Inserting external figures with GraphicP

Szabó Péter
Budapest University of Technology and Ecomomics
Department of Analysis
Műegyetem rakpart 3–9.
Budapest
Hungary H-1111
pts+eurotex@math.bme.hu
http://www.inf.bme.hu/~pts/

Abstract

This paper describes GraphicP, a new, unified LaTeX and plain TeX package that provides a fast and reliable method for including external images into TeX documents. The `\includegraphics` macro of GraphicP is a drop-in replacement of the same command of LaTeX graphics.sty and graphicx.sty, but with many enhancements. Drivers for xdvi, dvips, pdftex and dvipdfm are included. Useful tips are given for converting vector and bitmap images into a format usable for inclusion (typically EPS or single-page PDF).

Overview of image inclusion

There are many different technologies to embed external images, such as photos, figures, function plots and diagrams into TeX documents. Most methods involve the following steps:

1. designing (drawing) the image in an external program;
2. converting it to a file format recognised by TeX;
3. loading an embedder (a TeX package that can embed images);
4. placing inclusion commands at places of the `.tex`; document where the image should appear

There are additional, less common steps:

5. providing feedback to the image drawing program about the final image size and position;
6. replacing fonts and glyph sizes in the image to match the main text font, compensate the effect of scaling and to allow TeX math formulas;
7. replacing line widths to compensate scaling;

In some technologies, these steps are integrated, so the non-WYSIWYG image code can be typed directly into the `.tex` document. Examples are the LaTeX portable picture environment and its variants; epic, eepic, gastex, Xy-pic, MFpic, ConTeXt's inline METAPOST environment. This paper discusses only the generic process of embedding external images, not these integrated, specific solutions.

In WYSIWYG word processors, images are usually inserted by opening them in their appropriate appliaction, and using the clipboard to copy them to the document inside the word processor. Scale, rotate, move and resize operations are performed with the mouse, providing instant feedback to the author about the final appearance of the image and how it affects text flowing around it. Compared to this, the steps above seem to be overcomplicated, and a real pain to the author. This is partly because easy handling of images needs a WYSIWYG environment with instant feedback while editing. TeX – by design – lacks both of them. So inserting images to TeX documents is expected to be a tedious process, no matter how sophisticated tools are used. Nevertheless, it is worth improving the tools, so TeX can compete with other document preparation methods. Emphasis should be put on quality, stability, compatiblity and output size, not on ease of use.

This paper describes steps 2–4 in detail, using tools traditional for years, including Ghostscript, dvips, dvipdfm, pdftex, METAPOST, the convert utility of ImageMagick, The GIMP, as well as replacement tools from the new GraphicP package, available from CTAN. The a2ping.pl utility replaces epstopdf and others, img_bbox.pl replaces ebb from dvipdfm, pdfboxes.pl improves PDF files, graphicp.sty replaces traditional LaTeX graphics.sty and plain TeX epsf.tex. These new tools work together with the sam2p [1] raster image converter, which replaces tiff 2ps and the EPS and PDF export filters of other image processing software such as convert. This paper compares the new and the replacement tools

in detail, and proposes a more reliable and accurate general image embedding technology for both LaTeX and plain TeX, using the proper combination of these programs.

The embedder. TeX doesn't understand image files. In particular, TeX cannot extract an individual pixel, line or label from an image. The only reason why the embedder macros running in TeX read the image file that they need the bounding box for proper scaling. The bounding box (or *bbox*) is a rectangular area of the image file that contains all visible parts of it. The bbox is of the form $\langle llx\rangle$ $\langle lly\rangle$ $\langle urx\rangle$ $\langle ury\rangle$, in which the point at $(\langle llx\rangle, \langle lly\rangle)$ is the lower-left corner of the image, and the point at $(\langle urx\rangle, \langle ury\rangle)$ is the upper-right corner. It is a common tradition to have $\langle llx\rangle = 0$, $\langle lly\rangle = 0$, $\langle urx\rangle$ = image width, $\langle ury\rangle$ = image height. Once the bounding box has been extracted, and the desired width of the included image is known, the embedder can calculate the actual height and leave empty place for the image on the paper.

