The animate Package

Alexander Grahn [*]

https://gitlab.com/agrahn/animate

23rd March 2020

Abstract

A LaTeX package for creating portable, JavaScript driven PDF and SVG animations from sets of vector graphics or raster image files or from inline graphics.

Keywords: include portable PDF animation SVG animation animated PDF animated SVG dvisvgm html TeX4ht web animating embed animated graphics LaTeX pdfLaTeX LuaLaTeX PSTricks pgf TikZ LaTeX-picture MetaPost inline graphics vector graphics animated GIF LaTeX dvips ps2pdf dvipdfmx XeLaTeX JavaScript Acrobat Reader PDF-XChange Foxit Reader Firefox Chrome Chromium

Contents

[*] Animated GIF taken from phpBB forum software and burst into a set of EPS files using ImageMagick before embedding.

1 Introduction

This package provides an interface for creating PDF and SVG files with animated content from sets of graphics or image files, from inline graphics, such as LaTeX-picture, PSTricks or pgf/TikZ generated pictures, or just from typeset text. Unlike standard movie/video formats, package 'animate' allows for animating vector graphics. The result is roughly similar to the SWF (Flash) format, although not as space-efficient.

Package 'animate' supports the usual workflows for making PDF, i. e. pdfLaTeX, LuaLaTeX, LaTeX → `dvips` → `ps2pdf`/Distiller and (Xe)LaTeX → `(x)dvipdfmx`. For animated SVG, it supports the `dvisvgm` driver. The DVI/XDV used as input for `dvisvgm` can be generated with LaTeX, LuaLaTeX (in DVI mode) and XeLaTeX.

PDF files with animations can be viewed in Acrobat Reader (except on mobile devices), PDF-XChange and Foxit Reader. Animated SVG produced by means of LaTeX and `dvisvgm` are self-contained files that can be embedded into HTML using the `<object>` tag or opened directly in a Web browser, such as Chromium or Firefox.

The user interacts with the animation through optional animation controls or using the mouse like so: pressing the mouse button over the animation widget immediately pauses a playing animation and releasing it resumes playback. Pressing the shift-key at the same time reverses the playback direction. Keeping the mouse button pressed while moving the mouse pointer off the animation widget permanently pauses playback.

On mobile devices, interaction with an animated SVG is similar to that on desktop devices, just using finger touches instead of mouse button press/release. To permanently pause an animation without the pause button, touch the animation with a second finger, then lift both fingers off.

2 Requirements

pdfTeX, version ≥ 1.40, or LuaTeX, version ≥ 0.95, for direct PDF output

Ghostscript, version ≥ 9.15, or Adobe Distiller for PS to PDF conversion

`dvipdfmx` for DVI to PDF conversion

dvipdfmx, version ≥ 20190503, for DVI to PDF conversion

`dvisvgm` for DVI to SVG conversion (also requires Ghostscript)

PDF: Acrobat Reader, PDF-XChange, Foxit Reader

SVG: Chromium-based browsers, Firefox and others

3 Installation

Unzip the file 'animate.tds.zip' into the local TDS root directory which can be found by running '`kpsewhich -var-value TEXMFLOCAL`' on the command line.

After installation, update the filename database by running '`texhash`' on the command line.

TEX-Live and MiKTEX users should run the package manager of their system for installation.

4 Using the package

First of all, read Section 10 on problems related to this package. Then, invoke the package by putting the line

```
\usepackage[<package options>]{animate}
```

to the preamble of the document source, i. e. somewhere between `\documentclass` and `\begin{document}`.

'animate' honours the package options:

```
dvipdfmx
xetex
dvisvgm
export
autoplay
autopause
autoresume
loop
palindrome
draft
final
controls[=(all|true|on) |
          (none|false|off) | {[play][,step][,stop][,speed]}]
controlsaligned=left[+<indent>] | center | right[+<indent>]
width=<h-size>
height=<v-size> | totalheight=<v-size>
keepaspectratio
scale=<factor>
hiresbb
interpolate
pagebox
buttonsize=<size>
buttonbg=<colour>
buttonfg=<colour>
buttonalpha=<opacity>
step
nomouse
type=<file ext>
method=icon | widget | ocg
```

```
poster[=first | <num> | last | none]
alttext=none | {<alternative description>}
```

Except for ‘dvipdfmx’, ‘xetex’, ‘dvisvgm’ and ‘export’, the listed package options are also available (among others) as command options and will be explained shortly. However, if used as package options they have global scope, taking effect on all animations in the document. In turn, command options locally override global settings. Options without an argument are boolean options and can be negated, with the exception of package-only options ‘dvipdfmx’, ‘xetex’, ‘dvisvgm and ‘export’, by appending ‘=false’.

Important: Option ‘-Ppdf’ should *not* be used with dvips when converting DVI to PostScript. If you cannot do without, put ‘-D 1200’ *after* ‘-Ppdf’ on the command line. Users of LaTeX-aware text editors with menu-driven toolchain invocation, such as TeXnicCenter, should check the configuration of the dvips call.

All workflows require the ‘graphicx’ package to be loaded explicitly.

Options ‘dvipdfmx’ or ‘dvisvgm’ must be set for the document class, as in

```
\documentclass[dvipdfmx,...]{...}
```

or

```
\documentclass[dvisvgm,...]{...}
```

because these drivers cannot be auto-detected by ‘animate’, ‘graphicx’ and other packages.

Usually, a second LaTeX run is necessary to resolve internally created object references. A warning message will be issued if appropriate.

With option ‘export’, animation frames are output as individual pages of a multipage document that can be converted to other file formats, such as animated GIF. The ‘standalone’ document class must be used together with ‘export’:

```
\documentclass{standalone}
\usepackage[export]{animate}
```

or

```
\documentclass[export]{standalone}
\usepackage{animate}
```

5 The user interface

Package ‘animate’ provides the command

```
\animategraphics[<options>]{<frame rate>}{<file basename>}{<first>}{<last>}
```

and the environment

```
\begin{animateinline}[<options>]{<frame rate>}
    ... typeset material ...
\newframe[<frame rate>]
    ... typeset material ...
\newframe*[<frame rate>]
```

```
    ... typeset material ...
\newframe
\multiframe{<number of frames>}{[<variables>]}{
  ... repeated (parameterized) material ...
}
\end{animateinline}
```

While `\animategraphics` can be used to assemble animations from sets of existing graphics files or from multipage PDF, the environment '`animateinline`' is meant to create the animation from the typeset material it encloses. This material can be pictures drawn within the LaTeX '`picture`' environment or using the advanced capabilities of PSTricks or pgf/TikZ. Even ordinary textual material may be animated in this way. The parameter `<frame rate>` specifies the number of frames per second of the animation.

The `\newframe` command terminates a frame and starts the next one. It can be used only inside the '`animateinline`' environment. There is a starred variant, `\newframe*`. If placed after a particular frame, it causes the animation to pause at that frame. The animation continues as normal after clicking it again. Both `\newframe` variants take an optional argument that allows the frame rate to be changed in the middle of an animation.

