current GMA. This means, at least once since the beginning of the current GMA, each of them must have been specified on a **beginText**, **setFont** or one of the text block functions.

Permissible font names in the current PDF-PL package are:

- Courier
- Helvetica
- Helvetica-Bold
- Helvetica-Oblique
- Helvetica-BoldOblique
- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- Symbol
- ZapfDingbat

Default:

None. The PDF builder must at least once set the font name and font size before text can be put on the page.

clip ()

Makes the current path a clipping path.

This function invocation must occur within the definition of a closed path. More precisely, it must occur after the definition of the last clipping segment and the path painting operator.

In order to determine the actual clipping area, this operator uses the nonzero winding rule.

closepath ()

Closes the currently open path in the current GMA by appending a straight line connecting the current point with the beginning of the path.

closepathFillStroke ()

Closes the currently open path in the current GMA by appending a straight line connecting the current point with the beginning of the path. Then strokes and fills the path using the nonzero winding rule.

This is a succession of the closepath function followed by the fillStroke function.

ctm (*p1***,** *p2***,** *p3***,** *p4***,** *tx***,** *ty***)**

ctm stands for *current transformation matrix*. It transforms the current coordinate system into a new one thereby changing the appearance of the text and graphics on the output medium. The transformations of translation, rotation, scaling, and skewing are possible.

Parameters:

tx and *ty* are the amounts of translation in x- and y- direction, respectively. The origin of the new coordinate system is the point (*tx*, *ty*) in the old coordinate system.

p1, *p2*, *p3*, and *p4* specify rotation, scaling and skewing.

A pure rotation in the x-y-plane by the angle A is specified by

```
p1 = cos A
p2 = sin A
p3 = -sin A
p4 = cos A
tx = ty = 0
```

Rotation and translation can be combined (and their sequence does not matter) by also specifying the amount of translation in the x- and y-directions, *tx* and *ty*.

Pure scaling is specified by

```
p1 = scale factor in x-direction
p4 = scale factor in y-direction
p2 = p3 = tx = ty = 0
```

This transforms the coordinates so that one unit in x- direction of the new coordinate system is the same size as *p1* units in the previous coordinate system, and similarly for the y-direction. In other words if *p1* is greater than one the text or graphics to be rendered are now greater by the factor *p1* in x-direction than they were in the old coordinate system. Similarly, if *p1* is smaller than one, the rendered text or graphics will become smaller.

Pure skewing is accomplished by the following choice of parameters

```
p2 = tan A
p3 = tan B
p1 = p4 = 1
tx = ty = 0
```

This skews the x-axis of the new coordinate system by the angle A (measured clockwise from the x-direction) and the y- axis by the angle B (measured counter clockwise from the y- direction).

Transformations may be cascaded, i.e., one may be executed after another. However, it must be noted that the results may be different, if the sequence of the transformations is changed. As stated before, rotation and translation are independent. Their sequence does not matter. But, for example scaling the x-direction followed by a translation is not the same as first translating the x-axis (sliding it in is own direction) and then performing the scaling. See the sample program, t9.pl, *The Transformation Matrix*.

Default:

No transformation.

curveto (*x1, y1, x2, y2, x3, y3*)

Adds a Bezier curve to the current path definition and moves the current point.

Parameters:

Points (*x1*, *y1*) and (*x2*, *y2*) are the control points. Point (*x3*, *y3*) is the end point of the curve. The point coordinates are in user units. All six parameters are required. The point (*x3, y3*) is the new current point.

endpath ()

Ends the current path without closing it.

endGraphic ()

Ends the range of a new GMA. This function must be invoked before a new page may be started again.

endText ()

Ends a TMA for positioning text on the page. This function must be invoked before graphic marking functions may be invoked again within the encompassing GMA.

eoclip ()

Makes the current path a clipping path.

This function invocation must occur within the definition of a closed path. More precisely, it must occur after the definition of the last clipping segment and the path painting operator.

In order to determine the actual clipping area, this operator uses the even-odd rule.

evenOddFill ()

Fills the most recently ended path following the even-odd rule to determine which regions of the path to fill with the current filling color. For an explanation of the rule see the section *Path Painting Operators* in the PDF Manual.

evenOddFillStroke ()

Fills and strokes the most recently ended path following the even-odd rule to determine which regions of the path to fill with the current filling color. For an explanation of the rule see the section *Path Painting Operators* in the PDF Manual.