The embedder inserts the image file name, the computed horizontal and vertical scale factors into the DVI file as a `\special`. The printer driver is responsible for loading the image file and sending it to the printer properly scaled and rotated. The image file format must be compatible with the printer driver. For example, the popular printer driver dvips requires all images to be in the Encapsulated PostScript (EPS) format. pdftex accepts PDF, PNG, JPEG and TIFF images. dvipdfm accepts PDF, PNG, JPEG and METAPOST EPS images. Usually the author of the image prefers a different file format for development, so conversion is necessary. Some converters are safe, efficient and faithful in the sense that they create valid and small output without information loss, but others have to be used with great care.

The most important requirements of an embedder are:

1. ability to specify the image size and/or scaling
2. ability to specify rotation angle and mirroring
3. ability to clip unnecessary parts (crop)
4. extracting the bounding box properly from *all* file formats
5. full compatibility with dvips, pdfTeX, dvipdfm
6. compatiblity with other printer drivers
7. reuse of the same image object if embedded multiple times
8. running text around the image
9. support for floating figures

Requirement 9 has been well supported for a long time in LaTeX, by the `table` and `figure` environments. However, requirement 8 is solved only with limitations in floatflt.sty. The other requirements are correlated more strongly, they are implemented in the de facto standard for an embedder of LaTeX: graphics.sty (includes both `graphics.sty` and `graphicx.sty`). It is well-documented, and it provides a convenient syntax and unified (printer driver and file format-independent) interface for including any kind of rectangular image into LaTeX documents. Using it is rather easy:

```
...
\usepackage[dvips]{graphicx}
... \begin{document} ...
\begin{figure}
  \includegraphics[width=0.9\textwidth]%
    {footown.eps}
  \caption{The map of Footown}
\end{figure}
... \end{document}
```

graphics.sty is not available for plain TeX, but there is a similar but less powerful package (specific to EPS images), epsf.tex. An example:

```
\input epsf  % in plain \TeX
...
\epsfxsize=0.9\hsize \epsfbox{footown.eps}
```

Why GraphicP? graphics.sty contains several inconsistencies and weaknesses, which document authors must care of when embedding images. Some of these problems are just annoying quirks, others affect the image placement and scaling, visible to the reader. GraphicP intends to be a stable and accurate replacement for graphics.sty, fixing many problems, but without changing the syntax of the `\include graphics` command substantially. It doesn't contain fundamental additions, only small fixes and additions, and increased consistency and portability.

As GraphicP evolved, it has been extended with Perl scripts, for example a2ping.pl and other standalone programs in addition to the TeX macros in graphicp.sty. These external programs are optional, because a UNIX system is needed by some of them.

graphics.sty implements a framework, separating the general high level functionality from the printer-driver specific low-level one. For example, the command `\usepackage[pdftex]{graphicx}` loads the graphicx.sty user interface with the pdftex.def driver. (The other interface, graphics.sty differs in the syntax of the `\includegraphics` command).

This separation is a good design choice in general, but it makes more difficult to do fundamental changes. Another drawback of graphics.sty is that it

is tightly integrated into LaTeX, so it would be quite difficult to add plain TeX compatibility. GraphicP has been implemented from scractch. The name of its embedder, pgraphicp.sty, suggests that it can be used as a replacement of graphics.sty. For example, the LaTeX example of Footown above works by replacing the first line with

```
\usepackage[dvips]{graphicp}
```

Preparing the image for inclusion

PostScript is a two dimensional page description language that can fully describe the visual appearance of printed material. A PostScript document is composed of straight lines, curves, filled regions delimited by these, text, bitmap images and others. PDF is a portable document format mainly for distributing two dimensional, mainly static material in electronic form. The graphics model of PostScript and PDF is the same, so – in the ideal case – there is no loss of information or precision when converting between PostScript and PDF.

Other important differences, such as sheet trimming, colors, slide shows and web hyperlinking are not considered in this document.

