dvisvgm: Generating scalable vector graphics from DVI and EPS files

Martin Gieseking

Abstract

dvisvgm is a command-line utility that converts DVI and EPS files to the XML-based vector graphics format SVG, an open standard developed by the W3C. Today, SVG is supported by many applications including text processors, graphics editors, and web browsers. Therefore, it's a convenient format with which to enrich websites and non-TEX documents with self-contained, arbitrarily scalable TEX output. This article gives an overview of selected features of dvisvgm and addresses some challenges faced in its development.

1 How it all started

In 2005 I was working on a wiki-based cross media publishing system called *media2mult* [3], which was supposed to produce documents in various output formats from a single source without the need to force the authors to scatter format and layout specific settings throughout the input document. Since the conversion back-end was built on the XML formatting technology XSL-FO, which at that time didn't provide sufficient math support through MathML, I needed a way to embed scalable TEX output in the XSL-FO files. The preferred format for this task was SVG, because it is XML-based and therefore fit nicely into the other involved XML technologies; for example, the files could be post-processed easily by applying XSLT and XQuery scripts. Furthermore, SVG was decently supported by Apache's open-source XSL-FO processor FOP and the related Batik SVG toolkit.

Fortunately, two DVI to SVG converters were already available, *dvisvg* [7] by Rudolf Sabo and *dvi2svg* [2] by Adrian Frischauf. Both utilities looked promising, and created nice results from my initial test files. dvi2svg even supported color and hyperref specials, which was another advanced requirement for my needs. However, the main drawback for the planned document conversion engine was that both tools relied on pre-converted SVG font files derived from a selection of common TEX fonts, notably the *Computer Modern* family. There was no simple way to process DVI files referencing arbitrary fonts supported by the TEX ecosystem. The latter had to be generated in advance by some kind of DVI preprocessing and by extracting the glyph data from PostScript or TrueType fonts, e.g. as described in [4, pp. 272–274].

Sadly, around that time, the development of both utilities apparently stalled, and the website of dvi2svg disappeared several months later. The alternative approach of creating SVG files from PDF didn't work satisfactorily either, due to the missing conversion of hyperlinks across pages inside the document, and the weak support of METAFONT-based fonts, which were embedded as bitmap fonts if no vector versions were available.

Since I had already written a couple of small DVI utilities before and therefore had working DVI and TFM readers available, I started to build a simple SVG converter on top of them. The first public release of dvisvgm was in August 2005. Since then, it's been a private free-time project and has evolved a lot over the years, largely because of wonderful feedback, detailed bug reports and interesting feature suggestions. dvisvgm is included in TEX Live and MiKTEX, and is also available through MacPorts.

2 About dvisvgm and basic usage

dvisvgm is a command-line utility written in C++. It supports standard DVI files with a version identifier of 2, as well as DVI files created by pTEX in vertical mode (version 3) and XETEX (versions 5 to 7).[1] The latter are also known as XDV files and are created if XETEX is called with option `-no-pdf`.

The basic usage of dvisvgm is straightforward and similar to other DVI drivers. If no other options are specified, it converts the first page of the given DVI file to an SVG file with the same name. If the DVI file has more than one page, the page number is appended to the base name. For example,

```
dvisvgm myfile.dvi
```

creates the file `myfile.svg` if `myfile.dvi` consists of a single page only, else `myfile-01.svg`. This was originally because the initial releases of dvisvgm could process only single DVI pages in one run; the behavior is still retained for compatibility. To select a different page or a sequence of page ranges, the option `--page` is required. It accepts a single page number or a comma-separated list of ranges. Regardless of whether any page numbers are specified multiple times, e.g. by overlapping range specification, all selected pages are converted to separate SVG files only once. The command

```
dvisvgm --page=1,3,5-9,8-10 myfile.dvi
```

is identical to

```
dvisvgm --page=1,3,5-10 myfile.dvi
```

and converts the pages 1, 3, and 5 through 10. The file names get the corresponding number suffixes as

[1] When an incompatible change in XETEX's XDV format is made, the DVI version identifier is increased. The recent XETEX revision 0.99998 creates DVI files of version 7.

`--output=`⟨*pattern*⟩	SVG file name of page 1
`%f`	`myfile.svg`
`%f-%p`	`myfile-01.svg`
`newfile-%p`	`newfile-01.svg`
`%f-%4p-of-%P`	`myfile-0001-of-20.svg`
`%f-%4(p-1)`	`myfile-0000.svg`
`%f-%(P-p+1)`	`myfile-20.svg`
`../%f/svg/%3p`	`../myfile/svg/001.svg`

Table 1: Effect of several output patterns applied to `myfile.dvi` consisting of 20 pages.

above. It's also possible to give open page ranges by omitting the start or end number:

```
dvisvgm --page=-5,10-
```

converts all pages from the beginning up to page 5, as well as page 10 and all following ones. Regardless of the number of pages converted, dvisvgm always prescans the entire DVI file in advance to collect global data, like font definitions, PostScript headers and hyperlink targets. In this way it is possible to convert selected pages correctly even if required information is located on excluded pages.