This is a succession of the **evenOddFill** function followed by the **stroke** function.

fill ()

Fills the most recently ended path following the nonzero winding number rule to determine which regions of the path to fill with the current filling color. For an explanation of the rule see the section *Path Painting Operators* in the PDF Manual.

fillStroke ()

Fills and strokes the most recently ended path following the nonzero winding number rule to determine which regions of the path to fill with the current filling color. For an explanation of the rule see the section *Path Painting Operators* in the PDF Manual.

This is a succession of the fill function followed by the stroke function.

grestore ()

Restores the most recently saved environment. This function must be invoked within a GMA, but not within an enclosed TMA. See the gsave function for a list of the restored environment variables.

gsave ()

Saves the current environment. This function must be invoked within a GMA, but not within an enclosed TMA.

The environment variables saved are:

- color for stroking and filling
- line width
- line cap style
- line dash pattern
- line join style
- miter limit
- font name
- font size
- current point
- current clipping paths

See the grestore function for restoring the environment.

lineto (*x*, *y*)

Draws a straight line from the current point to the point (x, y).

Parameters:

The point (*x*, *y*) is the new current point. Both point coordinates are in user units.

The parameters are required.

moveto (*x*, *y*)

Moves the current point.

Parameters:

The point (*x*, *y*) is the new current point. Both point coordinates are in user units.

The parameters are required.

The default current point at the beginning of a new GMA is (0, 0), i.e. the left lower corner of the page.

rectangle (*x*, *y*, *w*, *h*)

Adds a rectangle to the current path definition.

Parameters:

(*x, y*) is the left lower point of the rectangle. w is its width and h its height. All four parameters must be user units. The point (*x, y*) becomes the new current point.

All four parameters are required.

setCharSpace (*c*), forceCharSpace (*c*)

sets the intercharacter space in a text string.

Parameters:

c is the character-space in user units to be added between two immediatly adjacent characters within a text string.

The parameter is required.

Default: 0

setColors (*sf, c*), forceColors (*sf, c*, s*1, s2, s3*)

Used within a GMA to set a new color for drawing lines and/or filling paths. Drawing applies to paths, lines and boxes. Filling applies to closed paths, boxes and the rendering of text strings.

Parameters:

sf must be one of the strings ‘s’, ‘f’, ‘sf’ or ‘fs’. An ‘s’ indicates that color is the new stroking color and an ëfí that it is also the new filling color.

c may be one of the colors predefined in the PDF-PL:

- black
- blue
- darkblue
- gray
- green
- red
- white
- yellow

If *c* is not one of the predefined colors is must specify a color space:

- ‘DeviceRGB’
- ‘DeviceCYMK’

- ‘DeviceGray’

In this case *c* must be followed by a color specification. For the color space DeviceRGB three components are required which specify the mix of red, green and blue. These three numbers, *s1*, *s2*, *s3*, must be between 0 and 1 each. For DeviceCYMK there must be four component values for cyan, magenta, yellow and black, again each between 0 and (1);. For DeviceGray there need be only one parameter, again between 0 and 1.

The first parameter is required. The others must be specified as stated.

Default:

Both fill and stroke default colors are black.

setDashPattern (*a, p*), forceDashPattern (*a, p*)

Specifies that all lines drawn are to be of a particular kind. For details and examples consult the section *Line Dash Pattern* in the PDF Manual or see the text following the second sample program *Line Samples*.

It should be noted that the drawing of the dashes otherwise follows the currently valid environmental variables. The width is determined by the current line width. Ends of dashes are treated with the current line cap style. Corners within dashes are treated with the current line join style.

Parameters:

a must be an array of numbers. These numbers specify, in user units, the length of successive strokes and gaps along a stroked path or line. If this number is odd, the last specified dash length is also the length of the last gap.

p must be a single number. It specifies the phase with which the dash pattern starts the path, measured in user units.

The parameters are required.

Default:

The default dash pattern is a solid line which can be specified by an empty array and a phase value of 0.

setFlatness (*f*), forceFlatness (*f*)

Sets a limit for the maximum permitted distance between a mathematical correct path and an approximation constructed from straight line segments. It is device dependent since it is measured in device pixels.

Parameters:

f is the distance in device pixels which is still tolerable. It must be 0 and 100 inclusive.

The parameter is required.

