PDF extra – extra PDF features for OpTEX

Version 0.3

Michal Vlasák, 2021

`PDFextra` is a third party package for OpTEX. It aims to provide access to more advanced PDF features, which are currently not supported in OpTEX – especially interactive and multimedia features. The development is hosted at `https://github.com/vlasakm/pdfextra`.

In the spirit of OpTEX you may use these macros in any form you want. Either by installing this package and doing `\load[pdfextra]` in OpTEX, or just by copying some useful parts of this package into your documents / packages. OpTEX namespacing is used, but it can be easily stripped, if you wish to incorporate these macros into other macro packages. The code currently depends on LuaTEX, but mostly uses only pdfTEX primitives and a few simple macros from OpTEX. Additionally, the package can be used in plain LuaTeX and LuaLaTeX (in a limited form), see section 1.7.

User documentation (`pdfextra-doc.tex`) and technical documentation interleaved with source code (`pdfextra.opm`) are all typeset in this PDF file. Some examples of usage are in the user documentation, but file `pdfextra-example.tex` contains more examples.

Contents

Chapter 1

User documentation

1.1 Defining files

Many commands provided by this package require you to supply a file ⟨*name*⟩. This is because many commands either work directly (like inserting attachments or multimedia) or can optionally use files (like inserting JavaScript). The "right" way to use ⟨*name*⟩ is to first define the ⟨*name*⟩ with:

`\filedef/`⟨*type*⟩`[`⟨*name*⟩`]{`⟨*path or URL*⟩`}`.

Where ⟨*name*⟩ is the name you will use to refer to this file. It is currently limited to ASCII only (as all "⟨*name*⟩s" required by this package). Interpretation of ⟨*path or URL*⟩ depends on the type, which may be:

- `e`, "embedded file". The file with path ⟨*path*⟩ will be embedded to the PDF file. A file that is embedded this once, can later be used many times in different contexts, e.g. you may use it to attach a video as an attachment but also have it play on page 1 and even other pages. This is the best way, because the resulting PDF file is self contained.
- `x`, "external file". ⟨*path*⟩ can only be a path to the current directory. To refer to the file only ⟨*path*⟩ is used, sort of like a reference. This way the file you want to refer to *has* to be present in the same directory as the PDF file when it is *viewed*!
- `u`, "URL file". ⟨*URL*⟩ is the URL of the file you want to refer to.

All these create the same type of object, which ideally could be used interchangibly everywhere a *file specification* is required in PDF. This is sadly not always true. The limitations to only certain types of `\filedef`'s will be mentioned in due sections. But as a rule of thumb, most of the time you want to embed the files into PDF. The external/URL references are good for refering to external files, although other methods are also possible there.

Because most of the times you want ⟨*name*⟩ to be embedded file, you may omit the prior definition and instead use the ⟨*path*⟩ itself. The file will be autoembedded.

Examples:

```
\filedef/u[doc-internet]{http://petr.olsak.net/ftp/olsak/optex/optex-doc.pdf}
\filedef/x[doc-local]{optex-doc.pdf}
\filedef/e[doc-embedded]{optex-doc.pdf}
```

1.2 Multimedia

It is possible to insert video, audio and 3D files for playback/display inside a PDF file (on a page). There are several different PDF mechanisms for inserting multimedia. For audio/video this package uses the so called "Renditions", which currently have the best support in PDF viewers (fully works in Acrobat and Foxit, partly in Evince and Okular). Rich Media annotations are used for 3D art (works only in Acrobat Reader), although it is possible to also insert audio/video using this mechanism it is very restricting and Renditions are recommended.

Use command `\render[`⟨*name*⟩`][`⟨*optional key-value parameters*⟩`]{`⟨*appearance*⟩`}` for inserting audio/video with Renditions or `\RM[`⟨*name*⟩`][`⟨*optional key-value parameters*⟩`]{`⟨*appearance*⟩`}` for inserting audio/video/3D as Rich Media annnotation. The result of this command is similiar to what `\inspic` produces[1]. Both commands expect ⟨*name*⟩ to be `\filedef`'d name of the file to play/display. As usual, fallback for interpreting ⟨*name*⟩ as path (and embedding it) is in place. It is recommended to only use embedded files with both mechanisms (Rich Media requires it, Renditions with other than embedded files do not work in Acrobat). Optional key-value parameters may be used to customize default values. They may be left out enitrely (including brackets). Last parameter, ⟨*appearance*⟩, defines so called "normal apperance", which is shown before the annotation is activated (audio/video starts playing or 3D scene is displayed). The dimensions of resulting multimedia annotation will be taken from ⟨*appearance*⟩. Most likely you want to use a "poster" picture (inserted with `\inspic`) as appearance – the dimensions will be taken from it and e.g. aspect ratio will be nicely preserved.

---

[1] But because annotations are involved, transformations using PDF literals will not work as expected.

Customization of Renditions is possible using key-value parameters, but beware that it mostly doesn't work in Evince and Okular (Acrobat and Foxit are fine in this regard). The available parameters are in Table 1.2.1. Most customizations of Rich Media concern 3D art. Available parameters are listed in Table 1.2.2.

Table 1.2.1 Key value parameters available for Renditions (`\render`)

Key	Possible values	Default	Description
`controls`	`true` or `false`	`false`	Whether to display audio/video player controls.
`volume`	decimal betwen 0 and 100	100	Audio volume.
`repeat`	integer ≥ 0	1	Number of repetitions (0 means loop forever).
`background`	OpTEX color	`\White`	Color used for part of the annotation not covered by video player (for wrong aspect ratios).
`opacity`	decimal between 0 and 1	1	Opacity of `background`.
`aactions`	`\renditionautoplay`	(none)	Can be used for autoplay on page open.
`name`	ascii string	⟨*name*⟩	Name for use with actions and scripts.

Table 1.2.2 Key value parameters available for Rich Media (`\RM`)

Key	Possible values	Default	Description
`activation`	`explicit` or `auto`	`explicit`	Whether to automatically activate the annotation on page open or display normal apperance until user clicks.
`deactivation`	`explicit` or `auto`	`explicit`	Whether to automatically deactivate the annotation on page close or require explicit deactivation by user (from right click menu).
`toolbar`	`true` or `false`	`true`	Whether to show 3D toolbar (with view and other options).
`views`	comma separated list of view ⟨*name*⟩s	⟨*name*⟩	List of names of 3D views to be used. The default is to try a view of same ⟨*name*⟩. Beware that unknown views are silently ignored.
`scripts`	comma separated list of script ⟨*name*⟩s	(none)	List of names of JavaScript script file ⟨*name*⟩s to be used.
`name`	ascii string	⟨*name*⟩	Name for use with actions and scripts.

The weird `name` key is only required if one media file is used more than once and control using actions or JavaScript scripts is needed.

