DVIPDF and Embedded PDF

Sergey Lesenko

Institute for High Energy Physics,
Protvino, Russia
E-mail: lesenko@mx.ihep.su

Abstract. *We explain how the current version of the DVIPDF program manages to integrate external multipage PDF files into its own PDF output.*

1. Introduction

Today's electronic documents for the Web are most often distributed in HTML or PDF format [1]. PDF format is the only candidate if a document contains vector graphics and we want them to be scalable for the user without any loss of quality. This article describes the protocol by which the *DVIPDF*[8][9] conversion program from TEX's DVI format to Adobe's Portable Document Format (PDF) now manages to embed pre-existing PDF vector graphics into its PDF output files.

Initially, there existed just one program yielding graphics output in PDF format, namely Adobe Distiller, which converts pre-existing PS files. But now, the number of programs in this arena is growing day by day. We now have Ghostscript by Peter Deutsch [4] (noncommercial), Illustrator by Adobe, and a remarkable program, Mayura Draw, by Rajeev Karunakaran [6] (noncommercial). Incidentally, Mayura Draw was used to create the flow diagram and the flowers in this article.

What we call PDF graphics can in fact be a mixture of graphics and text involving PostScript (PS) Type 1 fonts. Furthermore, the current version of *DVIPDF* permits insertion of multi-page segments of pre-existing PDF files into the PDF output.

Hopefully this new capability of *DVIPDF* will largely compensate for *DVIPDF*'s continuing inability to integrate EPS (Encapsulated PS) on its own[1]. Indeed, qualitatively speaking, any sort of (still) graphics image can now be efficiently integrated into *DVIPDF*'s output. Currently *DVIPDF* is

1 i.e. without first converting EPS to EPDF using, for example, Ghostscript.

supported by the `graphics` and `graphicx` packages [2][3]. It will also be supported, where graphics objects are concerned, by an extension of the `boxedeps` package [10] .

In conclusion, we will briefly speculate on the possibility and interest of using *DVIPDF* and Acrobat Reader in tandem to provide something qualitatively new, namely a freely distributable DVI viewer with unlimited graphics abilities.

2. EPDF, MPDF and OPDF

We introduce some terminology for the exposition to follow:

— MPDF (Main stream PDF file) refers to a PDF file into which we wish to insert vectorized graphics at points specified by suitable markup. This PDF file is an intermediary (and incomplete) product that normally exists only inside the *DVIPDF* program; it is derived from a DVI file, along with fonts. As one would expect, the markup in the MPDF is derived from the `\special` commands in the DVI file.

— EPDF (Embedded PDF file) refers to a PDF file containing one or more graphics objects that are available for insertion into the MPDF file and are designated in ways we will explain presently. We note a couple of restrictions that DVIPDF currently imposes: an EPDF file has to be in ASCII format (*not binary*) and its page content has to be non-compressed.

— OPDF (Output PDF file) refers to the final complete PDF file produced by the *DVIPDF* program by integrating into the MPDF file some of the graphics objects in one or more EPDF files or bitmapped files of norms PNG and JPEG. Since the integration by *DVIPDF* of PNG and JPEG graphics has been described before [9], all mention of concurrent integration of PNG and JPEG files will be omitted from what follows.

3. Simplest integration

Now we can discuss an introductory example of vector graphics integration. The ‘Flower’ drawing (Figure 1) is an EPDF file created in *Mayura Draw*. Its integration with some text is presented in Figure 2. This is the simple sort of integration that has for many years been very popular with an EPS files in place the EPDF file. (We will do something more impressive in the next section!)

It is instructive to follow what happens in this simplest integration process. It involves one of the more important concepts of the PDF file format, namely,

Figure 1

'Form XObject'. For brevity we use the term 'Form' throughout this paper. This is a composite object analogous to a box in the the TEX realm — in as much as each has a bounding box and can contain others of its kind. Where we want to embed a picture, a *reference* to a Form occurs in the MPDF file, in a part called a page stream. After all the page streams are completed the Form referenced is created from the EPDF file and installed further along in the OPDF file.

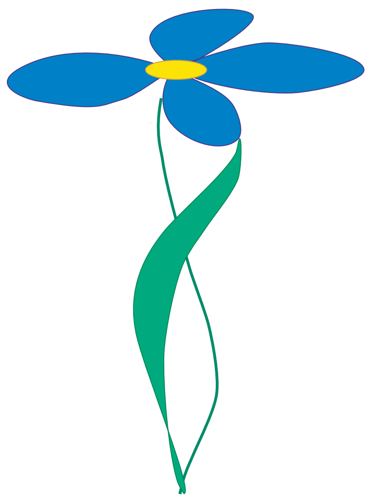
Let us be a bit more explicit about the integration of the flower. In the TEX typescript, the integration is commanded by:

```
\resizebox{4in}{!}{%
    \includegraphics[bb= 83 237 477 767]{flower.pdf}}%
```

Время и Случай

И обратился я, и видел под солнцем, что
не проворным достается успешный бег,
не храбрым – победа,
не мудрым – хлеб,
и не у разумных – богатство,
и не искусным – благорасположение,
но время и случай для всех их.