Any image (bitmap, vector and combined) can be faithfully represented in PostScript and PDF. The EPS (Encapsulated PostScript) file format is a single-page PostScript with some other minor restrictions, so it is perfectly suitable for representing inline images in a document. There is no restriction in PDF, any one-page PDF can be treated as an image. Most TeX printer drivers accept the images in either EPS or PDF, so the goal of this section is to convert any image to both of these formats. Note that TeX and dvips cannot embed normal PostScript documents, because the bbox is missing, and PostScript documents may contain device dependent or global state changing operators. Use a2ping.pl to convert PS to EPS.

Most Windows users working with a word processor haven't heard of PostScript, and have never used PDF for embedding. The Windows clipboard and OLE hides the file format; as long as the image can be copy-pasted, its format doesn't matter. This user-friendly approach is not available in TeX, beacause it is technically impossible to copy-paste binary image data into the human-readable `.tex` source. Thus each image has to be saved into a separate file, and the `.tex` file contains only references to these files in the form of `\includegraphics` commands. The file format depends on the printer driver, it should be – in general – EPS for TeX combined with dvips, and PDF for pdftex and dvipdfm.

Conversion to EPS or PDF. EPS and PDF files are not editable easily, so they should be converted or exported by the preferred image drawing appliaction of the author. Most modern vector graphics editing programs, including Illustratpr, CorelDRAW, Visio, Acrobat, InDesign and QuarkXPress have direct EPS export feature; some of them can export PDF.

One has to consider the quality, the compatibility and the size of the exported image. Some programs build up circles using a constant number of straight lines or cannot emit glyphs in vector format (⇒ low quality), some images contain proprietary or legacy junk, or they need new features of PostScript LanguageLevel 2 or 3 not supported by old printers (⇒ low compatibility), some programs emit 100 kB of procedure sets that are not used anyway, or they represent object inefficiently (⇒ big size). Thus it is worth knowing more than one way to do the same file format conversion. Unfortunately, there is no golden rule: the best EPS and PDF output can be found only by experimenting.

If the original image is in raster format, the sam2p command-line utility should be used to create a small and compatible PDF or EPS file. sam2p – with the help of tif22pnm, png22pnm and djpeg – can read today's most popular raster image formats. The author should save the image in an intermediate format (recommended: PNG or TGA), and feed that to sam2p.

The first page of a normal PostScript document can be converted to an EPS or PDF with a2ping.pl, part of GraphicP, for example: `a2ping.pl in.ps out.eps`. For the PS→PDF conversion, a2ping.pl calls Ghostscript with the device `pdfwrite`. Ghostscript 7.00 or more recent is recommended to avoid missing objects and low-quality glyphs. The PDF output is quite small. As an alternative, Acrobat Distiller can be used for converting PS to PDF, but it is not free, and the settings should be specified properly to create a small and compatible PDF.

There is a general EPS export method for any Windows and Macintosh application that is able to print. It needs the PostScript printer driver freely available from Adobe [10]. The PostScript printer description should be `adist4.ppd` [11]. Select the following features during installation: `adist4.ppd`, print to file, optimize for compatibility (ADSC), PS LanguageLevel 2, embed all the Type 1 (vector) and TrueType fonts as Type 1 fonts, embed even the standard fonts. One may choose between PS (DSC) or EPS output to avoid rotation. Scaling is not important, because the TeX embedder is able to rescale images. Translation can be compensated by

editing the bounding box comment by hand. After setting it up, any document can be printed from any application to a PostScript file – it works similarly as normal printers, but a file will be created on disk. The bounding box can then be modified by hand, and a2ping.pl may be applied if necessary. If the application can create EPS files by it self, it should be compared to the printer-driver-based generic method.

Adobe distributes a utility named PDFWriter that can be used from any Windows application to print to a PDF file. Using it is not recommended, because it completely confuses accented glyphs, and even other glyphs in some fonts.

PostScript has been the traditional page description language on UNIX systems for a long time, so most UNIX utilities have PostScript output. The most important of these are:

- Acrobat Reader can print to PostScript, so it can be used as a PDF to PS converter. In the File/Print dialog, select *File* and *Level 2 Only*, uncheck *Fit to Page* and *Download Far East fonts*, check *Download Fonts Once* and *Use Printer's Halftone Screen*.