Default:

0, which means that the device flatness is to be used.

setFont (*fn*, *fs*), forceFont (*fn*, *fs*)

Sets a new font name and/or font size. This permits changing the font parameters between text invocations within a TMA.

Parameters:

fn is the name of the font, for example ‘Helvetica’ or ‘Helvetica-Bold’ and *fs* is the size of the font in printers points. Other than that it must be a positive number there is no restriction on the size.

At least one of the parameters must be specified. However, if one is not specified, the PDF builder must make sure that at the point of this function invocation it is defined in the current GMA. This means, at least once since the beginning of the current GMA, it must have been specified on a **beginText** function.

Permissible font names in the current PDF-PL package are:

- Courier
- Helvetica
- Helvetica-Bold
- Helvetica-Oblique
- Helvetica-BoldOblique
- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- Symbol
- ZapfDingbat

Default:

None. The PDF builder must at least once set the font name and font size before text can be output on the page.

setJoinStyle (*s*), forceJoinStyle (*s*)

Sets the style in which two drawn lines are joined. For details and examples consult the section *Line Join Style* in the PDF Manual.

Parameters:

s must be 0, 1 or 2. These values specify miter joins, round joins or bevel joins respectively.

The parameter is required.

Default:

0, i.e. miter join.

setLineCapStyle (*c*), forceLineCapStyle (*c*)

Sets the style in which a drawn line is terminated. For details and examples consult the section *Line Cap Style* in the PDF Manual.

Parameters:

c must be 0, 1 or 2. These values specify butt end caps, round end caps and square end caps resp.

The parameter is required.

Default:

0, i.e. butt end capping of the lines.

setLinewidth (*w*), forceLinewidth (*w*)

Specifies a new width for lines to be drawn (for example, by lineto or curveto).

Parameters:

w is the new line width specified in user units. The parameter is required.

setMiterLimit (*m*), forceMiterLimit (*m*)

When miter join style is in effect, the miter limit determines whether the miter join (miter length) extends too far away from the point of joining the two lines. Then a bevel join is substituted. For a detailed figure consult the section *Miter Limi*t in the PDF Manual.

Parameters:

m is the limiting ratio of the miter length to the line width. Its value must be greater than or equal to 1. Joins for which this value is exceeded will be converted to bevel joins.

The parameter is required.

Default: 10

setTextHorizScale (*sf*), forceTextHorizScale (*sf*)

Changes the horizontal extent of text strings by the specified amount.

Parameters:

sf is the scale factor. It is specified as a percentage without the percent sign. A value of 100 means no scaling at all. It is applied to any text string before it is rendered on the output medium.

The parameter is required.

Default:

100, i.e. no scaling at all.

setTextLeading (*l*), forceTextLeading (*l*)

Sets the vertical distance of successive text baselines in a block of text.

Watch out for this, PDF builders! Using the leading default, which is 0, results in successive lines to be printed on top of each other. To the best of my knowledge, leading is the distance between *text boxes*, or rather in the old world of leaden type slugs, the distance between the type slugs of successive text lines. However, in the PDF-PL (and in the PDF world), leading is the distance between successive text lines, commonly called line spacing. The height (y-extent) of a type slug was equal to the point size. As a consequence the line spacing is equal to the point size plus the leading.

Parameters:

l is the vertical distance of successive baselines in a block of text.

The parameter is required.

Default: 0

setTextRender (*r*), forceTextRender (*r*)

Set the rendering of text on the output medium.

Parameters:

r may have one of 8 values:

0 fill text with current fill color

1 stroke text with current stroke color

2 fill and stroke text with the respective colors

3 do nothing, text is invisible

4 fill text and add it to the clipping path

5 stroke text and add it to the clipping path

6 fill and stroke text and add it to the clipping path

7 add text to the clipping path

Default:

0, fill text with current fill color.

setTextRise (*r*), forceTextRise (*r*)

Shifts the baseline on which the text rests up or down thus permitting superscripting or subscripting.

Parameters:

r is the amount of vertical shift (rise) specified in user units. A positive parameter results in an up shift, a negative parameter value results in a down shift.

The parameter is required.

Default: 0

setTextWordSpace (*w*), forceTextWordSpace (*w*)

Sets the space between adjacent words in a text string. Note that the blank character is considered a character in the PDF. Hence a word space of zero means the actually visible word space is only the width of the blank character. Hence, the default word space can be zero without causing troubles similar to the text leading.

Parameters:

w is the word space in user units.

The parameter is required.