– Екклесиаст 9:11



I returned and saw under the sun, that
the race is not to the swift,
nor the battle to the strong,
neither yet bread to the wise,
nor yet riches to men of understanding,
nor yet favor to men of skill;
but time and chance happeneth to them all.

– Ecclesiastes 9:11

Figure 2

This assumes LaTeX, and the `graphicx` package, activated by

```
\usepackage[dvipdf]{graphicx}
```

The bounding box (`bb`) given here was taken from a comment

```
/MediaBox [83 237 477 767]
```

that Mayura Draw inserted into the EPDF file flower.pdf. In a later section we will discuss ways of automating the choice and use of bounding boxes.

At the DVI level, there results three special commands with the following three arguments.

```
pdf: /S 0.731 0.731 <<
pdf: /GRAPH flower.pdf  llx=83 lly=237 urx=477 ury=767
pdf: /S >>
```

Here `/S` is for scaling.

In the OPDF file the Form reference in the page stream occurs as `/Fm1` in the midst of sizing and positioning commands, and further on, one encounters the form itself which begins as follows:

```
21 0 obj
<< /Type /XObject /Subtype /Form
     /FormType 1 /Matrix [1 0 0 1 0 0]
     /BBox [ 83 237 477 767 ]
     /Name /Fm1 /Resources 23 0 R /Length 24 0 R >>
stream ...
```

Hopefully these samples give some idea of the internal workings. Naturally, detailed understanding is for experts only, and requires familiarity with the PDF format specification [1].

4. Complex paste-up

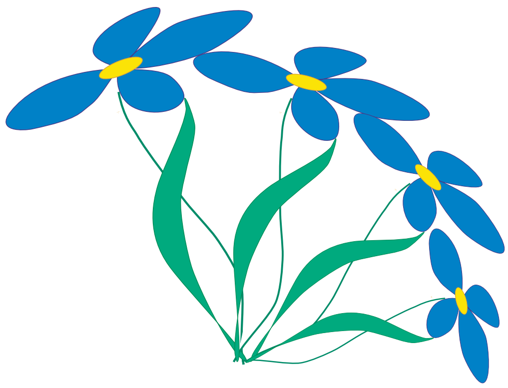
We now describe a less familiar and more interesting sort of integration. It is possible to apply the above techniques to scavenge various fragments from PDF files, perhaps several from a single PDF page, and then use the fragments like tiles to build up a new PDF. For example, Figure 2 can be scavenged to create Figure 3.

The strategy is obvious: scavenge three Forms from Figure 2 corresponding to the two text blocks and the flower. Then use TEX and *DVIPDF* in sequence to construct Figure 3 from these three pieces. Note that all of Figure 2 except the original Russian title is scavenged, and that similarly the only really new material in Figure 3 is the English title. The one flower is reused many times via a new linear mapping each time. Note also a significant economy: since the

Time and Chance

I returned and saw under the sun, that
the race is not to the swift,
nor the battle to the strong,
neither yet bread to the wise,
nor yet riches to men of understanding,
nor yet favor to men of skill;
but time and chance happeneth to them all.

– Ecclesiastes 9:11



И обратился я, и видел под солнцем, что
не проворным достается успешный бег,
не храбрым – победа,
не мудрым – хлеб,
и не у разумных – богатство,
и не искусным – благорасположение,
но время и случай для всех их.

– Екклесиаст 9:11

Figure 3

flower is a Form, the cost in storage space of using it many times in Figure 3 is surely going to be little more than that for using it once.

The most delicate point is to understand just how, starting from Figure 2, the three Forms are scavenged, namely the two blocks of text and the flower. If one examines the PDF file for Figure 2, one observes that the two blocks of text are part of one and the same primitive PDF 'object'; therefore it would be very awkward to extract them neatly. Thus a rough way is used to scavenge them: one extracts *the whole page*, views it as a Form, called say `/Fm0` and then (in just a few lines) formally derives three Forms `/Fm1`, ... ,`/Fm3` by cropping `/Fm0` down to the three boxes of material we need. This construction of `/Fm1`, ... ,`/Fm3` is summarized in the flow diagram (Figure 4).