  As an alternative, invoke `acroread -toPostScript -level2 ...` from the command line.
- Ghostscript can convert PostScript to PDF. The epstopdf utility by Thomas Esser makes this easier, but a2ping.pl is a better replacement.
- `dvips -E` creates EPS files, but it sometimes computes the bounding box wrong, so a combination of `dvips` and `a2ping.pl` is necessary.
- Figures created by XFig can be converted to EPS with a command `fig2dev -L eps in.fig >out.eps`. Also XFig can create EPS files, and newer versions can even make TEX typeset some of the labels: The *Special Flag* should be set to *Special* in the *Text Edit panel*, and the file exported as *Combined PS/LATEX*, with name `out.eps`. Then `\input out.eps_t` should be called. Unfortunately it is impossible to post-scale the image this way.
- Both Netscape Navigator and Mozilla can print to PostScript, but their output is rather ugly. (The printed output of Internet Explorer is not so ugly, but tables are often cropped at right.)
- The GIMP can print to both PostScript and EPS, but sam2p is usually a better solution.
- pdftops from xpdf can convert PDF to PS and EPS. Unfortunately it doesn't work with weird fonts or encodings well, and older versions simply discard PK fonts embedded by pdftex.

The output of METAPOST doesn't need conversion, because it's already EPS, and the PDF-specific drivers in GraphicP can embed it to a PDF document without external converters. It is possible to embed TEX output into TEX: the PDF files created by pdftex can be included directly, and the PostScript output should be run through a2ping.pl to create EPS.

Sometimes a multi-step conversion produces the best results. For example, the author of this article often uses the pipeline of PostScript printer driver, Acrobat Distiller, Acrobat (cropping and EPS output), Acrobat Distiller, pdftops to create a small and portable EPS file that can be embedded.

Content management. Additional information has to be remembered for each image: proposed image title (caption), which documents or floating figures contain the image, the file name, the editable source file of the image, how it was converted from its source, is it available in another format (e.g. both EPS and PDF), the `\label` the figure will have etc.

Keeping this meta information up to date is important if either the images or the document containing them are planned to be reused in the far future. The author of the document should decide how to face this orgaziation task. Only some hints are provided here.

On UNIX systems it is traditional to write a `Makefile` for compilation, even document compilation. The enough_tex.pl is provided in the GraphicP distribution for convenience: it runs TEX enough number of times to resolve all references and indices. It can be inserted into a `Makefile` instead of bare `latex` invocations. `Makefile`s may also automate image conversion, creating EPS and/or PDF from the source images before compiling the document.

It is wise to retain an EPS, PDF, PNG or JPEG copy of each image, so they can be opened decades later, because tools that can open these formats with the same schemantics as today are expected to be available for a long time. In the contrary, proprietary and closed file formats should only be used for temporary storage – if the company won't support the file format any longer, it will be impossible to open such images later.

The same is true for TEX texts: one should make a backup of all classes, styles and auxilary macros files, including those that were used to create the format file, plus all source and image files belonging to the specific document. A full backup eliminates the risk that a `.tex` source doesn't compile anymore, or it compiles with different line breaks. The Linux `strace` utility can list all files opened by `latex` and other programs.

Features of GraphicP

Runs on both plain TEX and LATEX. It contains a compatibility layer (laemu.sty) in plain TEX that provides the LATEX package loading, error and warning indication, and some other simple common macros. Care has been taken to use plain TEX constructs whenever possible, e.g. `\def` instead of `\newcommand`. During development and testing, I have been using plain TEX, because porting it to LATEX is almost trivial compared to the other direction.

More accurate calculations. Knuth has designed TEX not to use floating point numbers. This is an important portability advantage, because the different rounding implementations of the CPUs don't affect the positions of line and page breaks calculated by TEX. It would be a painful headache, for example, if the same TEX document compiled differently on the author's Linux system and the publisher's Solaris system.

Real numbers in TEX are represented in 15.16 signed fixed point notation. That is, 15 bits are reserved for the integer part and the sign, and the precision is 2^{-16}, so a rounding up to 2^{-17} pt may occur after each operation. Fixed point arithmetics has the important advantage that additions and substractions are always accurate.