The `\multiframe` command allows the construction of loops around pictures. The first argument `<number of frames>` does what one would expect it to do, the second argument `<variables>` is a comma-separated list of variable declarations. The list may be of arbitrary, even zero, length. Variables may be used to parameterize pictures which are defined in the loop body (third argument of `\multiframe`). A single variable declaration has the form

```
<variable name>=<initial value>+<increment>
```

`<variable name>` is a sequence of one or more letters *without* a leading backslash[1]. The first (and possibly only) letter of the variable name determines the type of the variable. There are three different types: integers ('`i`', '`I`'), reals ('`n`', '`N`', '`r`', '`R`') and dimensions or LaTeX lengths ('`d`', '`D`'). Upon first execution of the loop body, the variable takes the value `<initial value>`. Each further iteration increments the variable by `<increment>`. Negative increments must be preceded by '`-`'. Here are some examples: '`i=1+2`', '`Rx=10.0+-2.25`', '`dim=20pt+1ex`'. Within the loop body, variables are expanded to their current value by prepending a backslash to the variable name, that is `\i`, `\Rx` and `\dim` according to the previous examples. `\multiframe` must be surrounded by `\begin{animateinline}` and `\end{animateinline}` or by any of the `\newframe` variants. Two consecutive `\multiframe` commands must be separated by one of the `\newframe` variants.

By default, the animation is built frame by frame in the order of inclusion of the embedded material. However, extended control of the order of appearance, superposition and repetition of the material is available through the '`timeline`' option (see Section 6.2).

[1] This is different from `\multido` (package 'multido') where variable names have a leading '`\`' in the declaration.

Sets of graphics files
All files of the sequence should exist and be consecutively numbered. (Exception to this rule is allowed in connection with the 'every' option, see below.) <file basename> is the leftmost part of the file name that is common to all members of the sequence. <first> is the number of the first and <last> the number of the last file in the set. If <first> is greater than <last>, files are embedded in reverse order. File names may be simply numbered, such as 0 ... 99. If there are leading zeros, make sure that all file numbers have the same number of digits, such as 0000 ... 0099, and that the <first> and <last> arguments are filled in accordingly.

For example, given the sequence 'frame_5.png' through 'frame_50.png' from a possibly larger set that shall be used to build an animation running at 12 frames per second, the correct inclusion command would read

```
\animategraphics{12}{frame_}{5}{50}
```

The possible file formats depend on the output driver being used. In the case of LaTeX + dvips, files with the 'eps' extension are at first searched for, followed by 'mps' (METAPOST-generated PostScript) and 'ps'. With pdfLaTeX and LuaLaTeX the searching order is: (1) 'pdf', (2) 'mps', (3) 'png', (4) 'jpg', (5) 'jpeg', (6) 'jp2'[1], (7) 'j2k'[1], (10) 'jpx'[1], with XeLaTeX or LaTeX+dvipdfmx: (1) 'pdf', (2) 'mps', (3) 'eps', (4) 'ps', (5) 'png', (6) 'jpg', (7) 'jpeg', (8) 'bmp', and with LaTeX+dvisvgm: (1) 'pdf', (2) 'eps', (3) 'ps', (4) 'mps', (5) 'svg', (6) 'png', (7) 'jpg', (8) 'jpeg'. That is, files capable of storing vector graphics are found first. Make sure that all file names have *lower case* extensions.

This searching procedure can be skipped thanks to the package and command option 'type=<file ext>'. It enforces the embedding of files with the given file name extension <file ext>.

Command \graphicspath{} from the 'graphicx' package can be used to specify directories to be browsed for graphics files.

Multipage PDF inclusion
LuaLaTeX, pdfLaTeX, XeLaTeX and the dvipdfmx and dvisvgm backends are able to embed animation frames from multipage PDF. If the file '<file basename>.pdf' exists, it is taken as a multipage document where each page represents one frame of the animation. In this case, the last two arguments, <first> & <last>, are interpreted differently from above; they specify a zero-based range of pages to be included in the animation. Either or both of them may be omitted, '{}', in which case they default to 0 and $n-1$, where n is the total number of available pages. Arguments that fall outside this range are automatically corrected to the actual limits. If <first> is greater than <last>, pages are embedded in reverse order. Again, option 'type=<file ext>' can be used to enforce a particular file extension.

For example, the line

```
\animategraphics{12}{frames}{}{}
```

would create an animation from all pages of the file 'frames.pdf', running at 12 fps.

[1] Only LuaLaTeX currently supports JPEG2000.

6 Command options

The following options to `\animategraphics` and 'animateinline' have been provided:

6.1 Basic options

`label=<label text>`

The animation is given a label, `<label text>`, which must be unique. Labelling an animation enables its JavaScript programming interface by defining `anim['<label text>']`, which is a JavaScript reference to the animation object. The animation object provides a number of properties and methods that can be used for controlling the animation playback from within user defined JavaScript. For details, see Section 7.

`type=[<file ext>]`

Overrides the searching procedure for graphics files explained in the previous section and forces files with extension `<file ext>` to be used. Given with an empty argument as in 'type=', this option locally reinstates the default searching procedure if it was globally disabled through the package option.

`poster[=first | <num> | last | none]`

Specifies which frame to display and print if the animation is not activated. The first frame is shown by default. Thus 'poster' or 'poster=first' need not be explicitly set. A frame number `<num>` may as well be given; `<num>` is zero-based, that is, the first frame has number '0'.

`every=<num>`

Build animation from every `<num>`*th* frame only. Skipped frames are discarded and not embedded into the document. In the case of `\animategraphics`, skipped input files may be missing.

`autopause`

Pause animation when the page is closed, instead of stopping and rewinding it to the default frame.

`autoplay`

Start animation after the page has opened. Also resumes playback of a previously paused animation.

`autoresume`

Resume previously paused animation when the page is opened again.

`loop`

The animation restarts immediately after reaching the end.

`palindrome`

The animation continuously plays forwards and backwards.

`step`

Step through the animation one frame at a time per mouse-click. The <frame rate> argument will be ignored.

width=<h-size>
height=<v-size> | totalheight=<v-size>
keepaspectratio

Resize the animation widget. If only one of 'width' or '[total]height' is given, the other dimension of the animation widget is scaled to maintain the aspect ratio of the first frame's content. If both 'width' and '[total]height' are given together with 'keepaspectratio', the first frame's content is resized to fit within <h-size> and <v-size> while maintaining its original aspect ratio. Any valid TEX dimension is accepted as a parameter. In addition, the length commands \width, \height, \depth and \totalheight can be used to refer to the original dimensions of the first frame of the animated sequence.

scale=<factor>

Scales the animation widget by <factor>.

bb=<llx> <lly> <urx> <ury>

(\animategraphics only.) The four, space separated arguments set the bounding box of the graphics files. Units can be omitted, in which case 'bp' (PostScript points) is assumed.

viewport=<llx> <lly> <urx> <ury>

(\animategraphics only.) This option takes four arguments, just like 'bb'. However, in this case the values are taken relative to the origin specified by the bounding box in the graphics files.

trim=<left> <bottom> <right> <top>

(\animategraphics only.) Crops graphics at the edges. The four lengths specify the amount to be removed from or, if negative values have been provided, to be added to each side of the graphics.

pagebox
hiresbb
interpolate

(\animategraphics only.) Options from the 'graphics' bundle. Refer to the 'graphics' package manual for their meaning.

controls[=all | true | on]
controls=(none | false | off) | {[play][,step][,stop][,speed]}

Inserts control buttons below the animation widget. Visibility of buttons can be fine-tuned through optional keys. By default, if setting 'controls' alone, all available buttons are shown, while any of 'none', 'false' or 'off' suppresses them altogether. A comma-separated selection from 'play', 'step', 'stop' and 'speed' enables corresponding pairs or groups of buttons. Such a comma list must be enclosed in braces, i. e. controls={..., ..., ...}. If all buttons are shown, their meaning is as follows, from left to right: stop & first frame, step backwards, play backwards, play forwards, step forwards, stop & last frame, decrease speed, default speed, increase speed. Both 'play' buttons are replaced by a large 'pause' button while the animation is playing.