Examples of video insertion:

```
% embed file under name "video"
\filedef/e[video]{example-movie.mp4}
% insert video into page using Renditions mechanism with controls and autoplay
\render[video][
  name=bigvideo,
  controls=true,
  aactions=\renditionautoplay,
]{\picwidth=\hsize \inspic{example-image.pdf}}

% render the same file again, but with different dimensions, no controls
% and explicit activation
\render[video]{\inspic{example-image.pdf}}
```

When displaying 3D there are more things involved. First, only U3D and PRC can be included in PDF files. The simplest way to show the 3D scene on page is without any optional parameters:

```
\RM[part.prc]{\picwidth=\hsize \inspic{part.png}}
```

The resulting view will be what is defined in the 3D file. But it is possible to customize it, by creating custom view. Or even more of them – they will be available in the user interface for easy switching, first one is considered default. Parameters not defined in a custom view often take what is in the 3D file as default value. `\DDDview[⟨view name⟩][⟨key-value parameters⟩]` is the command for defining 3D views. The brackets surrounding key-value parameters have to be included even if no key-value parameters are used. The available parameters are explained in Table 1.2.3.

Table 1.2.3 Key value parameters available for 3D views (`\DDDview`)

Key	Possible values	Default	Description
`projection`	`perspective` or `ortho`	`perspective`	Projection type to use (perspective distorts the view, e.g. parallel lines are not shown as such, but is more natural to human eye, orthogonal projection is what is generally used with technical parts).
`scale`	decimal number	1	Scaling to use when orthogonal projection is used.
`FOV`	number between 0 and 180	30	Field of view for use with perspective projection.
`background`	OpTeX color	`\White`	Color used as background in the 3D scene.
`rendermode`	`Solid`, `SolidWireframe`, `Transparent`, `TransparentWireframe`, `BoundingBox`, `TransparentBoundingBox`, `Wireframe`, `ShadedWireframe`, `Vertices`, `ShadedVertices`, `Illustration`, `SolidOutline`, `ShadedIllustration`	(taken from 3D file)	Used rendering mode. See PDF standard[i] for more details.
`lighting`	`White`, `Day`, `Night`, `Hard`, `Primary`, `Blue`, `Red`, `Cube`, `CAD`, `Headlamp`	(taken from 3D file)	Used lighting scheme. See PDF standard[ii] for more details.
`method`	`media9`, `manual`, `u3d`	`media9`	Method used for defining 3D camera position and orientation.

[i] https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf#G12.2358303
[ii] https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf#G12.2358356

The value of `method` key influences what other key-value parameters are available. For details about the manual method see the technical documentation (2.8.3). The `u3d` method works only with U3D files (not with PRC files) and requires you to know the internal "path" of the view contained in the file and is hence not always useful (especially when the paths are weirdly constructed by exporting applications). But if you know the path you can use "`method=u3d, u3dpath=`⟨*path*⟩". All Unicode characters are allowed in ⟨*path*⟩.

Table 1.2.4 Key value parameters available for media9 method of 3D views (`\DDDview`)

Key	Possible values	Default	Description
`coo`	three space separated (decimal) numbers	`0 0 0`	“Center of orbit”. Coordinates of the point camera is supposed to look at.
`roo`	(decimal) number	`0`	Distance of camera from `coo`.
`c2c`	three space separated (decimal) numbers	`0 -1 0`	“Center of orbit to camera” vector. A directional vector (i.e. length doesn’t matter). The direction the camera will be looking at is opposite of this vector. Default is view towards positive y.

The most useful method of specifying the camera position/orientation is `method=media9`. The same method is also used in a number of other packages (movie15, rmannot, ConTEXt). Its input are key-value parameters, listed in Table 1.2.4. For illustration of the parameters check the documentation[2] of media9.

You can construct simple views only by using these few parameters. When talking about meanings/names of different views it is needed to know the real placement of the 3D object in the 3D world. But for a reasonably orientated object in the center (`coo=0 0 0`) constructing simple views is very easy (keep in mind `method=media9` is default):

```
\DDDview[front][
  projection=ortho,
  roo=400,
]
\DDDview[left][
  projection=ortho,
  roo=400,
  c2c=-1 0 0,
]
\DDDview[top][
  projection=ortho,
  roo=400,
  c2c=0 0 1,
]
\DDDview[isometric][
  projection=ortho,
  roo=500,
  c2c=-1 -1 1,
]
```

We can then perhaps use these views in `\RM`:

```
% Auto activated 3D Rich Media annotation. White background just ensures
% the right dimensions. Comma in `views` is escaped using "{}".
\RM[part.prc][
  activation=auto,
  views={front, left},
]{{\White\vrule width\hsize height\vsize}}

\RM[part.prc][ % the same 3D file, now with different views
  name=part2,
  activation=auto,
  views={top, isometric},
]{{\White\vrule width\hsize height\vsize}}
```

---

[2] https://mirrors.ctan.org/macros/latex/contrib/media9/doc/media9.pdf#figure.8

If you want to deduce the view parameters automatically it is possible. You can just not specify any view, but include the `3Dmenu.js` script from media9 package. It enables you to right click the annotation and select “Get Current View” (or even “Generate Default View” which finds the view all by itself). A window with generated parameters will show up. You can then copy the ones this package understands (`c2c`, `coo`, `roo`) and use them. You don’t have to be excessive with precision, because after calculation everything gets rounded to 6 decimal places anyways.

```
% using 3Dmenu.js to generate 3D view parameters automatically
\RM[part.prc][
  name=part3,
  activation=auto,
  scripts=3Dmenu.js,
]{{\White\vrule width\hsize height\vsize}}
```

The use of `scripts` isn’t limited to this though, there are many other possibilities. First, you can use as many scripts as you want (`scripts={script1.js, script2.js, ...}`), but be careful that 3D JavaScript is kind of special, and has to come from embedded files (here we were using the auto-embedding, but we could have used e.g. `\filedef/e[3dmenu]{3Dmenu.js}` and then “`3dmenu`” instead). For creating your own scripts check out the Acrobat 3D JavaScript API[3]. It is possible to do different transformations and achieve animations using “time events”. The implicit “context” of 3D scripts can be accessed from normal JavaScript actions (see 1.3.5). Vice versa 3D scripts may access the global JavaScript environment using `host` object.

More examples of 3D Rich Media, including usage of 3D JavaScript, are available in the example file `pdfextra-example.tex`. They show how it is possible to port the examples used by media9.

1.3 Actions

Actions are very important aspect of interactivity in the context of PDF. There are a few very useful types of actions, like “goto” actions which jump to other part of document. There are also a few ways how to *execute* actions. The most usual is a clickable area on page, but clickable bookmarks (“document outline”) also execute actions behind the scenes. OpTEX supports only basic “goto” and “URI” actions using `\ilink` (used for `\ref`, `\cite`, etc.) and `\ulink` (used for `\url`).