But what about multiplication and division? When scaling an image of size $wd \times ht$ to the desired width dwd, the actual height is calculated as $ht \cdot dwd/wd$. Multiplication and division are equally important operations of image scaling, and often the result of the multiplication exceeds the maximum TEX dimension of about 16000 pt.

TEX provides real number multiplication with the following trick:

```
\dimen0=42pt
\dimen1=0.333333333\dimen0
\showthe\dimen1
```

The calculated result (13.99979pt) is not accurate. Another weakness of the built-in multiplication is that introduces errors larger than the minimum 2^{-17} pt. For example 0.00001 in = 0.0011 pt, but 0.0000 05 in = 0.0 pt.

The TEX primitives are not suitable for scaling, because multiplication is not accurate enough, it results in overflow, and there is no built-in real number division at all. So a high precision, unoverflowable scaling operation has been implemented from scratch in div16b.sty. Its internal number representation is 30.32 signed fixed point, and it stores a number in two TEX `\count` registers. Input and output values are still 15.16 real numbers. Due to the increased precision, overflow can never occur, and the multiplication is always accurate, even $1\,\mathrm{sp} \cdot 1\,\mathrm{sp}$ isn't truncated. The division routine does repeated subtraction, possibly doubling or halving the divisor after each subtraction. Halving may introduce small internal rounding errors, but fortunately no error occurs when the scaling ratio is $a \cdot 2^b$, where a and b are integers.

Empirically, the error of the scaling algorithm in practical image sizes (100 . . . 3000 pt) is 0 . . . 2 sp, while the result of `\Gscale@div` in graphics.sty deviates sometimes as much as 1 pt, which is noticable. Another drawback of graphics.sty is that it calculates different widths inside normal latex and pdflatex.

Enforced dimensions. When the user specifies `[width=` or `[height=`, these will be enforced (using `\hss`), irrespective of what the driver generates. graphics.sty doesn't have this feature, and considering the less accurate image scaling, the differences up to a few pt can occur, and this can seriously effect further line and page breaks in the document.

Gives bbox hints. It is possible to convert a TEX page to EPS with `dvips -E`. dvips calculates the bounding box of the page automatically, taking into account the glyphs and rules on the page. Unfortunately, version 5.86e still detects the image bounding box wrong, especially with images descending below the baseline. GraphicP works around the problem by forcing dvips exclude the image from the calculation, and adds two small invisible (white) rules at the corners. This feature can be disabled.

A similar problem occurs in xdvi, which sometimes crops too much from the edges of the image. It is solved by forcing it not to crop at all. Cropping can be controlled from `\special{PSfile=...}`, but printer drivers interpret it differently, so the specification emitted by graphicp.sty disables cropping completely. This also solves a similar problem with dvipdfm, which would otherwise crop EPS images below the baseline.

File format detection. As opposed to graphics.sty, GraphicP doesn't rely on the the file name to determine the file format. Files with bogus or invalid extensions are treated properly, and EPS files created by METAPOST (`img.1`, `img.2` etc.) are also embedded well. The annoying bug of graphics.sty of failing to recognise `an.image.pdf` as `.pdf` is also eliminated.

File format decisions (implemented in pts_bbox.sty) are based on the first four bytes read from the file. GraphicP can distinguish between PNG, TIFF, JPEG, MPS (EPS created by METAPOST), EPS, DOS EPSF and PDF properly.

External bbox parsing. graphics.sty can read tiny `.bbx` files that contain only a `%%BoundingBox` comment. This is faster and more accurate than full image parsig, because `.bbx` files have extremely simple syntax. GraphicP accepts `\graphicPmeta` commands instead of `.bbx` files. Each of these commands describes a single image file, for example:

```
\graphicPmeta{i.1}{EPS.MPS}{0}{0}{99}{534}
\graphicPmeta{i.2}{EPS.MPS}{8}{9}{10}{76}
```

A list of these lines can be inserted right into the `.tex` document before typesetting the images, or it can be `\input` from a separate file (proposed extension: `.gpm`).