In order to change the names of the generated SVG files, `--output` can be used. It supports patterns containing the placeholders `%f`, `%p`, and `%P` which expand to the base name of the DVI file, the current physical page number, and the total number of pages in the DVI file, respectively. The command

```
dvisvgm --output=%f-%p-%P myfile
```

converts the first page of `myfile.dvi` to the SVG file `myfile-01-20.svg`, given that the DVI file contains 20 pages. The number of digits used for `%p` is adapted to that of `%P` but can be explicitly determined by a prepended number, e.g., `%4p`. Table 1 shows some further examples of specifying the naming scheme of the generated files. More details can be found in the dvisvgm manual page.[2]

Because of the lengthy text-based nature of XML documents, SVG files tend to be bigger than other vector graphics formats. To reduce the file size, the SVG standard specifies gzip- and deflate-compressed SVG files which normally use the extension `.svgz`. To create compressed files on the fly during a DVI conversion, the option `--zip` is available.

3 Font support

In contrast to PostScript and PDF, DVI provides no means to embed fonts into the file. Fonts are specified merely through their name, size, and a couple of additional parameters allowing the DVI driver to retrieve further data from the user's TEX environment. While this approach keeps DVI files compact, it also reduces their cross-platform portability and delegates significant processing to the driver. On the other hand, the requirement for a working TEX system enables full access to all font data, including those usually not embedded into PDF files. This is especially important regarding METAFONT-based fonts not available in other formats.

Since I was confronted with a wide variety of documents using a wide variety of fonts, it was important to provide dvisvgm with comprehensive font support including virtual fonts, various font encodings, CMaps, sub-font definitions, font maps, handling of glyph names and Japanese fonts which often use an extended TFM format called JFM. The proper mapping of PostScript character names, as used in Type 1 fonts, to corresponding Unicode points, required the inclusion of the Adobe Glyph List (AGL).[3]

Furthermore, the generated SVG files needed to be compatible with XSL-FO converters and SVG renderers, like web browsers. It turned out that each type of applications evidently focused on different aspects of the SVG standard, as some elements are not evaluated completely, leading to incorrect or incomplete visual results. To work around this, I added command-line options to alter the representation of glyphs and other graphic components in the generated SVG files as needed. The following sections cover some of the challenges involved in this area.

3.1 Vectorization of bitmap fonts

Today, many popular fonts used in (LA)TEX documents are available in OpenType, TrueType or PostScript Type 1 format, greatly simplifying their conversion to SVG as they are already vectorized. This also includes many fonts originally developed with METAFONT, such as the beautiful Old German decorative initials by Yannis Haralambous. However, during the first releases of dvisvgm, I regularly stumbled over documents that could not be converted completely because they relied on fonts only available as METAFONT source. Some of these were designed by the document authors to provide special characters or little drawings.

The problem is that although METAFONT allows detailed, high-level vectorial descriptions of glyphs, it doesn't create vector but bitmap output, in the form of GF (Generic Font) files. While it's possible to utilize bitmap fonts with SVG, the results are not satisfying, and this approach would not meet our objectives. Thus, it was necessary to find a way to

[2] `dvisvgm.sf.net/Manpage#specials`

[3] `github.com/adobe-type-tools/agl-specification`

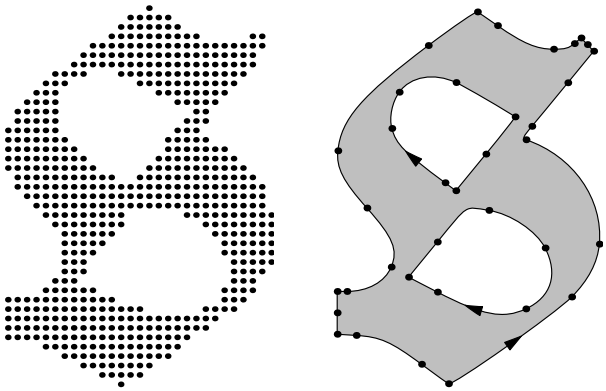


Figure 1: The Schwabacher "round s" extracted from GF font `yswab.600gf` (600 ppi) and the vectorization of the same glyph based on `yswab.2400gf`. It's composed of three closed oriented paths, with enclosed regions filled according to the non-zero rule, i.e. areas with a winding number $\neq 0$ are considered "inside".

vectorize the GF fonts during a DVI to SVG conversion without the need to perform this task separately in advance. Fortunately, there were already some open-source tracers available that could be incorporated into dvisvgm to do the hard work. Especially, the free *potrace* library [8] by Peter Selinger produces amazing results from monochromatic bitmaps, like the glyphs of GF fonts (see figure 1).

To avoid unintentional distortions of the generated paths, a high-resolution bitmap of the glyph is required. By default, dvisvgm calls METAFONT to create a GF font with a resolution of 2400 pixels per inch, which turned out to be a suitable choice, and runs potrace on the needed glyphs afterwards. The computed vector descriptions are then converted to SVG `glyph` or `path` elements and inserted into the SVG document tree. Furthermore, dvisvgm stores the vector data in a font cache located in the user's home directory to avoid repeated vectorizations of the same glyphs. When subsequently converting the same DVI file again, the glyph outlines are read from the cache, drastically increasing processing speed. Information on the data currently stored in the cache can be retrieved with the option `--cache`.

By default, dvisvgm vectorizes only the glyphs actually used on the processed DVI pages. Hence, only these are added to the cache. If the document is modified so that further, currently uncached glyphs are required, METAFONT and the vectorizer are run again to create the missing data. If desired, it's also possible to vectorize the entire GF font at once so that all its glyphs are cached instantly, with the option `--trace-all`.