`controlsaligned=left[+<indent>] | center | right[+<indent>]`

Animation controls are centred below the animation widget by default. With this option one can choose between centred, flush-left or flush-right alignment. The optional `<indent>` inserts additional horizontal space that pushes the control buttons away from the left or right edge of the animation widget towards the opposite side. `<indent>` must be a valid TEX dimension. Negative values may be used, as in

```
controlsaligned=right+-4em,
```

in which case control buttons protrude beyond the animation's right vertical edge by `4em`.

`buttonsize=<size>`

Changes the control button height to `<size>`, which must be a valid TEX dimension. The default button height is `1.44em` and thus scales with the current font size.

`buttonbg=<colour>`
`buttonfg=<colour>`
`buttonalpha=<opacity>`

By default, control button widgets are drawn with black strokes on transparent background. The background can be turned into a solid colour by the first option, while the second option specifies the stroke colour. The parameter `<colour>` is an array of colon-(`:`)-separated numbers in the range from 0.0 to 1.0. The number of array elements determines the colour model in which the colour is defined: (1) gray value, (3) RGB, (4) CMYK. For example, '`1`', '`1:0.5:0.2`' and '`0.5:0.3:0.7:0.1`' are valid colour specifications. Option `buttonalpha` adds transparency to the control buttons. Its parameter `<opacity>` is a number between 0.0 and 1.0, where 0.0 produces fully transparent and 1.0 fully opaque buttons.

`draft`
`final`

With '`draft`' the animation is not embedded. Instead, a box with the exact dimensions of the animation is inserted. Option '`final`' does the opposite as it forces the animation to be built and embedded. Both options can be used to reduce compilation time during authoring of a document. To get the most out of them it is recommended to set '`draft`' globally as a package or class option and to set '`final`' locally as a command option of the animation that is currently being worked on. After the document has been finished, the global '`draft`' option can be removed to embed all animations.

`nomouse`

Animation widget will not respond to mouse clicks. Unless the JavaScript interface, Sect. 7, p. 15, is used to control the animation, it is recommended to also set at least one of the '`autoplay`' or '`controls`' options.

`method=icon | widget | ocg`

The package implements three different animation methods. The '`icon`' method is the default method and usually gives the best performance in terms of animation frame rate. '`widget`' and '`ocg`' are alternative animation methods. In rare cases (standalone animations without animation controls) method '`ocg`' may lead to a better animation

performance than the other two. Moreover, it allows overlaying animations with other typeset material, that is, playing animations in the page background.

`measure`

Measures the frame rate during one cycle of the animation and prints the value to the JavaScript console of the Reader. (For testing purposes.)

`alttext=none | {<alternative description>}`

Changes or suppresses alternative description text. The default is '`animation by animate[2020/03/23]`'. This text is shown by some PDF viewers that are not capable of rendering JavaScript animations or used for accessibility purposes.

`begin={<begin text>}`
`end={<end text>}`

('`animateinline`' only.) `<begin text>` and `<end text>` are inserted into the code at start and end of each frame. Mainly used for setting up some drawing environment, such as

```
begin={\begin{pspicture}(... , ...)(... , ...)},
end={\end{pspicture}}
```

A short note on the '`tikzpicture`' environment: Unlike '`pspicture`', the '`tikzpicture`' environment is able to determine its size from the graphical objects it encloses. However, this may result in differently sized frames of a sequence, depending on the size and position of the graphical objects. Thus, in order to ensure that all frames of the sequence be displayed at the same scale in the animation widget, a common bounding box should be shared by the frames. A bounding box can be provided by means of an invisible '`rectangle`' object:

```
begin={
  \begin{tikzpicture}
  \useasboundingbox (... , ...) rectangle (... , ...);
},
end={\end{tikzpicture}}
```

6.2 The '`timeline`' option

`timeline=<timeline file>`

`<timeline file>` is a plain text file whose contents determines the order of appearance of the embedded material during the animation. It allows the user to freely rearrange, repeat and overlay the material at any point of the animation. This may greatly reduce the file size of the resulting PDF, as objects that do not change between several or all frames, such as coordinate axes or labels, can be embedded once and re-used in other frames of the animation. (Technically, this is done by the XObject referencing mechanism of PDF.)

If a timeline is associated with the animation, the graphics files or inline graphics embedded by `\animategraphics` and '`animateinline`' no longer represent the actual frames of the animation. Rather, they are a collection of *transparencies* that can be played with at will. However, it is now up to the author's responsibility to construct

a timeline that makes use of *each* of those transparencies and to put them into a sensible order. In order to identify the transparencies within the timeline file, they are numbered in the order of their inclusion, starting at zero.

A timeline-based animation can be thought of as a *living stack* of translucent transparencies. Each animation frame is a snapshot of the stack viewed *from above*. Transparencies are usually put on top of that stack and stay there for a given number of frames before expiring (becoming invisible). The lifetime of each transparency within the stack can be set individually. Once expired, a transparency can be put on the stack again, if desired. The stack may also be divided into an arbitrary number of *sub-stacks* to facilitate the creation of layers, such as background, foreground and intermediate layers. Sub-stacks allow the insertion of transparencies at depth positions of the global stack other than just the top. It is important to keep the stack-like nature of animations in mind because graphical objects on transparencies at higher stack positions overlay the content of transparencies at lower stack positions.

General structure of the timeline file
Each line of the timeline file that is not blank and which does not begin with a comment (‘%’) corresponds to *one* frame of the animation. There may be more transparencies than animation frames and vice-versa. A frame specification consists of three or four colon-(:)-separated fields:

```
[*]:[<frame rate>]:[<transparencies>][:<JavaScript>]
```

While any field may be left blank, the first two colons are mandatory. The fourth field, <JavaScript>, is explained on p. 14.

An asterisk (‘*’) in the leftmost field causes the animation to pause at that frame, very much as a \newframe* would do; a number in the second field changes the frame rate of the animation section that follows. In connection with the ‘timeline’ option, the asterisk extension and the optional <frame rate> argument of \newframe cease to make sense and will be tacitly ignored if present.