A Perl script called img_bbox.pl is included in GraphicP to generate these lines. For example, use the command `img_bbox.pl -tex *.eps *.pdf *.jpg *.tiff *.png >all.gpm`. The script supports more than 42 image formats, included all formats embeddable by graphicp.sty. File names may contain TEX control characters, they are properly escaped.

Getting the bbox of a PDF (the `/MediaBox`) is rather complicated, because it is deeply hidden somewhere in page tree of the binary PDF file. Neither GraphicP, nor graphics.sty can do this reliably when running inside tex, both of them expect the `/MediaBox` to be in a line on its own, and they can be confused when there are multiple such lines. The solution is to run GraphicP inside pdftex, or – to get all four bbox coordinates – to use img_bbox.pl, which can navigate the PDF page tree properly.

As an alternative, GraphicP contains another Perl script, pdfboxes.pl, which modifies an existing PDF file so that the bounding box will be available right in the beginning. This is essentially a proof-of-concept implementation: it proves that it is possible to insert a new object into a PDF file and modify all offset references to other objects, without the need to parse and regenerated the full PDF. sam2p 0.43 and above emits the bounding box early enough, so pdfboxes.pl is not needed.

a2ping.pl can detect all 3 bounding box types, the `setpagedevice` and other PostScript operators.

Internal bbox parsing. The bbox extraction capability of graphicp.sty is limited by the fact that TEX reads files line-by-line, thus it is hardly possible to parse a binary file properly. `\catcode`s are used extensively to ignore most of the binary "junk", so the dimensions of PNG, TIFF and JPEG files cannot be extracted, and PDF parsing is very limited. Fortunately, pdftex provides primitives to extract the bounding boxes of these binary files, and dvips doesn't support these file formats anyway.

All 3 types of EPS bounding boxes are supported, the user can choose between the exact and the rounded bbox, if both of them are present. The default is to choose `Exact`, then `Hires`, then normal (rounded to integer) bounding box. This can be overridden by `[hiresbb]` and `[exactbb]`.

Nonzero depth. Images can be lowered below the baseline, for example `\includegraphics[lower=20]{t.eps}` moves the image down by 20 bp. Also GraphicP can recognise negative lower-left in the bbox, and move the image below the baseline automatically (only with `[below]`). Horizontal movement is not possible, but the user can insert the appropriate `\kern` commands before and after the image.

Raster images are always aligned onto the baseline, so an explicit `[lower=]` or `[raise=]` should be used instead of `[below]` to move them vertically. Alternatively, sam2p has the `-m:lower:`⟨*dimen*⟩ option that creates a pre-lowered EPS or PDF from the raster image.

a2ping.pl automatically raises images up to the baseline, unless the `--below` option is given.

Avoids duplication. dvipdfm and newer versions of pdftex both support Form XObjects, a means for reusing material already typeset. GraphicP uses From XObjects to embed an image file only once, no matter how many times it appears in the document. This optimization is impossible in PostScript documents, because they are read sequentially, and caching images already read imposes a high risk of memory shortage.

METAPOST with all drivers. Although METAPOST generates EPS, these text files follow such a simple structure that they can be converted to PDF drawing operators (`\pdfliteral`) withing TEX, as done in ConTEXt's `supp-pdf.tex`, written by Hans Hagen. GraphicP loads this routine in case such an MPS image is to be loaded. However, these macros consume pretty much TEX memory, so they can be disabled by saying

```
\usepackage[nopdftexmpost]{graphicp}
```

In fact, METAPOST output is the only image format that is supported by all drivers of GraphicP. It is recognised by an ADSC comment in the EPS header, the file name doesn't matter – it can be `t.1`, `t.eps`, `t.ps` or anything else.