While the automatic vectorization of GF fonts works pretty well, the results can't beat the manually optimized glyphs provided by native vector fonts. Thus, it has been implemented as a fallback routine only triggered if no vector version of a required font is available in the user's TEX environment.

3.2 Font elements vs. graphics paths

The initial release of dvisvgm was designed to embed font data only in terms of SVG `font`, `font-face`, and `glyph` elements, as the SVG standard provided them for this very purpose. Once defined, the glyphs can be referenced by selecting the font and using the corresponding UTF-8-encoded characters inside a `text` element, as shown in the following SVG excerpt. Due to the elaborate nature of the XML syntax, only a single, relatively short `glyph` element defining a period is shown here, abridged:

```
<font id="yswab" horiz-adv-x="0">
  <font-face ascent="751" descent="249"
    font-family="yswab" units-per-em="1000"/>
  ...
  <glyph d="m149 151l-59 -59c-7 -7 -15 -14 -20
    -23l5 -8l74 -74h1l59 59c7 7 15 14 20 23l-5
    8l-75 74z" glyph-name="period" unicode="."
    horiz-adv-x="306" vert-adv-y="306"/>
  ...
</font>
...
<text font-family="yswab" font-size="14.35"
  x="50" y="100">Hello.</text>
```

The advantage of this method is that the SVG file contains both the textual information and the appearance of the characters. This enables SVG viewers to provide features such as text search and copying, which works nicely with Apache's *Squiggle* viewer, for example. Conversion tools, like the XSL-FO processor FOP, are written to maintain the text properties and can propagate them to other file formats. On the other hand, the big disadvantage is that few SVG renderers actually support `font` elements. In particular, all popular web browsers come with partial SVG support — and none of them evaluate fonts defined as shown above. Therefore, the displayed text is selectable and searchable but very likely does not look as expected (see figure 2). As stated by Daan Leijen [6], this is an irritating problem for applications like his authoring system *Madoko*, which would like to embed math formulas in terms of SVG files into HTML documents.

A workaround for this issue is to forgo font and character information and to convert the glyphs to plain graphic objects in the form of `path` elements. These are correctly processed by all SVG renderers and lead to the desired visual results. If dvisvgm is called with the option `--no-fonts`, the above

Figure 2: Screenshots of two SVG files opened in Firefox. Both were generated from the same DVI file. The left image uses `font`, the right `path` elements.

example is transformed to the following sequence of SVG elements:

```
<defs>
  ...
  <path d="m2.14 -2.17l-0.85 0.85c-0.1 0.1
    -0.22 0.2 -0.29 0.33l0.07 0.11l1.06 1.06
    h0.01l0.85 -0.85c0.1 -0.1 0.22 -0.2 0.29
    -0.33l-0.07 -0.1l-1.08 -1.06z" id="g2-46"/>
  ...
</defs>
...
<use x="50" y="100" xlink:href="#g2-72"/>
<use x="59.8" y="100" xlink:href="#g2-101"/>
<use x="64.3" y="100" xlink:href="#g2-108"/>
<use x="70.5" y="100" xlink:href="#g2-108"/>
<use x="71.1" y="100" xlink:href="#g2-111"/>
<use x="77.8" y="100" xlink:href="#g2-46"/>
```

Based on font parameters like the partition of the em square and the font size, all glyph descriptions are now condensed to isolated graphics path objects tagged with a unique identifier. The latter is utilized to reference the object through `use` elements in order to place instances of it at the appropriate positions. Although the resulting SVG files no longer contain textual information, the visual outcome is indistinguishable from the correctly rendered font data, while simultaneously maintaining high portability across SVG renderers.

3.3 Generating WOFF fonts

Although the conversion of glyphs to graphics paths leads to satisfying visual results, the lack of access to the text is a considerable disadvantage. Fortunately, all main web browsers come with full-featured support of WOFF, WOFF2, and TrueType fonts. The CSS rule `@font-face` allows linking the name of a font family with a font file, which may be either referenced by its name, or completely embedded into the CSS code in terms of base64-encoded data.

As of version 2, dvisvgm provides the option `--font-format` to select between several different formats. Currently, it accepts the arguments `woff`, `woff2`, `ttf`, and (the default) `svg`. Similar to the treatment of SVG fonts, all data of the newly supported font formats is embedded into the SVG files in order to maximize portability. The alternative approach, to reference local font files already present on the user's system, would clearly significantly decrease the size of the generated files, but is avoided at present due to a couple of drawbacks. Especially, the fact that SVG relies on Unicode tables provided by the font files which don't necessarily have to cover all glyphs present in the font is a problem. If the Unicode table doesn't define a mapping for a certain glyph, it is inaccessible from the SVG document. This turns out to be the case for many displaystyle math operators or character variants defined by several fonts, like the XITS math font, for example. Thus, dvisvgm derives a new font from the original one and assigns random code points in the Unicode Private Use Area to the "hidden" glyphs. The resulting file is then embedded into the corresponding SVG file. For a future version, it might be a nice feature to create external font files containing all glyphs required for the entire DVI document and then reference them inside the various SVG files.