This package offers generalization of this mechanism. The core of it is a way of specifying an action. This syntax is called ⟨*action spec*⟩ and is used for example by `\hlink` command, which can replace both `\ilink` and `\ulink`. ⟨*action spec*⟩ is a comma separated list of ⟨*type*⟩`:`⟨*arguments*⟩. where ⟨*type*⟩ refers to the action type and the syntax of arguments is dependant on ⟨*type*⟩. Leading spaces are ignored, trailing aren’t. You probably won’t often use the possiblity of specifying multiple actions, but it is for chaining execution of several actions.

`\pdfaction[`⟨*action spec*⟩`]` is available for lower level creation of different actions, but for clickable areas on page you will use `\hlink[`⟨*action spec*⟩`]{`⟨*text*⟩`}` with a very similiar syntax. Although note, that `\hlink`’s interface is also not really high level and wrapping it inside macros like `\ref` or `\url` might be beneficial. ⟨*text*⟩ will be typeset directly and the area it occupies will be clickable. Clicking it executes action defined by ⟨*action spec*⟩. Line breaks inside ⟨*text*⟩ will be possible, in that case several clickable rectangles will be created, one for each line. Normally in text you want the areas to be of the same height and depth (calculated from `\baselineskip`), to achieve sort of a lining, uniform effect. If you want to define big clickable buttons, you may need to turn off the lining effect using `\nolininglinks`. It respects groups, but a counterpart (`\lininglinks`) is also available.

There are a few predefined action types (⟨*type*⟩): **`url`**, **`extref`**, **`extpgref`**, **`named`**, **`transition`**, **`js`**, **`goto3dview`** and **`rendition`**. They will be explained in a moment. Any unrecognized ⟨*type*⟩ is understood as an “internal link”, where ⟨*type*⟩`:`⟨*link*⟩ is the destination of the link. Hence it is possible to use `\hlink` as `\ilink` for example with OpTEX’s normal types of internal links. For example:

```
See section~\hlink[ref:section]{2.2.13} or page~\hlink[pg:5]{5}.
```

`\ulink` may be replaced like this:

```
Visit CTAN's \hlink[url:https://www.ctan.org/]{website}.
```

Before we really get into different types of actions, there is a nicer command for setting the initial (“open”) action of PDF document, which is executed when the document is opened. It defaults to

---

[3] https://wwwimages.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/AcrobatDC_js_3d_api_reference.pdf

opening the page with the zoom level according to viewing user's preferences. But you can change this using `\openaction[⟨action spec⟩]`:

```
\openaction[pg:2]
```

1.3.1 External references

There are two actions analogous to internal links that can be used to link to external documents. `[extref:⟨name⟩:⟨named destination⟩]` can be used to refer to named locations in a PDF document prepared by `\filedef` with ⟨*name*⟩. `[extpgref:⟨name⟩:⟨page number⟩]` is similiar but refers to a page number. Although it would be nice, these actions aren't well supported by all viewers.

Example:

```
\hlink[extref:doc-internet:ref:langphrases]{OpTeX documentation,
        section \"Multilingual phrases and quotation marks".}
\hlink[extpgref:doc-internet:12]{OpTeX documentation, page 12.}
```

(Note that "`:`" in "`ref:langphrase`" is not part of the syntactic rule, it is just the value of ⟨*named destination*⟩ in this case.)

A little bit of customization is possible, see 2.4.4.

Because of the poor support you may find luck with the less universal url action with `#fragment`:

```
\hlink[url:http://petr.olsak.net/ftp/olsak/optex/optex-doc.pdf#ref:langphrases]
      {OpTeX documentation, section \"Multilingual phrases and quotation marks".}
```

1.3.2 Named actions

There are four defined in PDF standard, but viewers may support more. All in examples:

```
\hlink[named:NextPage]{Go to next page,}
\hlink[named:PrevPage]{go to previous page,}
\hlink[named:FirstPage]{go to first page,}
\hlink[named:LastPage]{go to last page.}
```

1.3.3 Transition actions

Transition actions generally make sense only when chained *after* jump actions. The specified transition/animation will occur, before destination is opened, but it will not override a transition defined for the particular page. The syntax is `[transition:⟨transition spec⟩]`. See 1.4 for more information about ⟨*transition spec*⟩.

Example:

```
\hlink[ref:yellow-slide, transition:Box:3:/M /O]
      {Go to yellow slide with long outward Box transition}
```

1.3.4 JavaScript actions

They allow executing pieces of JavaScript code using syntax: `[js:⟨name or script⟩]`. If ⟨*name or script*⟩ is a validly `\filedef`'d the script from file ⟨*name*⟩ will be executed (only embedded files are valid). Otherwise ⟨*script*⟩ will be used directly. Apart from reuse in different documents, scripts loaded from files may be encoded in UTF-16BE and hence support Unicode, inline ⟨*script*⟩s current don't.

Examples:

```
\openaction[js:{%
  app.alert("Javascript alert, open action");
  console.println("printing to console from openaction");
}]

\filedef/e[jstest]{test.js}
\hlink[js:jstest]{JavaScript action from file}
```

(Note how braces were used to guard the comma in JavaScript code from being interpreted as a action separator.)

In these actions you may want to use your own functions which should be defined before user has a chance of activating any other JavaScript actions. This is the purpose of "document level"

JavaScript actions. They function exactly the same way as normal JavaScript actions, but have names (although meaningless, they must be unique) and are executed in order of definition. Use `\dljavascript[`⟨*name*⟩`]{`⟨*name or script*⟩`}` for defining these actions:

```
\filedef/e[preamble]{preamble.js}
\dljavascript[preamble]{preamble}

\dljavascript[initialization]{%
  var data = 42;
  function getRandomNumber() {
      return 4; // chosen by fair dice roll, https://xkcd.com/221/
  }
  console.println("initialized with seed " + getRandomNumber());
}
```

The JavaScript API available is not the same as the one in the web browsers. It is instead specified by Adobe: `https://wwwimages.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf`.