The third field <transparencies> is a comma-separated *list* of *transparency specifications* that determines the transparencies to be put on the stack. Semicolons (;) are used to separate sub-stacks (= layers) from each other. A *single* transparency specification obeys the syntax

```
<transparency ID>[x<number of frames>]
```

where <transparency ID> is an integer number that identifies the transparency to be drawn into the current animation frame. As pointed out above, the transparencies are consecutively numbered in the order of their inclusion, starting at zero. The optional postfix ‘x<number of frames>’ specifies the number of consecutive frames within which the transparency is to appear. If omitted, a postfix of ‘x1’ is assumed, which causes the transparency to be shown in the current frame only. Obviously, <number of frames> must be a non-negative integer number. The meaning of postfix ‘x0’ is special; it causes the transparency to be shown in all frames, starting with the current one, until the end of the animation or until the animation sub-stack to which it belongs is explicitly cleared.

The letter ‘c’, if put into <transparencies>, clears an animation sub-stack, that is, it causes all transparencies added so far to be removed from the sub-stack, overriding any <number of frames> value. The effect of ‘c’ is restricted to the sub-stack in

which it appears. Thus, a ‘c’ must be applied to every sub-stack if the complete animation stack is to be cleared. Moreover, if applied, ‘c’ should go into the first position of the transparency list of a sub-stack because *everything* in the sub-stack up to ‘c’ will be cleared.

Timeline example with a single animation stack
Table 1 is an example of a single-stack animation. It lists the contents of a timeline file together with the resulting stack of transparencies. Note how the stack is strictly built from the bottom up as transparency specifications are read from left to right and line by line from the timeline file. In frame No. 4, the stack is first cleared before new transparencies are deposited on it. Also note that the stack is viewed from above and transparencies in higher stack position overprint the lower ones.

Table 1: Timeline example of a single-stack animation

frame No.	timeline file	transparency stack
0	`::0x0,1x2`	———1——— ———0———
1	`::2`	———2——— ———1——— ———0———
2	`::3`	———3——— ———0———
3	`::4`	———4——— ———0———
4	`::c,5x0,6`	———6——— ———5———
5	`::7`	———7——— ———5———
6	`::8`	———8——— ———5———
7	`::9`	———9——— ———5———

Figures 1 and 4 in Sect. 8.1 are animation examples with a single transparency stack.

Grouping objects into layers (= sub-stacks) using ‘;’
Due to the stack-like nature of the animation, the position of a transparency specification in the timeline file determines its *depth* level in relation to other transparencies. The timeline file is processed line by line and from left to right. In a single-stack animation, the stack is strictly built from the bottom up, such that earlier transparencies are overprinted by more recent ones. This may turn out to be inconvenient in certain situations. For example, it might be desirable to change the background image in the middle of an animation without affecting objects that are located in the foreground. For this purpose, transparency specifications can be grouped into layers (sub-stacks) using the semicolon (`;`) as a separator. New transparencies can now be put on top of the individual sub-stacks. After a line of the timeline file has been processed, the global stack is built by placing the sub-stacks on top of the other. Again, the left-to-right rule applies when determining the height of the sub-stacks in relation to each other within the global stack.

Table 2: Timeline example with two sub-stacks

frame No.	timeline file	transparency stack
0	`::  0x0 ; 2x0`	———2——— ..... ———0———
1	`::7,8x2 ; 3x0`	———3——— ———2——— ..... ———8——— ———7——— ———0———
2	`::  ; 4x0`	———4——— ———3——— ———2——— ..... ———8——— ———0———
3	`::c,1x0 ; 5x0`	———5——— ———4——— ———3——— ———2——— ..... ———1———
4	`::  ; 6x0`	———6——— ———5——— ———4——— ———3——— ———2——— ..... ———1———

The layer concept is best illustrated by an example. In the timeline of Table 2, transparencies are grouped into two sub-stacks only. One is reserved for the background images, transparencies No. 0 & 1, to be exchanged in frame No. 3, as well as for two other transparencies, No. 7 & 8, to be interspersed in frame No. 1. A second sub-stack takes the foreground objects that are successively added to the scene. The dotted lines in the third column of the table just mark the border between the two sub-stacks. In frame No. 3, ‘`c`’ first clears the bottom sub-stack before the new background image is inserted. (Instead, ‘`x3`’ could have been used with transparency No. 0 in frame No. 0.) As can be seen in the specifications of frames No. 2 & 4, sub-stacks need not be explicitly populated; the leading semicolons just ensure the proper assignment of transparencies to animation sub-stacks.

See the second animation, Fig. 2, in Sect. 8.1 for a working example that makes use of the timeline and the layer concept.

Associate JavaScript actions with animation frames
The optional fourth field `<JavaScript>` in a frame specification takes JavaScript code to be executed upon display of that frame. This could be used, for instance, to play a sound that was embedded using the 'media9' LaTeX package [6] or to execute JavaScript methods of the animation object. A non-trivial example is looping over a subrange of frames which can be programmed by setting the '`frameNum`' property of the animation object. See Section 7 for details of the animation programming interface.

The backslash '\' and percent '%' characters retain their special meaning from LaTeX and must be escaped by a backslash '\' in the JavaScript code. The same applies to unbalanced braces '{' and '}'. Thus, a code line such as

```
console.println('{}%}{\n');
```

would have to look like

```
console.println('{}\%\}\{\\n');
```

in the timeline file. The first pair of braces are balancing themselves and do not need to be escaped.

Note that JavaScript is executed at the start of displaying the frame. If something is to be executed at the end of a particular frame, the `<JavaScript>` field should be added to the next frame in the timeline file. However, this is not possible for the last frame in a timeline file. Here, the '`setTimeOut`' method can be used to delay the execution of commands:

```
app.setTimeOut('anim.myanim.frameNum=5;', 0.5*anim.myanim.dt)
```

In this example, the 6th frame will be displayed after half of the current frame's lifetime has elapsed.

Other things to note
When designing the timeline, care should be taken not to include a transparency more than once into the *same* animation frame. Besides the useless redundancy, this may slow down the animation speed in the Reader because the graphical objects of a multiply included transparency have to be rendered unnecessarily often at the same time. 'animate' is smart enough to detect multiple inclusion and issues a warning message along with the transparency ID and the frame number if it occurs. Here is an example of a poorly designed timeline:

```
::0
::1x0
::2
::3
::4,2
::5,1 % bad: transparency '1' included twice
::6
```

Also, 'animate' finds and warns about transparencies that have never been used in an animation timeline. This may help to avoid dead code in the final PDF.

7 Programming interface

The package provides a simple JavaScript programming interface which gives access to the animation objects in a PDF file. A particular animation property or method can be accessed by

```
anim['<anim label>'].<property or method>
```

or

```
anim.<anim label>.<property or method>
```

'anim' is an array of animation object references. Animations must be labelled using the 'label=...' command option in order to be present in the 'anim' array. As usual, properties and methods are accessed via the dot notation. Properties and methods of the animation object are summarized in Tables 3 and 4.

One potential use of the JavaScript interface could be within a timeline file associated with an animation. For example, loops over a sub-range of frames can be programmed by setting the 'frameNum' property. See Section 6.2, p. 14 for details.