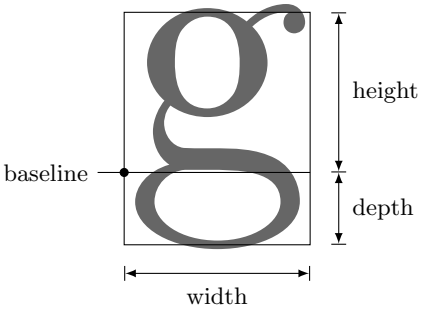
4 Bounding boxes

In contrast to DVI converters like *dvips* or *dvipdfm(x)* which are usually utilized to create self-contained final documents, the main application scenario of dvisvgm is to generate graphics files to be embedded into other documents, like web sites, EPUB or XSL-FO files. A typical example is the alignment of mathematical formulas typeset by TEX with the text of an HTML page. Therefore, the generated SVG graphics normally should get a minimal bounding rectangle that tightly encloses all visible parts without surrounding space so that the spacing and positioning of the graphics can be easily controlled inside the main document. Additional static margins present in the SVG file would make this more difficult. For this reason, dvisvgm computes tight bounding boxes for all converted pages. If a different bounding box is needed, though, the option `--bbox` can be used to add additional space around graphics (e.g. `--bbox=5pt`), to set an arbitrary box by specifying the coordinates of two diagonal corners, or to assign a common paper format, e.g. A4 or letter (e.g. `--bbox=letter`).

4.1 Tight text boxes

In order to compute tight bounding boxes, the converter requires information on the extents of each glyph present on the current page. The easiest way to get them is either to read the corresponding values directly from the font file or to use the width, height,

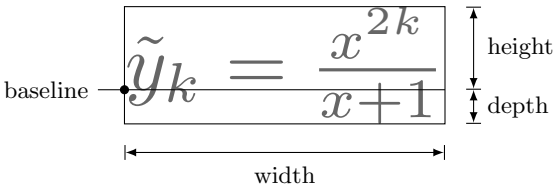
and depth values stored in a font's TFM (TEX Font Metrics) file. dvisvgm always prefers the latter if possible, because the TFM data tends to be more precise and usually leads to better results. This approach isn't perfect either, though. TFM files are primarily designed to provide TEX's algorithms with the font metrics needed to determine the optimal character positions of the processed document. The actual shapes of the characters don't matter for these computations and are in fact never seen by TEX. Furthermore, the character boxes defined by the width, height, and depth values don't have to enclose the characters' glyphs tightly. The boxes are especially allowed to be smaller so that parts of the glyphs can exceed their box as the top and bottom areas of the letter in the following example:



Obviously, this box is also somewhat wider than the enclosed glyph. Depending on the amount of divergence, the computation of the global SVG bounding box based on these values may eventually lead to visibly cropped characters and/or unwanted space at the borders of the generated graphics. That's why dvisvgm's option `--exact` was implemented some time ago. It tells the converter to trace the outlines of each glyph present on the page and to calculate their exact bounds. The path descriptions required for this task are taken from one of the available vector font files or, as a fallback, from the results of the above mentioned vectorization of METAFONT's bitmap output. While slightly more time-consuming, this approach works pretty well and helps to avoid the described text-related bounding box issues.

4.2 Aligning baselines

Another common problem that needs attention when embedding graphical LATEX snippets in e.g. HTML documents is the alignment of the baselines. Since the total height of the generated graphics comprises the height and depth of the shown text, the graphics must be shifted down by the depth value in order to properly line up with the surrounding HTML text. The vertical position can be changed with CSS property `vertical-align`, but how do we get the required depth values? Unfortunately, plain DVI files don't provide any high-level information such as typographic data. They essentially contain only the positions of single characters and rules. Thus, it's not easily possible to extract the baseline position in a reliable way.[4] Especially, two-dimensional math formulas with characters at several vertical positions, as shown in the figure below, are difficult to analyze at the DVI level without further assistance from TEX.



One helpful tool to work around this limitation is the *preview* package by David Kastrup. Particularly, its package option `tightpage` [5, p. 4] enriches the DVI file with additional data regarding height and depth which allows computing the vertical coordinate of the baseline. dvisvgm uses this information to calculate height and depth of the previously determined tight bounding box, which usually differs from the box extents provided by the preview data due to further preview settings, such as the length `\PreviewBorder`. The resulting box values are printed to the console and can be read by third-party applications afterwards to adjust the embedded SVG graphics accordingly. This happens automatically without the need to request this information explicitly. For instance, the conversion of the unscaled above formula typeset in 10 pt size leads to the following additional output:

```
width=39.02pt, height=10.43pt, depth=4.28pt,
```

where the unit `pt` denotes TEX points (72.27 pt = 1 in). If dvisvgm should apply the original, unmodified `tightpage` extents present in the DVI file, the command-line option `--bbox=preview` can be specified. Of course, the length values reported to the console then change appropriately as well.

It's important to consider that the extraction of the `tightpage` data requires a dvisvgm binary with enabled PostScript support (see section 6) because the preview package adds the box extents in terms of PostScript specials to the DVI file. If PostScript support is disabled for some reason, dvisvgm prints a

[4] There are some tricks to detect line breaks and the probable locations of the new starting baselines. One approach is to check the height of the DVI stack every time the virtual DVI cursor is moved by a positional operation. If the stack height underruns a certain threshold, a line break most likely occurred. While this technique works well for splitting hyperlink markings for example, it doesn't work reliably enough to derive the true baseline positions.

corresponding warning message and silently ignores the preview information and behaves as if no preview data were present.