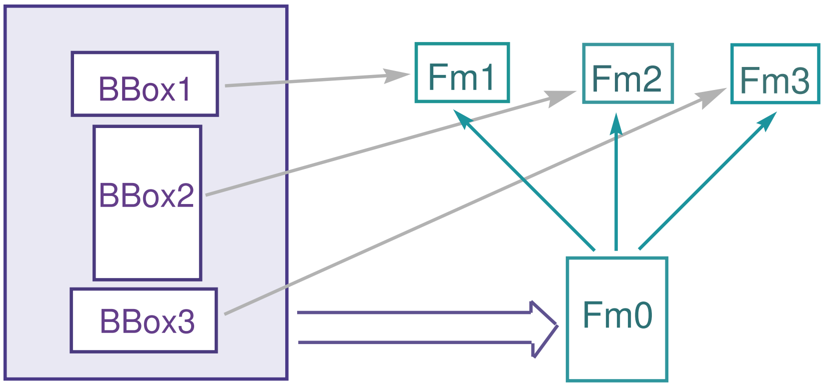


Figure 4

As we know from studying the construction of Figure 2, the flower does occur as a Form, say `/Fm4`, within the the page form `/Fm0`. Thus one could have extracted it without using the cropping device, and then used it in place of `/Fm2`. However there is nothing to be gained thereby in the present case.

5. Finding bounding boxes

For the user, the two main steps in using an EPDF graphics object in a TeX typescript are:

— Select bounding box on the graphics page,
— Make TeX exploit the bounding box parameters.

After the first of these two steps we store the bounding box parameters in the same sort of auxiliary file as is used by the `graphics` package for bitmap files. Such an auxiliary file has an extension `".bb"`. In the case of a one-page EPDF with a single bounding box selected, it can consist of a single line of the form:

```
%%BoundingBox: 0 0 144 72
```

The simplest (but also the clumsiest!) way to choose a bounding box is to print out the EPDF and to draw and measure the bounding box by hand. The units are in big points measured from the lower left corner of the PDF page.

A more convenient way uses the commercial program Adobe Acrobat Exchange and also some 'plug-ins' for it. One opens the EPDF file and crops to a user-selected region. Then the needed bounding box appears as a `/CropBox` comment in a new PDF file.

Hopefully both of the following methods for EPS files will soon be extended to PDF files.

Recall that GhostScript, in conjunction with a header file called `bb4gs.ps`, provides a quick way to determine the (cropped) bounding box for a one-page graphics object of norm EPS. The GSview program [7] does something similar, and more automatically. With 'hand marking' of regions, GSview is useful for selecting multiple bounding boxes from the same page.

6. Finding bounding boxes automatically

A more difficult problem is to make truly *automatic* the acquisition of bounding box data for graphics insertions from an EPDF. As TeX is not very suitable for parsing PDF files (even ASCII), this job is one for *DVIPDF*. This can require hints from the author such as page number in the EPDF, and more.

When the graphics object is alone on a single page PDF, we can often get a good approximation to the wanted bounding box by simply searching for a comment of type `/MediaBox` or `/CropBox`. For example Mayura Draw puts the comment

```
/MediaBox [ 50 209 488 786 ]
```

in the body of the EPDF file representing the the flower of Figure 1.

For the remainder of this section, we simply assume that *DVIPDF* is somehow able to find the bounding box *automatically*.

This situation is then handled by a protocol that involves TeX macros in addition to the parsing by *DVIPDF*. It requires two passes as described by L. Siebenman for integration of bitmapped images (PNG and JPEG); see Section 4 of [11]. On first pass, *DVIPDF* produces an auxiliary file with bounding box parameters. On the second pass, TeX is able to use this bounding box, and *DVIPDF* accomplishes the wanted insertion of the EPDF.

In practice, it may occur that the bounding box found automatically by *DVIPDF* should been changed. For example, the received bbox from MediaBox will be constant, if it is preferable to save the initial PDF file without changing, and we want to select a different bounding box. Using syntax from the `boxedeps` package, one might have to precede the insertion by a command `\TrimRight{40pct}`; this would cut off the righthand 15% of the automatically found bounding box.

7. Logistics and efficiency

To incorporate EPDF with MPDF we have to add all needed components or objects to database of MPDF, we have to rename all objects corresponding to our main database, and we have to avoid repeated loading of the same objects.

Insertion EPDF is based on one of the more important concepts of PDF, namely "*Form XObject*"; we use term "*Form*" in this paper. How does it work? When we want to embed a picture, we place *Form* reference in MPDF page stream. After producing all pages we place this *Form* really. It is produced on the base of the EPDF page stream with renaming of objects on the fly, and finally needed objects are added in MPDF. If this stream has its own *Forms*, the process of parsing is recursive.

First we mention some logistic problems arising from the fonts used in EPDF graphics objects. *DVIPDF* keeps a font database with systematic nomenclature to help solve them.