Also, the programming interface can be used to create custom buttons for playback control. The command \mediabutton from the 'media9' package provides a convenient way for achieving this:

```
\usepackage{media9}
\usepackage{animate}
...

\animategraphics[label=my_anim]{12}{...}{...}{...}

\mediabutton[
  jsaction={anim['my_anim'].playFwd();}
]{\fbox{Play}}
\mediabutton[
  jsaction={anim['my_anim'].frameNum=5;}
]{\fbox{Goto 6th frame}}
```

Table 3: Animation object properties

name	type	access	description
`numFrames`	Integer	read-only	Holds the total number of animation frames.
`frameNum`	Integer	read+write	Gets or sets the current frame being/to be displayed. Note that frame numbers are zero-based. Assigning a value less than zero or greater than `numFrames-1` results in an error.
`fps`	Number	read+write	Gets or temporarily sets the animation frame rate (frames per second). Reset after reaching the end of a sequence, or if the frame rate is set in a timeline or by `\newframe[<frame rate>]`. For scaling overall animation speed, consider using the `speed` property. Assigning a value less than zero results in an error.
`speed`	Number	read+write	Globally scales animation speed. The value must be greater than zero. A value of `1` means 'normal speed' as specified by the frame rate in the document source. Larger values mean 'faster', values between zero and one mean 'slower'.
`dt`	Number	read-only	Holds the time span (milliseconds) for display of the current frame.
`isPlaying`	Boolean	read-only	Holds the value `true` if the animation is currently playing, `false` otherwise.
`playsFwd`	Boolean	read-only	Holds the value `true` if the animation is played forward, independently of whether the animation is currently paused or not.

Table 4: Animation object methods

name	description
`playFwd()`	Starts playing the animation in the forward direction, setting the properties `isPlaying` and `playsFwd` to `true`.
`playBwd()`	Starts playing the animation in the backward direction, setting `isPlaying` to `true` and `playsFwd` to `false`.
`pause()`	Pauses animation, setting `isPlaying` to `false`.
`stopFirst()`	Stops animation and rewinds to the first frame. `isPlaying` is set to false.
`stopLast()`	Stops animation and goes to the last frame. `isPlaying` is set to false.

8 Examples

8.1 Animations from sets of files, using \animategraphics

Animations in this section are made from graphics files that were prepared with METAPOST. Run 'mpost --tex=latex' on the files ending in '.mp' in the 'files' directory to generate the graphics files. Both examples make use of the 'timeline' option to reduce the resulting PDF file size.

The first example, Fig. 1, originally written by Jan Holeček [4], shows the exponential function $y = e^x$ and its approximation by Taylor polynomials of different degree. Below the animation, a custom button was inserted using the JavaScript programming interface and the \mediabutton command from the 'media9' package.

```
\documentclass{article}
\usepackage{animate}
\usepackage{media9}
\usepackage{graphicx}

\begin{filecontents}{timeline.txt}
::0x0 % coordinate system & y=e^x, repeated until last frame
::1 % one blue curve per frame
::2
::3
::4
::5
::6
::7
::8
\end{filecontents}

\begin{document}

\begin{center}
  \animategraphics[
    label=taylor,
    controls, loop,
    timeline=timeline.txt
  ]{4}{exp_}{0}{8}

  \mediabutton[
    jsaction={
      if(anim['taylor'].isPlaying)
        anim['taylor'].pause();
      else
        anim['taylor'].playFwd();
    }
  ]{\fbox{Play/Pause}}
\end{center}

\end{document}
```

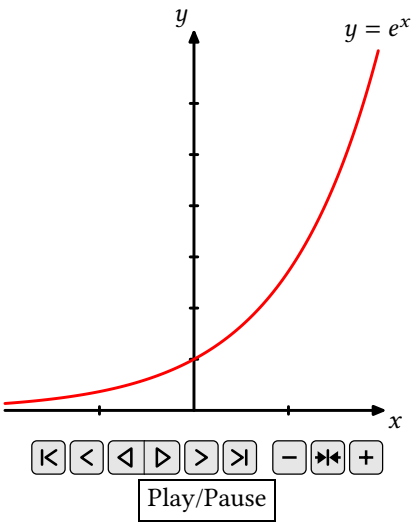

Figure 1

The second, somewhat more complex example, Fig. 2, animates the geometric construction of a scarabaeus. In addition to the use of a timeline, it introduces the layer concept. This example is adapted from Maxime Chupin's original METAPOST source file [1]. The present version separates stationary from moving parts of the drawing and saves them into different files. A total of 254 files, scarab_0.mps through scarab_253.mps, is written out by running '`mpost --tex=latex`' on the source file 'scarab.mp'. Files 0 through 100 contain the red line segments that make up the growing scarabaeus. Files 101 through 201 contain the moving construction lines and files 202 through 252 contain the gray lines which represent intermediate stages of the construction. The last file, No. 253, contains the coordinate axes, two stationary construction lines and the labels which do not move. A timeline file 'scarab.tln' is written out on-the-fly during the LaTeX run. It arranges the animation into three layers, forcing the gray lines into the background, the coordinate axes into the intermediate layer and the scarabaeus along with the moving construction lines into the foreground. The final animation consists of 101 individual frames.

```
\documentclass{article}
\usepackage{intcalc} %defines \intcalcMod for Modulo computation
\usepackage{animate}
\usepackage{graphicx}

\newcounter{scarab}
\setcounter{scarab}{0}
\newcounter{blueline}
\setcounter{blueline}{101}
\newcounter{grayline}
\setcounter{grayline}{202}

%write timeline file
```

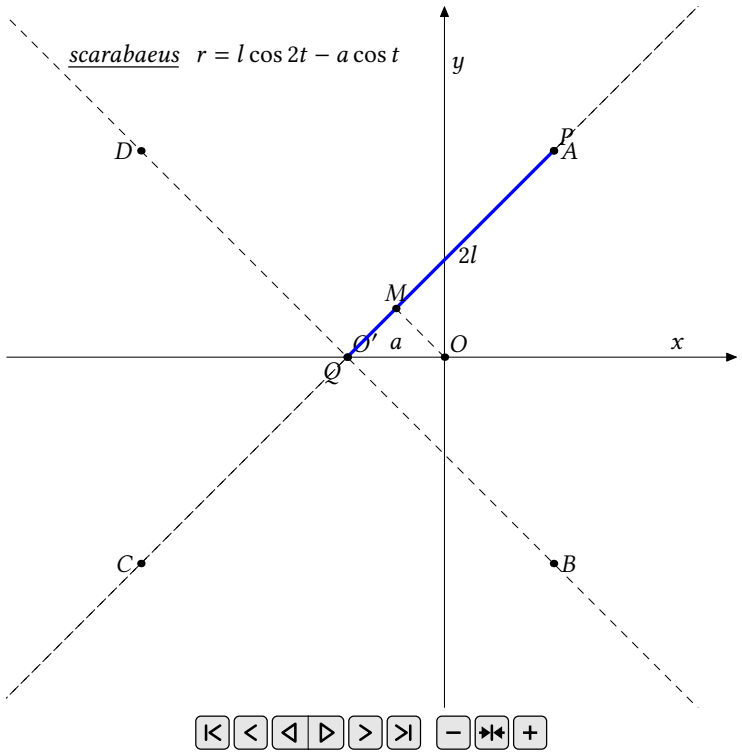

Figure 2

```
\newwrite\TimeLineFile
\immediate\openout\TimeLineFile=scarab.tln
\whiledo{\thescarab<101}{
  \ifthenelse{\intcalcMod{\thescarab}{2}=0}{
    %a gray line is added to every 2nd frame
    \immediate\write\TimeLineFile{%
      ::\thegrayline x0;253;\thescarab x0,\theblueline}
    \stepcounter{grayline}
  }{
    \immediate\write\TimeLineFile{%
      ::;253;\thescarab x0,\theblueline}
  }
  \stepcounter{scarab}
  \stepcounter{blueline}
}
\immediate\closeout\TimeLineFile

\begin{document}

\begin{center}
  \animategraphics[
    width=0.8\linewidth,
    controls, loop,
```

```
    timeline=scarab.tln
  ]{12}{scarab_}{0}{253}
\end{center}

\end{document}
```