A limitation of the current baseline computation is the restriction to unrotated single-line graphics. Graphics showing multiple lines of text are usually difficult to align with surrounding text and need special treatment not presently covered. The depth of such graphics is currently set to the depth of the lowest line, whereas everything above extends into the height part of the box.

4.3 `papersize` specials

Another way to define the size of the bounding rectangle is to add `papersize` specials to the TEX file, e.g. `\special{papersize=5cm,2.5cm}`, where the two comma-separated lengths denote width and height of the page. Since it's not very practical to manually enrich the documents with these commands, a couple of packages like *standalone* are available that compute the extents according to the page content and insert the specials transparently. Once present in the DVI file, dvisvgm can be told to evaluate the `papersize` specials and to apply the given extents as bounding box to the generated SVG files. Due to compatibility reasons with previous releases, this doesn't happen automatically but must be enabled with the option `--bbox=papersize`.

While the meaning of the `papersize` special itself is almost unambiguous and documented in the dvips manual, the semantics of multiple instances of the special present on the same page is not explicitly specified. Indeed, as recently discussed on the TEX Live mailing list, different DVI processors handle sequences of these specials differently. For example, dvips used to pick the first one on the page and ignore the rest, whereas dvipdfmx and others apply the last one — which, unsurprisingly, leads to different results. Since several popular packages, notably *hyperref* and *geometry*, insert `papersize` specials, it's likely that DVI pages often contain more than one and the user might stumble over this inconsistency at times. As of version 5.997 (2017), dvips got the new option `-L` to tweak this behavior. By default, it now also uses the last special, corresponding to `-L1`, whereas `-L0` restores the old behavior. dvisvgm's `papersize` support became available only after this unification effort and could therefore respect the preferred semantics without breaking previous behavior. So, it also always uses the last special present on a DVI page.

A further property of `papersize` specials is their global scope. Once applied, the size settings affect not only the current but also all subsequent pages until another `papersize` is seen. Thus, if all pages should have the same size, it's sufficient to specify the special only once at the beginning of the document.

5 Evaluation of specials

Although DVI is a very compact binary format to describe the visual layout of a typeset document, it is rather limited regarding the types of objects that can be placed on a page — only characters and solid rectangles are supported natively. Color, rotated text, graphics, hyperlinks and other features to enrich the documents are not covered by the format specification. To handle this, the DVI standard provides an operation called *xxx* which corresponds to TEX's `\special` command. It has no inherent semantics but merely holds the expanded, usually textual, argument of a `\special` command passed from the TEX document to the DVI file. Since it also doesn't affect the state of the DVI engine, each DVI driver is allowed to decide whether to evaluate any of the *xxx* operations or to ignore them altogether. Based on this mechanism, authors of LATEX packages and DVI processors can specify various special commands with defined syntax and semantics to enhance the capabilities of plain DVI documents, as already seen in the previous section about `papersize` specials.

Over the decades of TEX use, many sets of specials have been introduced. Some are well established and used by various packages. These include, among others, the color, hyperref, and PostScript specials. The recent version of dvisvgm supports these, as well as PDF font map specials, tpic specials, and the line drawing statements of the emTEX specials. To check the availability of a certain special handler in the current version of dvisvgm, the option `--list-specials` can be used. It prints a short summary of the supported special sets. It's also possible to ignore some or all specials during a DVI conversion with option `--no-specials`; this accepts an optional list of comma-separated handler names, which are identical to those listed by `--list-specials`, in order to disable only selected specials. For example, `--no-specials=color,html` disables the processing of all color and hyperref specials.

While a detailed description of all supported special commands is beyond the scope of this article, the following sections give some brief information on the hyperref and dvisvgm specials which might be helpful to know. Aspects of the PostScript handler are addressed in the subsequent section 6.

5.1 hyperref specials

The hyperref package provides commands to add hyperlinks to a LATEX document. Depending on the selected driver, it produces code for dvips, dvipdfmx,

`--linkmark=`⟨*style*⟩	visual result
`box` (default)	linked text
`box:blue`	linked text
`line`	linked text
`line:#00ff00`	linked text
`yellow`	linked text
`yellow:violet`	linked text
`none`	linked text

Table 2: Examples showing the visual effect of `--linkmark` on hyperlinked texts.

XeTeX, or any of the many other supported targets. In order to create hyperlink specials understood by dvisvgm, hyperref must be told to emit "HyperTeX" specials, with the package option `hypertex`.

By default, a linked area in the SVG file is highlighted by a box drawn around it in the currently selected color. On the request of several users, the option `--linkmark` was added to allow changing this behavior. It requires an argument determining the style of the marking. While `box` is the default, argument `line` underlines the clickable area rather than framing it, and `none` suppresses any visual highlighting of hyperlinks completely. A dvips color name or hexadecimal RGB value appended to these styles and separated by a colon, assigns a static color to the box or line. Finally, a style argument of the form *`color1:color2`* leads to a box filled with *color1* and framed with *color2*. Table 2 shows some examples to give an idea of the effect of the style arguments.

5.2 dvisvgm specials

Besides the mentioned sets of special commands, dvisvgm also provides some of its own to allow authors of LaTeX packages to insert additional SVG fragments into the generated files and to interact with the computation of the bounding box. Their general syntax looks like this:

`\special{dvisvgm:`⟨*cmd*⟩ ⟨*params*⟩`}`

The *cmd* denotes the command name and *params* the corresponding parameters. For the sake of simplicity, only the text after the colon is mentioned when referring to dvisvgm specials herein.