METAPOST doesn't emit a high resolution bbox by default, but `context/mp-tool.mp` adds the appropriate code to `extra_endfig`. Alternatively, one can type this into the beginning to the `.mp` file:

```
extra_endfig := extra_endfig & "special ("
& ditto & "%%HiResBoundingBox: " & ditto
```

```
&"&decimal xpart llcorner currentpicture&"
                    & ditto & " " & ditto
&"&decimal ypart llcorner currentpicture&"
                    & ditto & " " & ditto
&"&decimal xpart urcorner currentpicture&"
                    & ditto & " " & ditto
&"&decimal ypart urcorner currentpicture"
& ");";
```

METAPOST creates EPS files with the wrong extension. When reading `ajob.mp`, the figure under the scope of `beginfig(42)` will have the file named `ajob.42`. Passing a negative number of `beginfig` will create `ajob.ps`. These settings are hard-wired into METAPOST, thus the files have to be renamed before inclusion with graphics.sty. GraphicP doesn't have this limitation.

Better Babel compatibility. Many Babel languages make the characters " and ‘ active. This conflicts with the METAPOST loader and other PDF-specific code graphics.sty borrows from ConTEXt, so

```
\usepackage[pdftex]{graphicx}
\usepackage[magyar]{babel}
```

is the correct loading order. Users of GraphicP don't have to care, because it uses external code only for METAPOST EPS to PDF conversion, wrapped by proper `\catcode` resets, so no error occurs.

Backward compatibility. Although the TEX and LATEX interfaces used hasn't changed much in the last few years, pdftex is under development. GraphicP adjusts itself to the version of pdftex running it. GraphicP has been tested with pdftex 0.12r (Debian Slink), 0.14 and 1.00a (Debian Woody).

graphics.sty is not UNIX-specifix, it should work in any architecture or OS TEX has been ported to, but the Perl scripts provided in the GraphicP distribution require a UNIX system currently. They can be ported to other platforms easily if there is considerable interest.

For educational purposes, graphicp.sty (provides `\includegraphicP`) and graphicx.sty (provides `\includegraphics`) can be loaded in this order.

Draft support. It is often desirable to omit images from the document, especially in the development phase where fast compile–redisplay cycles are of primary importance. Of course, the place of the images still has to be reserved. Contrary to graphics.sty, GraphicP supports this kind of draft mode by specifying the appropriate driver, e.g.:

```
\usepackage[driver=invisible]{graphicp}
```

The supported draft drivers are:

invisible a transparent rectangle is displayed without the image

blackbox a solid black box is displayed, showing the bounding box – wastes a lot of ink

frame a black rectangular frame shows the bounding box

namedframe the image file name is displayed in the black rectangular frame. Uses `\textan` (in asciiall.sty) if available to display weird characters in file names.

The real drivers are:

pdftex the default driver when pdfTEXis detected. Cannot display EPS images.

dvi a common subset of dvips and dvipdfm. This is the default when running under normal (non-pdf-)TEX. Can display EPS and MPS only.

dvips cannot display PDF and raster images.

dvipdfm calls Ghostscript to convert EPS to PDF automatically.

The file `texmf/dvipdfm/config` should be updated to improve bounding box calculation and others during EPS to PDF conversion (don't forget to enter the actual path to a2ping.pl):

```
D "zcat -f %i | ./a2ping.pl --below - %o"
```

EPS fixups. a2ping.pl detects and emits all 3 bbox types with proper rounding, removes DOS EPSF binary junk, removes HP UEL header, adds some ADSC comments, can read and write from a pipe, converts PS to EPS, calls sam2p to convert raster images to EPS, removes form feed from end of EPS.

For the PDF→EPS direction, a2ping.pl invokes pdftops. a2ping.pl can output multipile-page PS, PDF and HP PCL5 documents, with corrected paper size (forced), resolution, duplex and tumble settings. The ouput of Ghostscript is post-processed if necessary.

a2ping.pl can shift images, so the lower-left corner is in the origin, but it can also retain the original bbox. This works for both EPS and PDF.

a2ping.pl is not only a converter, but it contains many fixup routines, so it can be used to fix EPS files from a source incompatible with dvips.