8.2 Animating PSTricks graphics, using 'animateinline' environment

Fig. 3 is an inline graphics example adapted from [3].

```
\documentclass{article}
\usepackage{pst-3dplot}
\usepackage{animate}

%draws a torus sector
\newcommand{\torus}[2]{% #1: angle of the torus sector,
  % #2: linewidth of leading circle
  \psset{Beta=20,Alpha=50,linewidth=0.1pt,origin={0,0,0},unit=0.35}%
  \begin{pspicture}(-12.3,-6.3)(12.3,7)%
    \parametricplotThreeD[xPlotpoints=100](80,#1)(0,360){%
      t cos 2 mul 4 u sin 2 mul add mul
      t sin 2 mul 4 u sin 2 mul add mul
      u cos 4 mul
    }%
    \parametricplotThreeD[yPlotpoints=75](0,360)(80,#1){%
      u cos 2 mul 4 t sin 2 mul add mul
      u sin 2 mul 4 t sin 2 mul add mul
      t cos 4 mul
    }%
    \parametricplotThreeD[yPlotpoints=1,linewidth=#2](0,360)(#1,#1){%
      u cos 2 mul 4 t sin 2 mul add mul
      u sin 2 mul 4 t sin 2 mul add mul
      t cos 4 mul
    }%
  \end{pspicture}%
}

\begin{document}

\begin{center}
\begin{animateinline}[poster=last, controls, palindrome]{12}%
  \multiframe{29}{iAngle=80+10, dLineWidth=2.9pt+-0.1pt}{%
    %iAngle = 80, 90, ..., 360 degrees
    %dLineWidth = 2.9pt, 2.8pt, ..., 0.1pt
    \torus{\iAngle}{\dLineWidth}%
  }%
\end{animateinline}%
\end{center}
```

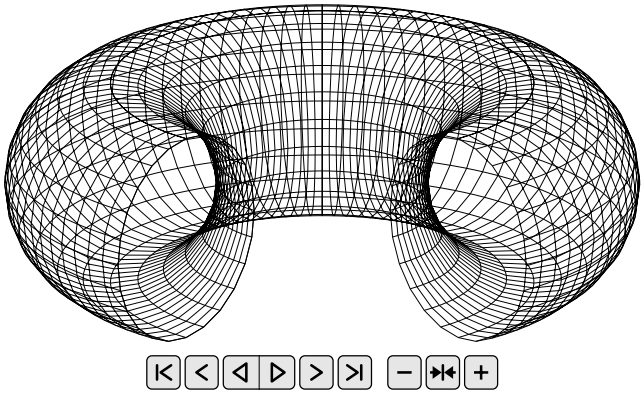

Figure 3

```
\end{document}
```

Another inline example, Fig. 4, is an animation of the Lorenz Attractor. The Lorenz Attractor is a three-dimensional parametric curve whose coordinates are obtained by integrating the set of three ordinary differential equations

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha(y - x)\\ \frac{\mathrm{d}y}{\mathrm{d}t} &= x(\beta - z) - y\\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - \gamma z\end{aligned}$$

with respect to the independent parameter t. The shape of the attractor strongly depends on the values chosen for the coefficients α, β and γ as well as on the initial conditions, that is, the coordinates x_0, y_0 and z_0 of the starting point of the curve. Here we use the values $\alpha = 10$, $\beta = 28$, $\gamma = 8/3$ and the starting point $\mathbf{x}_0 = (10, 10, 30)$.

The right hand sides of the equations above are defined in the macro `\lorenz` as algebraic expressions. The initial value problem is solved by the macro `\pstODEsolve` from the PSTricks package 'pst-ode' and plotted by the macro `\parametricplotThreeD` from the PSTricks package 'pst-3dplot'.

A timeline file, written on-the-fly, is used to assemble the curve segments frame by frame to the growing attractor which, in turn, is put on top of the x-y-z coordinate system. After the attractor has been completed, the transparency stack is cleared. Then, transparencies containing the complete curve and the coordinate system seen from different viewpoints are put in a row to produce the animated fly-around.

```
\documentclass{article}
\usepackage{multido}
\usepackage{pst-3dplot}
\usepackage{pst-ode}
\usepackage{animate}

\begin{document}
```