The command `raw` followed by arbitrary text appends the text to the group element representing the current page. The sibling command `rawdef` does almost the same but appends the text to the initial `defs` element present at the beginning of the SVG file. Both specials are allowed to insert any string and thus can contain XML metacharacters, such as angle brackets, e.g.:

```
raw <circle cx="{?x}" cy="{?y}" r="5"/>.
```

The macros `{?x}` and `{?x}` expand to the x and y coordinate of the current DVI position in the "big point" units (72 bp = 1 in) required in SVG files. The entire character string is then copied to a literal text node of the SVG tree and not evaluated further. Therefore, it's crucial to ensure that the insertions don't break the validity of the resulting SVG document, especially if multiple `raw` or `rawdef` commands are used to assemble complex element structures.

Another aspect to take care of regarding raw insertions is the adaptation of the bounding box. As outlined in section 4, dvisvgm computes a tight bounding box for the generated SVG graphics by default. Graphical or textual elements inserted via the raw commands are not taken into account. As a consequence, the bounding box may be too small, so that some parts of the graphic lie outside the viewport. To work around this, dvisvgm offers a special that allows for intervening in the calculation of the bounding rectangle. The command

`bbox` ⟨*width*⟩ ⟨*height*⟩

updates the bounding box so that a virtual rectangle of the given width and height and located at the current DVI position will be fully enclosed. It's also possible to append an optional *depth* parameter to the command:

`bbox` ⟨*width*⟩ ⟨*height*⟩ ⟨*depth*⟩

This encloses another rectangle of the same width but with the negative height *depth*. At present, the dimensions must be given as plain floating point numbers in TeX pt units without a unit specifier. In a future release, it will be possible to use the various common length units to ease usage of this command. For example, to update the bounding box for the above raw circle element, the two successive dvisvgm specials `bbox 5 5 5` and `bbox -5 5 5` can be used.

In addition to these relative bounding box specials, two absolute variants are supported, which are only briefly mentioned here. More details about them can be found on the manual page.

`bbox abs` ⟨x_1⟩ ⟨y_1⟩ ⟨x_2⟩ ⟨y_2⟩
`bbox fix` ⟨x_1⟩ ⟨y_1⟩ ⟨x_2⟩ ⟨y_2⟩

The first variant encloses a virtual rectangle given by the coordinates (x_1, y_1) and (x_2, y_2) of two diagonal corners, whereas the second one sets the final coordinates of the SVG bounding box, which will not be changed or reset afterwards.

6 PostScript support

One of the biggest enhancements of the DVI format was certainly the introduction of PostScript specials and their processing by Tomas Rokicki's dvips. Besides placing advanced drawings in TeX documents,

it supports injecting code between DVI commands, allowing for the implementation of text transformations, coloring and much more. While dvips can copy the code of the PostScript specials almost literally to the generated files and delegate their processing to the PostScript interpreter, DVI drivers targeting a different output format have to evaluate it somehow.

The implementation of a full-featured PostScript interpreter was certainly out of the scope of dvisvgm. However, I wanted the utility to be able to properly convert as many DVI files as possible, ideally including ones created using PSTricks or Ti*k*Z. The most straightforward approach to achieve this was to delegate the complex processing of PostScript code to the free PostScript interpreter *Ghostscript* and let it emit a reduced set of easily parsable statements that dvisvgm could evaluate. This turned out to work reasonably well, especially as a fair amount of PostScript code can completely be processed by Ghostscript without the need to worry about the involved operations. Only a relatively small set of operators that affect the graphics state must be overridden and forwarded to dvisvgm in order to create appropriate SVG components or to update drawing properties.

In contrast to the other programming libraries dvisvgm relies on and which are directly linked into the binary, the Ghostscript library (`libgs`) can be tied to dvisvgm in two different ways. Besides disabling PostScript support completely, it's possible to either link to the Ghostscript library directly, or to load it dynamically at runtime. In the first case, PostScript support is always enabled, while in the second one it depends on the accessibility of the Ghostscript library on the user's system. If `libgs` can't be found or accessed for some reason, dvisvgm prints a warning message and disables the processing of PostScript specials, which of course will likely lead to inaccurate conversion results. To help dvisvgm locate the library, the option `--libgs` or environment variable `LIBGS` can be used, e.g. to specify the absolute path of the correct file. More detailed information on this topic can be found on the FAQ page of the project website.[5]

Although dvisvgm can properly convert a fair amount of PostScript code, there are still some operators and features it does not support yet. These include all bitmap-related operations as well as linear, radial, and function-based shading fills. Furthermore, text output triggered by PostScript code is always converted to SVG `path` elements similar to those described in section 3.2. The differentiated handling of fonts including the conversion to WOFF only works in conjunction with DVI font definitions.

6.1 Handling clipping path intersections

In order to restrict the area where drawing commands lead to visible results, SVG allows the definition of *clipping paths*. Every clipping path is defined by a set of closed vector paths consisting of an arbitrary number of straight and curved line segments. The regions enclosed by these paths define the visible area, i.e. after applying a clipping path, only those portions of the subsequently drawn graphics that fall inside the enclosed area are visible, while everything else is discarded. Clipping is a basic functionality of computer graphics and supported by various formats and languages, like PostScript, Asymptote, METAPOST, and Ti*k*Z. So why is it mentioned here? Because one variant of defining clipping paths in SVG may lead to unpredictable, flawed visual results due to absent or incomplete support in SVG renderers.