Work pending

cropping Now `\includegraphics` always embeds the whole image. It should have a `clip=` option, just as in graphics.sty.

transformations all 8 combinations of flipping and rotation by 90° should be added

imtrix a unified way for replacing labels in EPS and PDF images by those generated by TEX. Would be similar to psfrag.sty.

compatible options `\includegraphics` should have the following options, doing the same as in graphics.sty: `bb=`, `totalheight=`, `keepaspectratio=`, `type=`, `ext=`, `read=`, `viewport=`.

optional bbox dots the 1 sp wide white dots that forcibly mark the bounding box of the image should be made optional

arithmetics more complicated arithmentic expressions than `width=0.9textwidth` should be allowed in `width=` and similar options

testing with images originating from various programs

Conclusion

The most important benefit of TEX is that it helps authors and typesetters to produce beautiful printed documents. Although inserting figures to TEX documents isn't easy, TEX help us to make the images consistent and pretty. There are serious quirks and limitations during production, conversion and inclusion, but once the image has been included properly, it remains there without accident: it won't overlap the bottom margin (unless explicitly requested), it will be numbered and floated properly, it will never be torn from its caption etc. The total size of the images doesn't affect TEX: it runs happily (albeit slowly) on a book with thousands of large images, and never crashes. Beyond creativity, the authors must have the technical knowledge to create documents with images, but they can also enjoy many benefits unique in computer typography.

The author uses several tools when dealing with images: image editors, converters, the embedder and the printer driver. It is essential that these tools work properly and they can communicate with each other. GraphicP provides an embedder implemented from scratch that gives more flexibility to the author and instructs the printer drivers in a more compatible way than the traditional LATEX embedder, graphics.sty. GraphicP also contains many converters that fill the gap between the various output file formats of the powerful image editors and the formats the printer drivers work well with. The converters do not enhance the visual appearance of the image, but they ensure that bounding box and orientation information is emitted properly, and they also do some syntactical changes. They try to work smarty, without image-specific instructions from the caller.

Existing printer drivers are fairly good, provided that the embedder gives them specific and correct instructions what to do. However, existing WYSIWYG image editors cannot cooperate with TEX well. Their output has often to be adjusted by hand – or using specific converters.

The aim of graphicp.sty is not to compete with or replace graphics.sty, but to provide a proof-of-concept alternative showing that some of its functionality can be implemented better. The key, unmatched benefits of graphics.sty are the framework approach, support for many printer drivers, and the time it has been tested. Extending graphics.sty with the features of GraphicP while retaining these benefits would be a glorious, but enormous work.

The scripts and other programs of GraphicP are, as far as its author knows, unique. They can be used together with both graphics.sty and graphicx.sty, and even for generic image processing purposes unrelated to TEX. GraphicP is hoped to increase the efficiency of everyday image processing tasks done by TEX authors and publishers.

References

[1] Szabó Péter: *Inserting figures into TEX documents.* In proceedings to EuroBachoTex 2003.

[2] Hàn Thế Thánh, Sebastian Rahtz and Hans Hangen. *The pdfTEX user manual.* `teTEX:doc/pdftex/base/pdftexman.pdf.gz`, 1999.

[3] Mark A. Wicks. *Dvipdfm User's Manual.* `teTEX:doc/programs/dvipdfm.dvi.gz`, 1999.

[4] Tomas Rokicki. *Dvips: A DVI-to-PostScript Translator.* `teTEX:doc/programs/dvips.dvi.gz`, 1997.

[5] D.P. Carlisle. *Packages in the 'graphics' bundle.* `teTEX:doc/programs/grfguide.ps.gz`, 1999.

[6] Keith Reckdahl. *Using Imported Graphics in LATEX 2ε.* `teTEX:doc/programs/epslatex.ps.gz`, 1997.

[7] Adobe Developer Support. *PostScript Language Document Structuring Conventions Specification, Version 3.0.* Adobe Developer Technologies. 1992.

[8] Ed Taft, Steve Chernicoff and Carline Rose: *PostScript Language Reference.* Addison–Wesley, 1999.

[9] Jim Meehan, Ed Taft, Steve Chernicoff and Carline Rose: *PDF Reference. Second edition.* Addison–Wesley, 2000.

[10] `http://www.adobe.com/support/downloads/main.html`, select *PostScript printer drivers*

[11] PPD for a generic printer: `http://www.uniprint.ee/Web/OpenResource.aspx?ResFile=47` and `http://www.rgraphics.com/downloads/ADIST4.PPD`