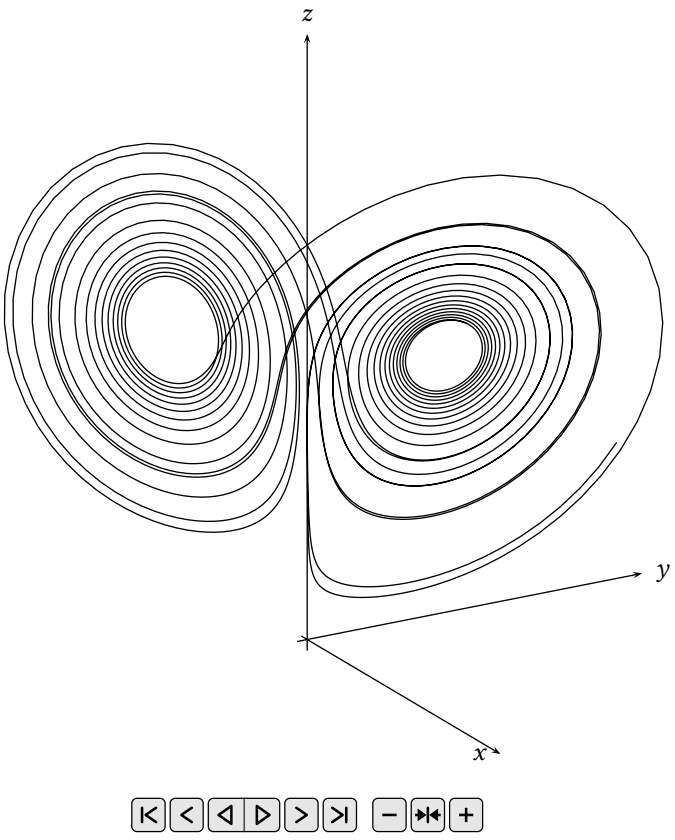

Figure 4

```
%Lorenz' set of differential equations
\def\lorenz{%
  10*(x[1]-x[0]) | %dx/dt
  x[0]*(28-x[2]) - x[1] | %dy/dt
  x[0]*x[1] - 8/3*x[2] %dz/dt
}%
%
%write timeline file
\newwrite\OutFile%
\immediate\openout\OutFile=lorenz.tln%
\multido{\iLorenz=0+1}{101}{%
  \immediate\write\OutFile{::\iLorenz x0}%
}%
\immediate\write\OutFile{::c,101}%
\multido{\iLorenz=102+1}{89}{%
  \immediate\write\OutFile{::\iLorenz}%
}%
```

```
\immediate\closeout\OutFile%
%
\psset{unit=0.155,linewidth=0.5pt}%
\noindent\begin{animateinline}[
  timeline=lorenz.tln,
  controls,poster=last,
  begin={\begin{pspicture}(-36,-13)(36,55)},
  end={\end{pspicture}}
]{10}
  %coordinate axes
  \psset{Alpha=120,Beta=20}%
  \pstThreeDCoor[xMax=33,yMax=33,zMax=55,linecolor=black]%
\newframe
  %attractor segments
  \gdef\initCond{10 10 30}% initial condition
  \pstVerb{/lorenzXYZall {} def} %takes the whole attractor
  \multiframe{100}{rtZero=0+0.25,rtOne=0.25+0.25}{%
    %compute current attractor segment, store it in 'lorenzXYZseg'
    \pstODEsolve[algebraic]{%
      lorenzXYZseg}{0 1 2}{\rtZero}{\rtOne}{26}{\initCond}{\lorenz}%
    %empty initial condition --> next \pstODEsolve continues
    \gdef\initCond{}% from last state vector
    %append segment to the whole attractor stored in 'lorenzXYZall'
    \pstVerb{%
      /lorenzXYZall [lorenzXYZall lorenzXYZseg] aload astore cvx def}%
    %plot the current segment
    \listplotThreeD[plotstyle=line]{lorenzXYZseg}%
  }%
\newframe% required between two \multiframe
  %fly-around (whole attractor)
  \multiframe{90}{rAlpha=116+-4}{%
    \psset{Alpha=\rAlpha,Beta=20}%
    \pstThreeDCoor[xMax=33,yMax=33,zMax=55,linecolor=black]%
    \listplotThreeD[plotstyle=line]{lorenzXYZall}%
  }%
\end{animateinline}

\end{document}
```

The last inline example in Fig. 5 is a ticking metronome written by Manuel Luque [5]. The short clicking sound was embedded by means of the 'media9' package. Whenever the pendulum reaches one of its reversal points, playback of the sound file is started using JavaScript. The JavaScript code was inserted at the corresponding frame specifications in a timeline file. Since the PSTricks macros for drawing the metronome body and the pendulum are quite long they have been moved into an external file, `files/pstmetronome.tex`. Note that the sound can be heard only on Win and Mac platforms. Even then, mileage may vary. A dual core CPU may be necessary for fluent playback.

```
\documentclass[12pt]{article}
```

```
\usepackage{pstricks,pst-node,pst-plot,pst-tools,pst-text}
\usepackage{animate}
\usepackage{media9}

%writing timeline to external file
\begin{filecontents}{metro.txt}
::0x0,1 : annotRM['click'].callAS('play');
::2
::3
::4
::5
::6
::7
::8
::9
::10
::11
::12
::13
::14
::15
::16
```

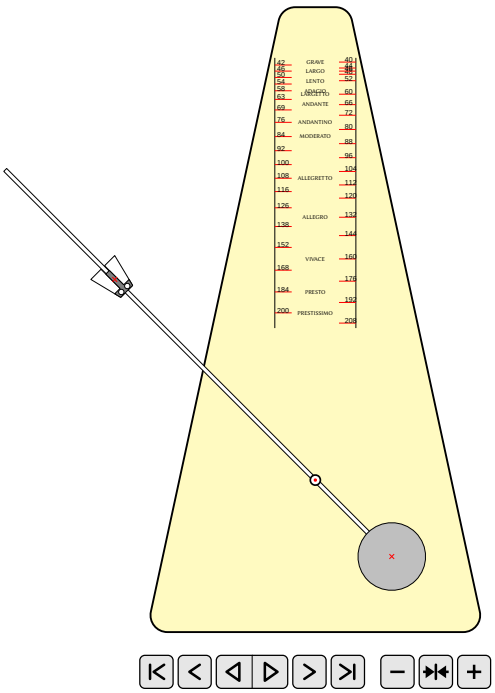

Figure 5

```
::17
::18
::19
::20
::21
::22
::23
::24
::25
::26 : annotRM['click'].callAS('play');
\end{filecontents}

\begin{document}

\begin{center}
  %loading metronome macros from external file
  \input{files/pstmetronome}
  %
  %sound inclusion: click.mp3
  \makebox[0pt][r]{\includemedia[
    width=1ex,height=1ex,
    label=click,
    addresource=click.mp3,
    activate=pageopen,transparent,noplaybutton,
    flashvars={source=click.mp3&hideBar=true}
  ]{}{APlayer.swf}}%
  %
  %animated metronome
  \begin{animateinline}[
    controls,
    width=0.7\linewidth,
    palindrome,
    begin={\begin{pspicture}(-9.5,-5)(9.5,15)},
    end={\end{pspicture}},
    timeline=metro.txt
  ]{25}
    %metronome without pendulum
    \metronomebody
  \newframe
    %half period of pendulum swing (26 frames)
    \multiframe{26}{i=0+4}{
      \pendulum{\i}
    }
  \end{animateinline}
\end{center}

\end{document}
```

9 Animated SVG

Thanks to Martin Gieseking's `dvisvgm` utility [2] that ships with all major TEX distributions, package 'animate' can produce self-contained animated SVG, with all the bits and pieces already included that are necessary to run in modern Web browsers as standalone files or as embedded objects within a Web page made of HTML. Animations have the same look and usability, including optional control buttons, as if they were embedded in a PDF document. Animated SVG even work on mobile devices.

As `dvisvgm` is linked against the Ghostscript library, it can parse and convert embedded PostScript to inline SVG code. It is therefore compatible with the popular TikZ and PSTricks LATEX packages.

SVG is a single-page graphics format. Therefore, it is recommended to produce documents with a single animation per file or page. Thereafter, `dvisvgm` converts every page of the DVI input to a standalone animated SVG file. You may want to use the 'standalone' document class for creating standalone SVG animations. Pass '`dvisvgm`' as a global document class option. In this way, it gets conveyed to 'animate' and other packages to be loaded, such as 'graphicx' or TikZ.

The following code may serve as a template for generating standalone animated SVG:

```
\documentclass[dvisvgm]{standalone}

\usepackage{animate}
\usepackage{graphicx}

%\usepackage{xcolor}
%\pagecolor{white} % opaque background with solid colour

%\usepackage{pstricks} % enable as needed
%\usepackage{tikz}

\begin{document}

%
% \animategraphics{..}{...}{...}{...}
%
% or
%
% \begin{animateinline}{..} ... \end{animateinline}
%

\end{document}
```

Note that when animating external graphics with `\animategraphics`, only PDF and PostScript (EPS, PS, MPS) files are converted to inline SVG code; files in other formats (SVG, PNG, JPEG) remain external and must be bundled with the final SVG output. Thus, for obtaining self-contained SVG, it is recommended to convert PNG, JPEG and SVG files to PDF or PostScript first. Also note that PostScript files must have all required fonts embedded. This is not always the case for METAPOST-generated Post-

Script. Here, embedding of fonts is ensured by putting '`prologues := 3;`' into the header of the METAPOST input.