Besides defining the clipping path explicitly, which is nicely supported by almost all renderers I know of, it's also possible to tell the SVG renderer to compute the intersection of two or more paths and restrict the subsequent drawing actions to the resulting area. The following example defines a lens-shaped path called *lens* by combining two arcs of 90 degrees. The result is assigned to clipping path *clip1*. The second clipping path *clip2* reuses path *lens* but rotated by 90 degrees clockwise around its center.

```
<clipPath id="clip1">
  <path id="lens" d="
    M 0 0
    A 50 50 0 0 1 50 50
    A 50 50 0 0 1 0 0 Z"/>
</clipPath>
<clipPath id="clip2" clip-path="url(#clip1)">
  <use xlink:href="#lens"
    transform="rotate(90,25,25)"/>
</clipPath>
```

The crucial part of this definition is the `clip-path` attribute, which restricts the drawing area of *clip2* to the interior of *clip1* so that the resulting clipping region leads to a curved square, as shown in figure 3.

Graphic elements restricted to *clip2*, like the following rectangle, are now supposed to be clipped at the border of this square.

```
<rect x="17" y="0" width="16" height="50"
  clip-path="url(#clip2)"/>
```

Unfortunately, this isn't the case with all SVG renderers.[6] Since successive calls of the PostScript operators `clip` and `eoclip` cause consecutive path intersections, which dvisvgm translates to `clipPath`

[5] `dvisvgm.sf.net/FAQ`

[6] Examples can be seen at `dvisvgm.sf.net/Clipping`.

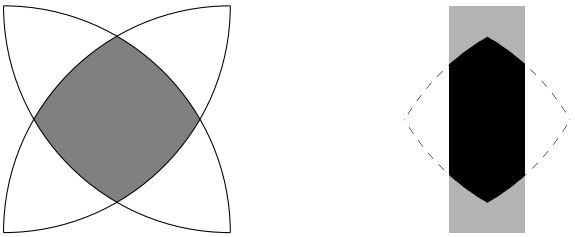


Figure 3: Intersection of two lens-shaped paths (left), and a rectangle clipped on the resulting area.

elements with `clip-path` attributes by default, the generated SVG files are not portable either. In order to prevent the creation of these, the `--clipjoin` option was added some time ago. It tells dvisvgm to compute the path intersections itself with the help of Angus Johnson's great *Clipper* library[7], which provides an implementation of the Vatti polygon clipping algorithm. For this purpose, dvisvgm approximates all clipping paths by polygons, runs the Vatti algorithm on them to compute the boundaries of the intersection areas, and reconstructs the curved segments of the resulting paths afterwards. In this way, we usually get a compact yet smoothly approximated outline of the final clipping paths. The application of option `--clipjoin` to clipping path *clip2* of the above example leads to the following self-contained path definition composed of four cubic Bézier curve segments:

```
<clipPath id="clip2">
  <path d="
    M 43.2 25
    C 38.8 32.5 32.5 38.8 25 43.2
    C 17.5 38.8 11.2 32.5 6.8 25
    C 11.2 17.5 17.5 11.2 25 6.8
    C 32.5 11.2 38.8 17.5 43.2 25 Z"/>
</clipPath>
```

6.2 Approximation of gradient fills

One of the more powerful and impressive PostScript features is the advanced support of various shading algorithms to fill a region with smooth transitions of colors in several color spaces. These algorithms include Gouraud-shaded triangle meshes, tensor-product patch meshes, and flexible function-based shadings, as well as linear and radial gradients. The current SVG standard 1.1 provides elements to specify gradient fills too but they are limited to the last two mentioned above, and are furthermore somewhat less flexible than the PostScript equivalents. Therefore, it's not possible to map arbitrary gradient definitions present in EPS files or PostScript specials to plain SVG gradient elements. In order to nonetheless convert a subset of them, dvisvgm approximates color gradients by filling the specified area with small monochromatic segments as shown in figure 4.

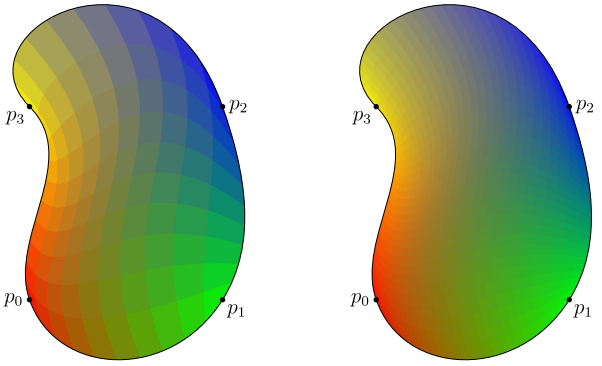


Figure 4: Approximation of tensor product shading using a grid of 10×10 and 30×30 color segments, respectively.