1.3.5 3D JavaScript actions

There are no "special" JavaScript actions for dealing with 3D. Normal JavaScript actions are used. You just have to access the 3D context of the annotation you want to control. For annotation with ⟨*name*⟩ the context is made available in `\DDDcontext{`⟨*name*⟩`}`. Under this context object you find all global definitions from the respective 3D scripts. For example if a "`turn`" function is defined, it is possible to call it like this:

```
\hlink[js:\DDDcontext{part3}.turn();]{Turn by 90 degrees along $x$ axis.}
```

1.3.6 GoTo3Dview actions

GoTo3Dview actions allow changing the view of the 3D scene to one of the predefined views (those listed by `views` comma separated list of `\RM`). The syntax is `[goto3dview:`⟨*name*⟩`:`⟨*view*⟩`]`. ⟨*name*⟩ is name of the annotation whose view we want to change and ⟨*view*⟩ is the intended view. You can refer to last inserted Rich Media annotation using empty name. For ⟨*view*⟩ you can either use "`(`⟨*view name*⟩`)`" (e.g. "`(part)`"), index of the view in the view list (zero based, e.g. "`0`" for the first view) or one of the special values: "`/N`" (next), "`/P`" (previous), "`/F`" (first) "`/L`" ("last").

```
% let's define an annotation with a few views
\RM[part.prc][
  activation=auto,
  views={front, left, top, isometric},
]{{\White\vrule width\hsize height\vsize}}

% try the different methods of reffering to views
\hlink[goto3dview::/N]{Next view},
\hlink[goto3dview::(left)]{left view} and
\hlink[goto3dview:part.prc:3]{third view}.
```

1.3.7 Rendition actions

Rendition actions can be used to control playback of "Rendition annotations". They use the syntax `[rendition:`⟨*name*⟩`:`⟨*operation*⟩`]`, where ⟨*name*⟩ refers to the name of the rendition to control and ⟨*operation*⟩ is one of `play`, `stop`, `pause` or `resume`. As a convenience, you can refer to last inserted rendition using empty name. If a file has been `\render`ed more than once with the same name, the action will influence the first instance.

Beware that currently these actions do not work in Evince and Okular (but do in Acrobat and Foxit). Examples:

```
% rendition with name=video, that we want to control
\render[video]{\inspic{example-image.pdf}}

% we want the rendition action to have yellow border and red content
```

```
\def\_renditionborder{1 1 0}
\let\_renditionlinkcolor\Red

To start playing the video, click \hlink[rendition::play]{\"Play"}.
After that you can \hlink[rendition:video]{pause}.
```

1.4 Transitions and other page attributes

In PDF there are a few settings that can be set as *page attributes*. This means that they apply only to said page. For setting these page attributes, there are two options:

- value for "current page" (or rather the page where the command appears),
- default value used if "current page" value is not set.

While this package contains mechanism to handle all page attributes, not that many are useful for end user. The interesting ones remaining are:

- Transitions and page durations. When page has the transition attribute any jump to this page will display the requested animation (customized by the corresponding parameters). Page duration is the time before PDF viewer will auto advance to the next page. `\transition[`⟨*transition spec*⟩`]` sets transition for the "current page", `\transitions` sets the default. ⟨*transition spec*⟩ has three parts:

  ⟨*animation type*⟩`:`⟨*duration*⟩`:`⟨*raw PDF attributes*⟩

  ⟨*animation type*⟩ is one of `Split`, `Blinds`, `Box`, `Wipe`, `Dissolve`, `Glitter`, `Fly`, `Push`, `Cover`, `Uncover` and `Fade`, or the special value `R` which essentially means no animation and instantaneous transition (regardless of the set duration of transition). ⟨*duration*⟩ is the duration of transition in seconds (integer or decimal number). ⟨*raw PDF attributes*⟩ may be used to customize the animations (for example `/M` can set the direction of motion of `Split`, `Box` and `Fly`, e.g. `/M /I` for inward motion). For the raw PDF attributes refer to the standard itself[4]. `:`⟨*raw PDF attributes*⟩ or even `:`⟨*duration*⟩`:`⟨*raw PDF attributes*⟩ may be omitted. Default values specified by PDF standard will be used in that case (1 second duration and default values for all attributes). Page durations can be set either using `\defaultpageduration[`⟨*duration*⟩`]` or `\pageduration[`⟨*duration*⟩`]`, where duration is in seconds (default is no auto advancement, i.e. ∞). Examples:

  ```
  % unless stated otherwise all pages will have Wipe animation
  % with 1 second duration
  \transitions[Wipe:1]

  % but this page has a 1 second Fade animation
  \transition[Fade]

  \pg+ % (if using \slides)

  % and this page has 3 second Split animation
  % with vertical direction and inward motion
  \transition[Split:3:/Dm /V /M /I]
  ```

  Note that transitions are only displayed when in *full-screen mode*. You can use `\fullscreen` to have the document automatically open in full-screen mode.
- Additional actions. It is possible to define actions which will respond to page events: page open (`/O`) and page close (`/C`). They can be set using `\defaultpageactions[`⟨*additional actions spec*⟩`]` (the default for all pages) and `\pageactions[`⟨*additional actions spec*⟩`]` (current page override). ⟨*additional actions spec*⟩ consists of braced pairs of "event" (O or C in this case) and ⟨*action spec*⟩. Example:

  ```
  \pageactions[
    {O} {js:{app.alert("Page open, random = " + getRandomNumber());}}
    {C} {js:{app.alert("Page close!");}}
  ]
  ```

---

[4] https://wwwimages2.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf#G11.2295795

1.5 Attachments

Every file embedded into a PDF file may optionally be presented in the user interface as an embedded file. This allows readers of the document to save or open the file.

PDF allows two ways of presenting attachments – using annotations (the attachments is represented by a rectangular icon on a page) or global array of attachments (attachments are visible in PDF viewers toolbar). The first method is intended more for document reviewers than for primary insertion. Hence this package supports only the second type.

You may add an embedded file to the global attachments array using `\attach[`⟨*name*⟩`]`. As usual, name is either a ⟨*name*⟩ defined with `\filedef` or alternatively a path to file which will be embedded (and `\filedefd`) automatically. It is not possible to `\attach` files referenced by path or URL.

If you want to automatically display toolbar with embedded files, consider using `\showattached` (see section 1.6).

1.6 Document view

You can choose what is shown when document is opened with commands:

- `\fullscreen` (the document is opened in full-screen mode),
- `\showoutlines` (show bookmarks/outlines toolbar)
- `\showattached` (show attachments in toolbar)

The commands are mutually exclusive and only the first appearing one will be respected.

You can request two page view (odd pages on the right) using `\duplexdisplay`. It is useful for more natural display of double sided documents. Because it may not be desirable to automatically apply this, it is independent of `\margins`.

1.7 Usage in plain LuaTEX or LuaLaTEX

You can use this package also from plain LuaTeX by adding to your document:

```
\input pdfextra
```

or for LuaLaTEX:

```
\usepackage{pdfextra}
```

See the file `pdfextra-example-latex.tex` for the adaptation of the OpTEX examples to LATEX. The usage of the macros described in this document is the same, but there are limitations:

- *Color*. Where this package expects "OpTEX color" key value argument (e.g. `\Blue`), you have to use an RGB triplet instead (e.g. `0.0 0.0 0.0` or the shorter `0 0 0`).

