Basic image formats

In TEX a graphic is not really known as graphic. The core task of the engine is to turn input into typeset paragraphs. By the time that happens the input has become a linked list of so called nodes: glyphs, kerns, glue, rules, boxes and a couple of more items. But, when doing the job, TEX is only interested in dimensions.

In traditional TEX an image inclusion happens via the extension primitive `\special`, so you can think of something:

```
\vbox to 10cm {%
  \hbox to 4cm {%
    \special
      {image foo.png width 4cm height 10cm}%
    \hss
  }%
}
```

When typesetting TEX sees a box and uses its dimensions. It doesn't care what is inside. The special itself is just a so called *whatsit* that is not interpreted. When the page is eventually shipped out, the dvi-to-whatever driver interprets the special's content and embeds the image.

It will be clear that this will only work correctly when the image dimensions are communicated. That can happen in real dimensions, but using scale factors is also a variant. In the latter case one has to somehow determine the original dimensions in order to calculate the scale factor. When you embed eps images, which is the usual case in for instance dvips, you can use TEX macros to figure out the (high res) bounding box, but for bitmaps that often meant that some external program had to do the analysis.

It sounds complex but in practice this was all quite doable. I say 'was' because nowadays most TEX users use an engine like pdfTEX that doesn't need an external program for generating the final output format. As a consequence it has built-in support for analyzing and including images. There are additional primitives that analyze the image and additional ones that inject them.

```
\pdfximage
  {foo.png}%
\pdfrefximage
  \pdflastximage
  width 4cm
  height 10cm
\relax
```

A difference with traditional TEX is that one doesn't need to wrap them into a box. This is easier on the user (not that it matters much as often a macro package hides this) but complicates the engine because suddenly it has to check a so called extension whatsit node (representing the image) for dimensions.

Therefore in LuaTEX this model has been replaced by one where an image internally is a special kind of rule, which in turn means that the code for checking the whatsit could go away as rules are already taken into account. The same is true for reusable boxes (xforms in pdf speak).

```
\useimageresource
  {foo.png}%
\saveimageresource
  \lastsavedimageresourceindex
  width 4cm
  height 10cm
\relax
```

While dvips supported eps images, pdfTEX and LuaTEX natively support png, jpg en pdf inclusion. The easiest to support is jpg because the PDF format supports so called jpg compression in its full form. The engine only has to pass the image blob plus a bit of extra information. Analyzing the file for resolution, dimensions and colorspace is relative easy: consult some tables that have this info and store it. No special libraries are needed for this type of graphic.

A bit more work is needed for pdf images. A pdf file is a collection of (possibly compressed) objects. These objects can themselves refer to other objects so basically we have a tree of objects. This means that when we embed a page from a pdf file, we start with embedding the (content stream of the) page object and then embed all the objects it refers to, which is a recursive process because those objects themselves can refer to objects. In the process we keep track of which objects are copied so that when we include another page we don't copy duplicates.

A dedicated library is used for opening the pdf file and looking for objects that tell us the dimensions and fetching objects that we need to embed. In pdfTEX

the poppler library is used, but in LuaTEX we have switched to pplib which is specially made for this engine (by Paweł Jackowski) as a consequence of some interchange that we had at the 2018 BachoTEX meeting. This change of library gives us a greater independence and a much smaller code base. After all, we only need access to pdf files and its objects.

One can naively think that png inclusion is as easy as jpg inclusion because pdf supports png compression. Well, this is indeed true, but it only supports so called png filter based compression. The image blob in a png file describes pixels in rows and columns where each row has a filter byte explaining how that row is to be interpreted. Pixel information can be derived from preceding pixels, pixels above it, or a combination. Also some averaging can come into play. This way repetitive information can (for instance) become a sequence of zeros even when actual changes in pixel values took place. And such a sequence can be compressed very well which is why the whole blob is compressed with zlib.

In pdf zlib compression can be applied to object streams so that bit is covered. In addition a stream can be png compressed, which means that it can have filter bytes that need to be interpreted. But the png file format can do more: the image blob is actually split in chunks that need to be reassembled. The image information can be interlaced which means that the whole comes in 7 separate chunks that get overlaid in increasing accuracy. Then there can be an image mask part of the blob and that mask needs to be separated in pdf (think of transparency). Pixels can refer to a palette (grayscale or color) and pixels can be encoded in 1, 2, 4, 8 or 16 bits where color images can have 3 bytes. When multiple pixels are packed into one byte they need to be expanded.

This all means that embedding a png file can demand a conversion and when you have to do that each run, it has a performance hit. Normally, in a print driven workflow, one will have straightforward png images: 1 byte or 3 bytes which no mask and not interlaced. These can be transferred directly to the pdf file. In all other cases it probably makes sense to convert the images beforehand (to simple png or just pdf).

So, to summarize the above: a modern TEX engine supports image inclusion natively but for png images you might need to convert them beforehand if runtime matters and one has to run many times.

Hans Hagen