Each segment gets the average color of the covered area according to the selected gradient type and color space. The maximum number of segments created per column or row can be changed by option `--grad-segments`. Greater values certainly lead to better approximations, but concurrently increase the computation time, the size of the SVG file, and, perhaps most important, the effort required to render the file. To slightly counteract this drawback, dvisvgm reduces the level of detail if the extent of the segments falls below a certain limit. In case of tensor-product patches, the segments are usually delimited by four cubic Bézier curves and will then be simplified to quadrilaterals. The limit at which this simplification takes place can be set by option `--grad-simplify`.

An issue that can occur in conjunction with gradient fills is the phenomenon of visible gaps between adjacent segments, even though they should touch seamlessly according to their coordinates. This effect results from the anti-aliasing applied by most SVG renderers in order to produce smooth segment contours which usually takes not only the foreground but also the background color into account. Therefore, the background color becomes visible at the joints of the segments. If desired, the option `--grad-overlap` can be used to prevent this effect. It tells dvisvgm to create bigger, overlapping segments that extend into the region of their right and bottom neighbors. Since the latter are drawn on top of the overlapping parts, which now cover the former joint lines, the visible size of all segments remains unchanged. In this manner we get visual results similar to those shown in figure 4.

[7] `angusj.com/delphi/clipper.php`

6.3 Converting EPS files to SVG

Besides the family of special commands provided to embed literal PostScript code directly into DVI files, dvips also introduced a special called `psfile`. Its purpose is to reference an external PS or EPS file and insert its content, possibly after some processing, at the current location of the document. The LaTeX command `\includegraphics` from the *graphicx* package, for instance, produces a `psfile` special if the dvips driver is selected. Also, the vector graphics language *Asymptote* [1] uses this special in its intermediate DVI files to combine the graphical and typeset components of the resulting drawings. Because of these major application areas, it was important to make dvisvgm capable of processing `psfile` specials, in order to cover a broader range of documents.

Since the technical details of the command are probably not of much interest for general users, they are not discussed here in more depth. However, one nice bonus feature that was technically available instantly after finishing the implementation of the `psfile` handler can be mentioned. Due to the handler being able to process separate files, it seemed natural to make this functionality available through the command-line interface and so provide an EPS to SVG converter. Little additional code was required to realize this. Thus, as of version 1.2, dvisvgm offers option `--eps` which tells the converter not to expect a DVI but an EPS input file and to convert it to SVG. For example,

```
dvisvgm --eps myfile
```

produces the SVG file `myfile.svg` from `myfile.eps`. This is implemented by creating a single `psfile` special called together with the bounding box information given in the EPS file's DSC header. To do the conversion, only the PostScript handler is required, with none of the DVI-related routines and associated functionality, like font and other special processing.

Since there are already some standalone utilities like ImageMagick and Inkscape available that can do this, dvisvgm's EPS to SVG functionality is probably needed less frequently but might nonetheless be beneficial for the TeX community.

7 Acknowledgments

I would like to thank Karl Berry, Mojca Miklavec, and, posthumously, Peter Breitenlohner for their invaluable work to make dvisvgm available in TeX Live and help in tracking down several issues. Also, thank you to Khaled Hosny for implementing the command-line option `--no-merge` and for providing a Python port of my formerly XSLT-based helper script *opt2cpp*. I furthermore appreciate the amazing work of John Bowman and Till Tantau who added support of dvisvgm to Asymptote and Ti*k*Z/PGF, respectively.

There are many more people whom I can't list here individually but who helped enormously to improve the program by reporting bugs, providing code, and sending feature suggestions. Thank you very much to all of you.

References

[1] John Bowman. Asymptote: Interactive TeX-aware 3D vector graphics. *TUGboat*, 31(2):203–205, 2010. `tug.org/TUGboat/tb31-2/tb98bowman.pdf`.

[2] Adrian Frischauf and Paul Libbrecht. dvi2svg: Using LaTeX layout on the web. *TUGboat*, 27(2):197–201, 2006. `tug.org/TUGboat/tb27-2/tb87frischauf.pdf`.

[3] Martin Gieseking and Oliver Vornberger. media2mult: A wiki-based authoring tool for collaborative development of multimedial documents. In Miguel Baptista Nunes and Maggie McPherson, editors, *Proceedings of the IADIS International Conference on e-Learning*, pages 295–303, Amsterdam, Netherlands, 2008.

[4] Michel Goossens and Vesa Sivunen. LaTeX, SVG, Fonts. *TUGboat: The Communications of the TeX Users Group*, 22(4):269–280, 2001. `tug.org/TUGboat/tb22-4/tb72goos.pdf`.

[5] David Kastrup. The `preview` package for LaTeX. `ctan.org/pkg/preview`, April 2017.

[6] Daan Leijen. Rendering mathematics for the web using Madoko. In Robert Sablatnig and Tamir Hassan, editors, *Proceedings of the 2016 ACM Symposium on Document Engineering*, pages 111–114, Vienna, Austria, 2016. `www.microsoft.com/en-us/research/wp-content/uploads/2016/08/doceng16.pdf`.

[7] Rudolf Sabo. DVISVG. Master's thesis, Masarykova Univerzita, Brno, Czech Republic, 2004. `dvisvg.sourceforge.net/dipl.pdf`.

[8] Peter Selinger. Potrace: A polygon-based tracing algorithm. `potrace.sourceforge.net`, 2003.

◇ Martin Gieseking
University of Osnabrück
Heger-Tor-Wall 12
49074 Osnabrück, Germany
`martin dot gieseking (at) uos dot de`