  But for text color setting, you can get away with wrapping commands from LaTEX's `color` package, e.g. to customize link border/color:

  ```
  \sdef{_renditionlinkcolor}{\color[red]}
  ```

- *Initialization.* In OpTEX you normally have to initialize hyperlinks with the command `\hyperlinks`⟨*color for internal links*⟩⟨*color for external links*⟩.
  This is not required by this package. You can instead set the color by setting `\_linkcolor` (fallback for all link types), `\_ilinkcolor` / `\_elinkcolor` (internal / external links). E.g. if you have the LATEX `color` package loaded, you can get blue links like this:

  ```
  \sdef{_linkcolor}{\color[blue]}
  ```

  In addition to the above this means all links will be blue except "rendition" links.
- *Openaction.* If the package `hyperref` is used, then `\openaction` will not work.
- *Labels / hyperlink destinations.* OpTEX uses very simple and consistent scheme for labels / hyperlink destinations:
  - `ref:`⟨*label*⟩ – result of `\label[`⟨*label*⟩`]`
  - `toc:`⟨*tocrefnum*⟩ – result of `\chap`/`\sec`/`\secc` titles
  - `pg:`⟨*gpageno*⟩ – created on each page with global numbering from 1
  - `cite:`⟨*bibpart*⟩`/`⟨*bibnum*⟩ – bibliography references,
  - `fnt:`⟨*gfnotenum*⟩ – link form text to footnote
  - `fnf:`⟨*gfnotenum*⟩ – link from footnote to text
  - `url:`⟨*url*⟩ – used by `\url` or `\ulink`,

Hence these labels / destinations can be used with `\hlink`, e.g. to make text `page 5` a link to page 5, one can use:

```
\hlink[pg:5]{page 5}
```

This is not possible in plain LuaTEX (no destinations are created) or LuaLaTEX (different and incompatible destination names). You would have to create your own destinations adhering to the naming convention ⟨*type*⟩`:`⟨*arguments*⟩ to be able to use `\hlink` as intended for some links.

Chapter 2

Technical documentation

This is the technical documentation. It is intended for those who want to know how this package works internally. Casual users shouldn't need to read this. But if you would like to customize anything or perhaps just use some part of this package, feel free to copy paste and use anything you want in OpTeX's spirit.

This documentation is interleaved within the source itself, both are contained in a single file, `pdfextra.opm` (according to OpTeX conventions). The user documentation is instead contained in `pdfextra-doc.tex`, which itself `\input`'s the documented source file `pdfextra.opm` so that the user and technical documentation is available in a single PDF file, `pdfextra-doc.pdf`.

`pdfextra.opm`

```
\_def\_pdfextra_version{0.3}
\_codedecl \RM {Extra PDF features (v\_pdfextra_version)}
\_namespace{pdfextra}
```

2.1 Package initialization

We ensure that hyperlinking is active. Our fallback `\_linkcolor` must exist. We also use it for `\hyperlinks` if the user didn't enable `\hyperlinks` yet (we don't want to override user setting).

`pdfextra.opm`

```
\_ifdefined\_ilinkcolor\_else
   \_ifdefined\_linkcolor
      \_ea\_let\_ea\_linkcolor \_ifdefined\Blue\Blue\_else\_empty\_fi
   \_else
      \_let\_linkcolor\linkcolor
   \_fi
\_fi

\_ifx\_dest\_destactive\_else
   \_hyperlinks\_linkcolor\_linkcolor
\_fi
```

We are in the OpTeX package namespace. A couple of shortcuts are defined here: `\.isdefined`, `\.trycs`, `\.cs` `\.slet`, `\.slet`, `\.sdef` and `\.sxdef`. They all hard code the package name, because we already have too many levels of indirection.

`pdfextra.opm`

```
\_def\.isdefined#1{\_isdefined{_pdfextra_#1}}
\_def\.trycs#1{\_trycs{_pdfextra_#1}}
\_def\.cs#1{\_cs{_pdfextra_#1}}
\_def\.slet#1#2{\_slet{_pdfextra_#1}{_pdfextra_#2}}
\_def\.sdef#1{\_sdef{_pdfextra_#1}}
\_def\.sxdef#1{\_sxdef{_pdfextra_#1}}
```

2.2 Helper macros

The macros here are just helpers for the macros to follow. They are not useful generally, but proved useful in the expandable context of writing to PDF files.

Already the first one limits the use to LuaTeX (but who needs other engines anyways :). `\.emptyor`⟨*possibly empty text*⟩⟨*text to use when first argument is nonempty*⟩ checks whether the first argument is empty, if not it expands the second argument which can use the text from the first argument with `\.nonempty`. `\.attrorempty`⟨*attribute name*⟩⟨*value*⟩ builds upon the first one and is really useful for PDF dictionaries, when we don't want to write an attribute without a value (a default specified by standard will be used instead).

`pdfextra.opm`

```
\_def\.emptyor#1#2{%
   \_immediateassignment\_edef\.nonempty{#1}%
   \_ifx\.nonempty\_empty\_else #2\_fi
}
\_def\.attrorempty#1#2{\.emptyor{#2}{/#1 \.nonempty}}
```

There is a dillema for handling colors. While typesetting it is possible to use greyscale, CMYK or RGB colors. But there are contexts where it is possible to only use RGB colors. We want to provide the user with two possibilities of specifying colors:

- RGB color using PDF triplet (e.g. `1 0 0`),
- OpTEX color using control sequence (e.g. `\Blue`)

Both are handled by `\.colortorgbdef`⟨*cs*⟩⟨*color specification*⟩, which defines ⟨*cs*⟩ to the corresponding PDF RGB triplet. The indirection with defining a macro is because we want to use the processed color within expansion only contexts where grouping is not possible.

pdfextra.opm

```
\_def\.colortorgbdef#1#2{\_bgroup
   \_def\_setrgbcolor##1{##1}%
   \_def\_setcmykcolor##1{\_cmyktorgb ##1 ;}%
   \_def\_setgreycolor##1{##1 ##1 ##1}%
   \_xdef#1{#2}%
   \_egroup
}
```

`\.xaddto\macro`⟨*text*⟩ is a natural extension of OpTEX's `\addto` that expands ⟨*text*⟩ and is global. `\.tmp` is used throught the package for temporary values.

pdfextra.opm

```
\_def\.xaddto#1#2{\_edef\.tmp{#2}%
   \_global\_ea\_addto\_ea#1\_ea{\.tmp}%
}
```

This package defines a few commands in the form `\macro[`⟨*name*⟩`][`⟨*optional arguments*⟩`]{`⟨*text*⟩`}`. To make it possible to omit the `[`⟨*optional arguments*⟩`]` `\.secondoptdef` is defined.