Default: *0*

textBlockCenter (*x*, *y*, *fn*, *fs*, *l*, *ta*)

Prints an array of text lines starting with the first line centered at the point (x, y). Subsequent lines are printed below, each one centered at (x, y) but the y position shifted downward by the leading l.

Parameters:

(*x, y*) is the center point of the first line, *fn* and *fs* are the font name and font size, *l* is the leading, *ta* is the array of text lines. The values *x, y,* and *l* must be specified in user units.

All parameters are required.

textBlockLeft (*x***,** *y***,** *fn***,** *fs***,** *l***,** *ta***)**

Prints an array of text lines starting with the first line left adjusted at the point (*x*, y). Subsequent lines are printed below, each one left adjusted at (*x, y*) but the y position shifted downward by the leading *l*.

Parameters:

(*x, y*) is the starting point of the first line, *fn* and *fs* are the font name and font size, *l* is the leading, *ta* is the array of text lines. The values *x, y,* and *l* must be specified in user units.

All parameters are required.

textBlockOffset (*o***,** *x***,** *y***,** *fn***,** *fs***,** *l***,** *ta***)**

Prints an array of text lines, each line horizontally adjusted by a specified fraction as described below.

Parameters:

o is the offset. It must be a value between 0 and 1 inclusive. Each line is positioned such that the *o* fraction of it is positioned to the left of the point (*x, y*), while *y* is vertically adjusted by the leading l. Hence, the invocation of **textBlockCenter** (*x*...) is identical to the invocation of **textBlockOffse**t (0.5, *x*....).

fn and *fs* are the font name and font size, *l* is the leading. The values *x, y,* and *l* must be specified in user units. ta is the array of text lines.

All parameters are required.

textBlockRight (*x***,** *y***,** *fn***,** *fs***,** *l***,** *ta***)**

Prints an array of text lines starting with the first line right adjusted at the point (*x, y*). Subsequent lines are printed below, each one right adjusted at (*x, y*) but the *y* position shifted downward by the leading *l*.

Parameters:

(*x, y*) is the starting point of the first line, *fn* and *fs* are the font name and font size, *l* is the leading. The values *x, y,* and *l* must be specified in user units. *ta* is the array of text lines.

All parameters are required.

textCenter (*x***,** *y***,** *t***)**

Positions the text string centered using current font and font size.

Parameters:

The point (*x*, *y*) specifies the position of the center of the text string. The values *x* and *y* must be in user units.

t is the text string to be printed.

The parameters are required.

textLeft (*x*, *y*, *t*)

Positions the text string left adjusted using current font and font size.

Parameters:

The point (*x*, *y*) specifies the position of the left end of the text string. The values *x* and *y* must be in user units.

t is the text string to be printed.

The parameters are required.

textRight (*x*, *y*, *t*)

Positions the text string right adjusted using current font and font size.

Parameters:

The point (*x*, *y*) specifies the position of the right end of the text string. The values *x* and *y* must be in user units.

t is the text string to be printed.

The parameters are required.

vCurveto (*x1*, *y1*, *x2*, *y2*)

Adds a Bezier curve to the current path definition and moves the current point.

Parameters:

The current point is the first control point, (*x1*, *y1*) is the second control point and (*x2*, *y2*) is the end point of the curve. The point coordinates are in user units.

All four parameters are required.

(*x2*, *y2*) is the new current point.

yCurveto (*x1, y1, x2, y2*)

Adds a Bezier curve to the current path definition and moves the current point.

Parameters:

Point (*x1, y1*) is a control point and point (*x2, y2*) is the second control point as well as the end point of the curve. The point coordinates are in user units.

All four parameters are required.

(*x2, y2*) is the new current point.

textWidth (*s*)

Measures the width (i.e. the horizontal width or the length) of a text string. Uses the font name and size from the currently active environment.

Parameters:

s is the text string for which the width is to be measured.

Returns:

The width of the text string in user units.

5.0 Known Problems

The functions **gsave** and **grestore** are not working reliably. The execution of the samples t4.pl and t9.pl (both using the **gsave** and **grestore** functions) gave rise to error messages yet the results could still be displayed and printed.

You may find that the **setDashPattern** function may not respond correctly to a default phase value (see the second dashed line on page 2 of the example in section *Line Samples)*. This is an Acrobat Exchange 2 bug which will be fixed in Exchange 3. But the PDF-PL generates a correct PDF operator and parameters.