Two fonts of the same name may in fact be different. Their versions may be different; even the familiar Times of Adobe has changed some shapes and parameters. When are different versions 'essentially equivalent'? Another complication: the encodings may be different. Should they be realigned?

Acrobat Reader loads just one font with a given name; thus although one can store distinct fonts under the same name in a PDF file, such a tactic is not just chancy; it is guaranteed to fail.

Two radically different partial fonts derived by subsetting from the same standard font can sometimes bear the same (augmented) name. This is unlikely, but it can occur, particularly when different engines have been used to extract the subsets. For example *DVIPDF* itself is one such engine; it augments the original font's name using a cyclic redundancy check value (CRC)[5] computed from codes and widths of characters used. Other engines use other recipes.

One of the weak points of the PDF format is the relative bulkiness of PDF files compared with HTML files. PDF files do strain Internet's capacity; thus it is very important that PDF files posted on the Internet be no bigger than absolutely necessary. *DVIPDF* strives to be space efficient when integrating graphics.

When, like the flower in this article, a graphics object is used several times with simple variations, it is important that only one be stored in the PDF file, and that all variations be produced by suitable references to that one. *DVIPDF* keeps a graphics database with systematic nomenclature to permit this.

Such efficiency in EPS integration is possible (with *DVIPS* for example) but is rarely attempted; *DVIPDF* makes it routine practice.

Exploiting *DVIPDF*

As explained in [8], *DVIPDF* is, at the programming level, an offshoot of T. Rokicki's freeware *DVIPS*, and when mature it may even be reintegrated with *DVIPS*. In contrast to Adobe Acrobat Exchange *DVIPDF* is a 'blackbox' program in portable C code, and for many users it badly needs a more comfortable interface. Currently *DVIPDF* (console mode for Windows 95) permits a simple 'drag-and-drop' interface to be used. There are two obvious approaches to interfacing.

— Make *DVIPDF* a tail end to TeX, allowing even the most standard TeX installation to output PDF files automatically.
— Make *DVIPDF* a front end to Acrobat Reader.

The second approach seems particularly interesting. Indeed, Acrobat Reader is the 'definitive' PDF reader, and further the concept of 'plug-ins' for it has been formalized in a reasonably platform-independent way.

Such 'plug-ins' convert Acrobat Reader into a DVI reader that is able to view graphics of norms PNG, JPEG, (E)PDF — and most graphics falls comfortably within their scope. There are problems here, related to variable `\special` syntax, but they seem to be solvable [11]. The concept of an electronic posting that consists not of a single file, but rather of a directory of files, including graphics files and a DVI file, has advantages as well as disadvantages. Indeed the autonomous graphics files are both attractive and useful.

DVIPDF 'plug-ins' for Acrobat Reader would permit the scientific community to continue indefinitely to benefit from some still unique talents of the DVI format: enviable space efficiency; enviable coverage of heritage printers and screen displays; and finally enviable browsing speed as exemplified by the freeware viewers *XDVI* and Textures.

Acknowledgments

I would like to thank Laurent Siebenmann for his useful suggestions and the English edition of this paper.

Bibliography

[1] Tim Bienz, Richard Cohn.
Portable Document Format Reference Manual.
Adobe Systems Incorporated, 1993.
Addison-Wesley Publishing Company. ISBN 0-201-62628-4.

[2] David Carlisle, Sebastian Rahtz. *The graphics package*.
`ftp://ftp.tex.ac.uk`

[3] David Carlisle, Sebastian Rahtz. *The graphicx package*.
`ftp://ftp.tex.ac.uk`

[4] L. Peter Deutsch. *Aladdin Ghostscript*. Electronic distribution:
`ftp.cs.wisc.edu://pub/ghost/aladdin`

[5] L. Peter Deutsch. *RFC 1952: GZIP 4.3 specification*.
`ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html`

[6] Rajeev Karunakaran. *Mayura Draw*.
Electronic distribution: `http://www.wix.com/mdraw210.zip`

[7] Russell Lang. *GSview*. Electronic distribution:
`/ftp.cs.wisc.edu://pub/ghost/rjl/gsview*.zip`

[8] Sergey Lesenko. The DVIPDF Program.
TUGboat 17, 3, September 1996, pp. 252–254.

[9] Sergey Lesenko. *DVIPDF and Graphics*.
"TEX and Scientific Publishing on the Internet".
Proceedings TUG'97, San Francisco.
TUGboat 18, 3, 1997, (to be published).

[10] Laurent Siebenmann. DVI-based electronic publication.
TUGboat 17, 2, June 1996, pp. 206–215.

[11] Laurent Siebenmann. *The boxedeps package*.
`ftp://ftp.tex.ac.uk`