Use one of

```
latex
platex
dvilualatex
xelatex -no-pdf
```

to produce DVI or XDV output from the LaTeX source. After this, SVG is obtained by running

```
dvisvgm --font-format=woff --exact --zoom=-1 --page=1,- --optimize ...
```

on the intermediate DVI or XDV file.

Option '`--font-format=woff`' (or '`--font-format=woff2`') prompts `dvisvgm` to embed document fonts in a format that is understood by Web browsers. It ensures that typeset text looks as in normal PDF output.

Option '`--exact`' tells `dvisvgm` to calculate exact bounding boxes around font glyphs. This avoids clipping of glyphs in the SVG output, as glyphs usually tend to be slightly bigger than their boxes defined in the font files.

The purpose of '`--zoom=-1`' is to produce responsive SVG. If embedded in a Web page, this kind of SVG will automatically scale to fill up the available space of its surrounding container, usually an `<object>` tag (see below). If viewed standalone in a Web browser, a responsive SVG fills up the complete browser tab.

By default, `dvisvgm` processes only the very first page of the input file. To convert multipage DVI/XDV with several animations, add option '`--page=1,-`'.

With option '`--optimize`', `dvisvgm` applies several optimizations to reduce the output file size.

As SVG derives from XML it is not known to be particularly economical in terms of file size. Compressed SVG, with file extension '`svgz`', shortens download times and is supported by most Web browsers. It can be generated by adding option '`-z`'. Also, option '`--precision=1`' may be used to reduce the SVG file size. It limits the precision of floating point numbers, such as coordinates, to one decimal figure. Sometimes, animations may behave stangely after applying this option. Then, of course, it should be omitted.

The recommended way to include animated SVG into HTML is to use the `<object>` tag. The `<img>` tag does not work here, as it ignores the embedded JavaScript. However, it may still be used as fallback. Also, it allows for search engine indexing, if desired:

```
<object type="image/svg+xml" data="animatedImage.svg">
  <!-- optional (increases loading time):
       fallback & search engine indexing -->
  <img src="animatedImage.svg" />
</object>
```

In TEX4ht documents, the whole `<object>...</object>` tag can be inserted by wrapping it in a `\HCode{...}` command.

10 Bugs

- The maximum frame rate that can actually be achieved largely depends on the complexity of the graphics and on the available hardware. In Acrobat Reader, you might want to experiment with the 2D graphical hardware acceleration feature. Go to menu 'Edit' → 'Preferences' → 'Page Display' → 'Rendering' to see whether hardware acceleration is available. A 2D GPU acceleration check box will be visible if a supported video card has been detected. Also, enabling or disabling the page cache ('Edit' → 'Preferences' → 'Page Display' → 'Rendering' → 'Use page cache') may affect the rendering performance.
- Animated SVG is best viewed in Web browsers that are based on the Blink rendering engine. The most prominent representatives are Chrome, its open-source base Chromium and Opera. Unfortunately, Firefox is very slow.
- The `dvips` option '`-Ppdf`' should be avoided entirely or followed by something like '`-D 1200`' on the command line in order to set a sensible DVI resolution. This does *not* degrade the output quality! The configuration file 'config.pdf' loaded by option '`-Ppdf`' specifies an excessively high DVI resolution that will be passed on to the final PDF. Eventually, Acrobat Reader gets confused and will not display the frames within the animation widget.
- Animations do not work if the PDF was produced with Ghostscript versions older than 9.15.
- If the '`animateinline`' environment is used in a right-to-left typesetting context (RTL) and using the (pdf)LATEX and XELATEX engines, every frame's content should be enclosed in a pair of `\beginR` and `\endR` commands in order to correctly typeset RTL text contained therein. This can be conveniently done by means of the '`begin`' and '`end`' options of the '`animateinline`' environment.
- Animations with complex graphics and/or many frames may cause LATEX to fail with a '`TeX capacity exceeded`' error. The following steps should fix most of the memory related problems.

  MiKTEX:

  1. Open a DOS command prompt window (execute 'cmd.exe' via 'Start' → 'Run').
  2. At the DOS prompt, enter
     `initexmf --edit-config-file=latex`
  3. Type
     `main_memory=12000000`
     into the editor window that opens, save the file and quit the editor.
  4. To rebuild the format, enter
     `initexmf --dump=latex`
  5. Repeat steps 2–4 with config files `pdflatex` and `xelatex`

TEX Live:

1. Find the configuration file 'texmf.cnf' by means of
   `kpsewhich texmf.cnf`
   at the shell prompt in a terminal.
2. As Root, open the file in your favourite text editor, scroll to the '`main_memory`' entry and change it to the value given above; save and quit.
3. Rebuild the formats by
   ```
   fmtutil-sys --byfmt latex
   fmtutil-sys --byfmt pdflatex
   fmtutil-sys --byfmt xelatex
   ```

- PDFs with animations cannot be embedded (via `\includegraphics`, `\includepdf`) into other documents as the animation capability gets lost.
- Animations should not be placed on *multilayered* slides, also known as overlays, created with presentation making classes such as Beamer or Powerdot. Those document classes turn overlays into separate PDF pages and re-insert the animation on every page thus produced. The animations are independent from each other and do not share the current playing state, such as frame number, playing speed and direction. Therefore, put animations on flat slides only; slides without animations may still have overlays, of course. On TEX.SE [7], a method is suggested for placing an animation on a slide with overlays. It makes use of the programming interface introduced in Sect. 7, p. 15.

11 Acknowledgements

I would like to thank François Lafont who discovered quite a few bugs and made many suggestions that helped to improve the functionality of the package. Many thanks to Jin-Hwan Cho, the developer of `dvipdfmx`, for contributing the `dvipdfmx` specific code, and to Walter Scott for proof-reading the documentation.

References

[1] Chupin, M.: *Syracuse MetaPost/Animations.* URL: http://melusine.eu.org/syracuse/metapost/animations/chupin/?idsec=scara

[2] *dvisvgm: A fast DVI to SVG converter* URL: http://dvisvgm.de

[3] Gilg, J.: PDF-Animationen. In: *Die TEXnische Komödie*, Issue 4, 2005, pp. 30–37

[4] Holeček, J.; Sojka, P.: Animations in pdfTEX-generated PDF. In: *TEX, XML, and Digital Typography*, Springer, 2004, pp. 179–191. doi:10.1007/978-3-540-27773-6_14

[5] Luque, M.: *PSTricks : applications.* URL: http://pstricks.blogspot.com

[6] *The media9 Package.* URL: http://www.ctan.org/pkg/media9

[7] *Beamer: animate package and overlay.* URL: https://tex.stackexchange.com/a/385209