`\.secondoptdef\`⟨*macro*⟩⟨*parameters*⟩`{`⟨*body*⟩`}` defines `\macro` with first mandatory argument in brackets (saved to `\.name`). Second optional argument in brackets is scanned using helper macro defined with `\optdef` and is saved to `\_opt` token list). Additional ⟨*parameters*⟩ can be specified as with `\optdef` (numbered from `#1`).

pdfextra.opm

```
\_def\.secondoptdef#1{%
   \_def#1[##1]{\_def\.name{##1}\.cs{sopt:\_string#1}}%
   \_ea\_optdef\_csname _pdfextra_sopt:\_string#1\_endcsname[]%
}
```

When processing comma separated lists sometimes it is needed to ignore the remaining text. For this we use `\.untilend` macro which ignores everything up to dummy `\.end`. This is analogous to OpTEX's `\_finbody` used for the same purpose. Sometimes `\.end` is used as sentinel and compared in `\ifx` tests, hence we define it to a unique value.

pdfextra.opm

```
\_def\.untilend#1\.end{}
\_def\.end{_pdfextra_end}
```

For various uses it is necessary to know the number of page where something happens. This has to be handled asynchronously with `\write`. Here we use OpTEX specific `.ref` file and associated macros, but this could be replaced as long as the same interface is exposed.

`\.setpageof`⟨*name*⟩ writes `\.Xpageof`⟨*name*⟩ to the `.ref` file. In the next TEX run `\.Xpageof` finds out the page number (`\gpageno`) from OpTEX's `\_currpage` and saves it so that `\.pageof`⟨*name*⟩ can retrieve it. In the first run we can't be sure of the page where the content will end up. As a rough estimate we take the current page – this actually works well for slides where page breaks are manual.

When `.ref` file is read along with the defintion of `\.Xpageof` this package has not been loaded yet. Hence we can't use namespaced variants of `\.isdefined`, etc.

pdfextra.opm

```
\_refdecl{%
   \_def\.Xpageof#1{\_isdefined{_pdfextra_pageof:#1}\_iffalse
      \_sxdef{_pdfextra_pageof:#1}{\_ea\_ignoresecond\_currpage}\_fi
   }%
}

\_def\.setpageof#1{\_openref \_ewref\.Xpageof{{#1}}}

\_def\.pageof#1{%
   \.trycs{pageof:#1}{%
      \_the\_numexpr\_gpageno+1\_relax % best effort = current page num
   }%
}
```

2.3 Handling of files

Handling of files is a big topic of this package. Files are everywhere – files containing multimedia, JavaScript script files, attachments, externally referred files… Therefore a more sophisticated mechanism for handling files is needed. The mechanism introduced in this section handles all three cases of a *file specification*:

- files embedded in the PDF ("e", embedded file),
- files determined by path ("x", external file),
- files determined by URL ("u", url file).

Although ideally all three would be interchangible this is not always the case, because e.g. some media files must be embedded and linking to external resources does not work with embedded files.

In most cases there are two many names and other associated values involved:

- Some kind of a "friendly" name. This one is sometimes shown by PDF viewers.
- The real name of the file. Also shown but in different contexts.
- The path or URL used to determine the file.
- MIME type of the file.

For example when talking about OpTeX's documentation we might have a friendly name of "opdoc", file name of "`optex-doc.pdf`", URL of "`http://petr.olsak.net/ftp/olsak/optex/optex-doc.pdf`" and MIME type of "application/pdf". Different subset of them is required in different contexts, but the user should only have to specify the friendly name (by which they will refer to the file) and the path/URL of the file. The rest will be deduced. The friendly name is used as a handle and *is usable* in all places where file specification is required (although it may not produce conforming output, see above).

In this two step process – definition and (re)use – we introduce a command for defining files: `\filedef/`⟨*type*⟩ `[`⟨*friendly name*⟩`]{`⟨*path or URL*⟩`}`. The macro itself does general definitions and dispatches the type dependant work to other macros in the form `_filedef:`⟨*type*⟩.

pdfextra.opm

```
\_def\.filedef/#1#2[#3]#4{%
   \.sxdef{filename:#3}{(\.filename{#4})}%
   \_edef\.tmp{\.exttomime{\.fileext{#4}}}%
   \_ifx\.tmp\_empty
      \_opwarning{MIME type of '#4' unknown, using '\.defaultmimetype'}%
      \_edef\.tmp{\.defaultmimetype}%
   \_fi
   \.sxdef{filemime:#3}{\.tmp}%
   \.cs{filedef:#1}{#3}{#4}%
}
\_nspublic \filedef ;
```

Types "e", "x", "u" are predefined, anything else would essentialy be a variant of these.

External file ("x") is determined only by path.

pdfextra.opm

```
\.sdef{filedef:x}#1#2{%
   \.slet{filespec:#1}{filename:#1}%
}
```

URL file ("u") is determined by URL. Using all sorts of characters is allowed by using `\_detokenize`. This time it is necessary to create full *file specification* – a dictionary, where the "file system" is URL.

pdfextra.opm

```
\.sdef{filedef:u}#1#2{%
   \.sdef{filespec:#1}{<</FS /URL /F (\_detokenize{#2})>>}%
}
```

Embedded files ("e") are the most interesting ones. For further use (e.g. for displaying the embedded files as attachments) MIME type is required. It is saved in the stream as a `\Subtype`, encoded as a PDF name (e.g. `/video#2Fmp4`). The embedded file stream must be wrapped in a full *file specification*, which has the `/EF` ("embedded file") entry. Also the friendly name is used for some purpose by PDF viewers, so it set in `/Desc` (description).

pdfextra.opm

```
\.sdef{filedef:e}#1#2{%
   \_edef\.tmp{\.cs{filemime:#1}}%
   \_isfile{#2}\_iffalse
      \_opwarning{file '#2' not found}%
   \_fi
   \_pdfobj stream
      attr{/Type /EmbeddedFile /Subtype \_ea\.mimetoname\_ea[\.tmp]}
      file {#2}%
   \_pdfrefobj\_pdflastobj
   \.sxdef{filestream:#1}{\_the\_pdflastobj\_space 0 R}%
   \_pdfobj {<</Type /Filespec
      /F \.cs{filename:#1}
      /Desc (#1)
      /EF << /F \_the\_pdflastobj \_space 0 R >>%
   >>}%
   \_pdfrefobj\_pdflastobj
   \.sxdef{filespec:#1}{\_the\_pdflastobj\_space 0 R}%
}
```

Now the less interesting part – determining the file names from paths and determining MIME types. The file name is the part after the last “`/`” (if any). The file extension is the part after last “`.`” (if any).

pdfextra.opm

```
\_def\.filename#1{\_ea\.filenameA#1/\.end}
\_def\.filenameA#1/#2{\_ifx\.end#2#1\_else\_afterfi{\.filenameA#2}\_fi}

\_def\.fileext#1{\_ea\.fileextA#1.\.end}
\_def\.fileextA#1.#2{\_ifx\.end#2#1\_else\_afterfi{\.fileextA#2}\_fi}
```

MIME type is determined from file extension (e.g. `mp4` is “video/mp4”). For mapping of file extensions to MIME types we abuse TEX’s hash table which gets populated with “known MIME types”. This necessarily means that the database is incomplete. Users can define their own additional mappings, or they can contribute generally useful ones to this package.

The default MIME type (used for unknown file extensions) is “application/octet-stream” – binary data.

The uninteresting MIME type database itself is at the very end (2.9).

pdfextra.opm

```
\_def\.mimetoname[#1/#2]{/#1\_csstring\#2F#2}

\_def\.defaultmimetype{application/octet-stream}
\_def\.exttomime#1{\.trycs{mimetype:#1}{}}
```

Here we define an OpTEX style “is-macro” that checks whether the file has already been defined – `\.isfiledefined`{⟨*name*⟩}`\iftrue` (or `\iffalse`). The case where the file has not been defined using `\filedef` can be handled in a lot of ways. As a default we interpret ⟨*name*⟩ as path and try to embed it. Because the path from ⟨*name*⟩ is used as the “friendly name” the file will be embedded only once even when requested more times.

pdfextra.opm

```
\_def\.isfiledefined#1#2{\.isdefined{filespec:#1}\_iftrue\_else
   \_afterfi{\.fileundefined{#1}}\_fi#2%
}

\_def\.fileundefined#1{\_isfile{#1}\_iftrue\.filedef/e[#1]{#1}\_else
   \_opwarning{file '#1' not found, ignored}\_ea\_unless\_fi
}

% strict requirement of preceeding `\filedef` can be set like this:
%\_def\.fileundefined#1{\_opwarning{file '#1' is not defined, ignored}\_unless}
```

2.4 PDF actions

The core of interactivity in PDF are actions. They are all initialy handled by `\pdfaction[`⟨*action spec*⟩`]`. ⟨*action spec*⟩ is a comma separated list of ⟨*type*⟩`:`⟨*arguments*⟩. Leading spaces in the elements of the list are ignored using undelimited-delimited argument pair trick.

An invocation could look like this:

```
\pdfaction[
   js:{app.alert("Yay JavaScript, going to page 5");},
   ilink:pg:5,
   transition:Wipe,
]
```

This is why we have to be very careful when loading the contents between `[]` to arguments. In particular, we can't split immediatly using `[#1:#2]`, because this would discard the braces guarding the comma in the JavaScript code. However we also need to find out the *type* of action which is taken as a type of the first action (`js` in this case). `\.pdfactiontype[`⟨*action spec*⟩`]` does this – we don't mind that there the braces are lost.

`\pdfaction` processes the list, to create a chain of actions using `/Next` field. The handling of each action type is up to macro `\_pdfextra_`⟨*type*⟩`action`, which receives `[`⟨*type*⟩`:`⟨*arguments*⟩`]`. Because of this a single type handler can handle multiple different actions, as is the case with `\.ilinkaction` which is the fallback for unknown action types.

pdfextra.opm

```
\_def\.pdfaction[#1#2]{\.pdfactionA#1#2,\.stop\.end}
\_def\.pdfactionA#1,#2#3\.end{%
   <<%
   \.pdfactionB[#1]%
   % next action
   \_ifx\.stop#3\_else\_space
      /Next \_afterfi{\.pdfactionA#2#3\.end} % intentional space
   \_fi
   >>
}
\_def\.pdfactionB[#1:#2]{\.trycs{#1action}{\_ea\.ilinkaction}[#1:#2]}
\_nspublic \pdfaction ;

\_def\.pdfactiontype[#1:#2]{#1}
```

2.4.1 Additional actions

Some PDF objects, like pages and some annotations, can also have "additional actions". These are actions which will be executed when an event happens – like page getting opened for `/O` action in page's additonal actions or `/PO` in annotation's additional actions. For constructing these additional actions we define a helper macro `\.pdfaactions`. The use is as something follows:

```
/AA << \.pdfaactions{ {O} {⟨action spec 1⟩} {C} {⟨action spec 2⟩} } >>
```

To produce something this:

```
/AA << /O <<⟨action 1⟩>> /C <<⟨action 2⟩>> >>
```

pdfextra.opm

```
\_def\.pdfaactions#1{<<\.pdfaactionsA #1\.end\.end>>}
\_def\.pdfaactionsA#1#2{\_ifx\.end#1\_else /#1 \_ea\.pdfaction\_ea[#2]\_ea\.pdfaactionsA\_fi}
```

2.4.2 Link annotations

The main use of actions – annotations of `/Subtype /Link`. Annotation of this type creates an active rectangular area on the page that executes a PDF action (or chain of them in the general case). `\hlink[`⟨*action spec*⟩`]`⟨*text*⟩ is macro that typesets ⟨*text*⟩ and makes area occupied by it active according to ⟨*action spec*⟩. All action types are supported, the mechanism is completely generic.

The `\pdfstartlink`/`\pdfendlink` primitives are used to denote the part of the page where ⟨*text*⟩ appears as active. LuaTEX will then handle even the situations where ⟨*text*⟩ gets broken across multiple lines (by creating multiple rectangular annotations to cover all `\hbox`es).

pdfextra.opm

```
\_def\.hlink[#1]#2{\_bgroup\_def\#{\_csstring\#}%
    \_edef\.type{\.pdfactiontype[#1]}%
    \_quitvmode\_pdfstartlink \.linkdimens
       attr{\_pdfborder{\.type}}%
       user{/Subtype /Link /A \.pdfaction[#1]}\_relax
    \_localcolor\.linkcolor{\.type}#2\_pdfendlink\_egroup
}

\_nspublic \hlink ;
```

Use `\hlink` as the backing command for OpTeX's "higher level" linking commands (`\ilink` and `\ulink`).

The lower level ones (`\xlink` and its predecessor `\link` actually have completely different semantics with regards to color, so we keep them as they are.

pdfextra.opm

```
\_protected\_def\_ilink[#1]#2{\.hlink[#1]{#2}}
\_protected\_def\_ulink[#1]#2{{\_escapechar=-1 \_ea}\_expanded
   {\_noexpand\.hlink[url:\_detokenize{#1}]}{#2}}

\_public \link \ilink \ulink ;

%\_protected\_def\_link[#1]#2#3{\_hlink[#1]{#3}}
%\_protected\_def\_xlink#1#2#3#4{\_hlink[#1:#2]{#4}}
```

Two customizations of `\hlinks` are possible:

- Dimensions of rectangular areas created by `\pdfstartlink`/`\pdfendlink`. This is done using `\.linkdimens` (analogous to OpTeX's `\linkdimens`). Dimensions that are unset are taken from the respective `\hbox`es. `\lininglinks` sets the dimensions for running text – it covers all space of a line using `\baselineskip`. `\nolininglinks` sets no dimensions, this is useful for buttons, that may have larger height/depth than a line.
- The color is determined from the type of link (that is, the first action in ⟨*action spec*⟩) by checking `\_`⟨*type*⟩`linkcolor` (compatible with OpTeX). As a fallback `\_ilinkcolor` is used (set by OpTeX's `\hyperlinks`) for all links except for URLs, where `\_elinkcolor` is used instead. If even these fallback colors are not defined (`\hyperlinks` isn't used), then the most generic `\_linkcolor` will be taken or no color will be set.

pdfextra.opm

```
\_def\.lininglinks{%
   \_def\.linkdimens{height.75\_baselineskip depth.25\_baselineskip}%
}
\_def\.nolininglinks{\_def\.linkdimens{}}
\.lininglinks

\_nspublic \lininglinks \nolininglinks ;

\_def\.linkcolor#1{\_trycs{_#1linkcolor}{\_trycs{_ilinkcolor}{\_trycs{_linkcolor}{}}}}

 % \_urllinkcolor = \_elinkcolor with fallbacks
\_def\_urllinkcolor{\.linkcolor{e}}
```

2.4.3 Open action

The document itself has one action defined in the document catalog. It is called `/OpenAction`. We allow the user to set it using the familiar ⟨*action spec*⟩ syntax with the command `\openaction[`⟨*action spec*⟩`]`.

Internally we could directly set it by appending to the catalog using the primitive `\pdfcatalog`, but LuaTeX (pdfTeX really) allows setting the action with special syntax. This has the benefit that it is not allowed to set the action more than once.

pdfextra.opm

```
\_def\.openaction[#1]{\_pdfcatalog{} openaction user{\.pdfaction[#1]}\_relax}
\_nspublic \openaction ;
```

2.4.4 Jump actions

These are the most typical actions. Even LuaTeX itself handles them, although we don't use the possibility for maintaining generality. There are a few types of jump actions:

- `/GoTo` actions are the classic internal links to named destinations in the PDF file (created by `\pdfdest` primitive or OpTeX's `\dest`). The destination names include also the type of internal link (e.g. `ref:section1`). They are handled by `\.ilinkaction[`⟨*type*⟩`:`⟨*name*⟩`]`.
- `/URI` actions which are in most cases used as "goto URL" actions. These are not that useful directly, because special characters should be handled before this actions is used (like with `\url`). The low level use is `\.urlaction[url:`⟨*url*⟩`]`.
- "Goto remote" actions, which can jump to a destination in another PDF file – either determined by name, or by page number. The external files are expected to be defined by `\filedef` (but not the embedded variant). The use is either `\.extrefaction[extref:`⟨*name*⟩`:`⟨*named destination*⟩`]` for links to named destination or `\.extpgrefaction[extpgref:`⟨*name*⟩`:`⟨*page number*⟩`]` for page destinations. Customization is possible with `\.extrefextra`, by default opening in a new windows is requested.

pdfextra.opm

```
\_def\.ilinkaction[#1:#2]{/S /GoTo /D (#1:#2)}

\_def\.urlaction[#1:#2]{/S /URI /URI (#2)}

\_def\.extrefaction[#1:#2:#3]{/S /GoToR
   /F \.cs{filespec:#2}
   /D (#3)
   \.extrefextra
}
\_def\.extpgrefaction[#1:#2:#3]{/S /GoToR
   /F \.cs{filespec:#2}
   /D [\_the\_numexpr#3-1\_relax\_space /Fit]
   \.extrefextra
}

\_def\.extrefextra{/NewWindow true}
```

Transition action is not really a jump action in of itself, but is only useful when chained after jump actions, so we define it here. Transitions (as page attributes) are handled more thoroughly in section 2.5.1.

The use would look something like: `\.transitionaction[transition:`⟨*animation type*⟩`:`⟨*duration*⟩`:`⟨*raw PDF attributes*⟩`]`, where all fields omitted from right take the default values.

pdfextra.opm

```
\_def\.transitionaction[#1:#2]{/S /Trans \.attrorempty{Trans}{\.maketrans[#2]}}
```

2.4.5 Named actions

User can request arbitrary "named" action with `\.namedaction[named:`⟨*name*⟩`]`. See user documentation for details.

pdfextra.opm

```
\_def\.namedaction[#1:#2]{/S /Named /N /#2}
```

2.4.6 JavaScript actions

JavaScript actions have two forms, either `\.jsaction[js:`⟨*name*⟩`]` or `\.jsaction[js:`⟨*script*⟩`]`. The first variant uses contents of `\filedef`'d ⟨*name*⟩, the second one uses ⟨*script*⟩ directly. There is no special catcode handling.

pdfextra.opm

```
\_def\.jsaction[#1:#2]{/S /JavaScript
   /JS \_ifcsname _pdfextra_filestream:#2\_endcsname \_lastnamedcs \_else
      (#2)
   \_fi
}
```

2.5 Page attributes

PDF represents pages as dictionaries. The dictionaries get generated by LuaTEX, which fills in some attributes *attributes* (like `/Content` with contents of the page and `/Annots` with array of annotations). We can add more using `\pdfpageattr` primitive token list register. While not that many are generally useful, there are a few interesting ones. For example transitions can be set using page attributes, or we might want to set additional actions (`/AA`) to listen for page events.

While the so called "page objects" are in a tree structure (for fast lookup), only the leaves are real "pages". PDF allows some attributes to be inherited from parent page objects, but not all of them and certainly not those we are interested in.

The mechanism introduced in this section is optional, because it takes complete control over `\pdfpageattr`. It gets activated when `\initpageattributes` is first used (which happens automatically for some functionality exposed by this package), but may be activated by the user for any other purpose. Only attributes listed in `\pageattributes` are processed.

We set the attributes anew for each page, by hooking into OpTEX's `\_begoutput`. Because `\pdfpageattr` token list doesn't get expanded before written out to PDF, we expand it using the assignment in `\edef` trick. The token list gets expanded, but the assignment is not made until it reaches main processor when the temporary control sequence gets expanded.

pdfextra.opm

```
% pdfpagattr managament (default for all pages vs current page override)
\_def\.pageattributes{{Trans}{Dur}{Rotate}{AA}}
\_def\.initpageattributes{%
   % add hook for setting primitive \pdfpageattr
   \_addto\_begoutput{\_edef\.tmp{\_pdfpageattr={\.pdfpageattributes}}\.tmp}%
   % no need to do this twice
   \_let\.initpageattributes=\_relax
}
\_nspublic \pageattributes \initpageattributes ;
```

The user interface we want to expose has two parts:

- setting the page attribute for just this one page (`\.pdfcurrentpageattr`),
- setting the default attribute (used when current page value is not set) (`\.pdfdefaultpageattr`).

The first one of course brings in the typical TEX problem of knowing the page where something occurs. As always, the page number contained in `\gpageno` during processing of said content may of course not actually be the number of the page where the content ends up! Hence, we need to note the page number with a delayed write, using `\.setpageof` and later `\.pageof`. The different settings of page attributes should have distinct names, we use the `\.pageattrcount` counter for this.

pdfextra.opm

```
\_newcount\.pageattrcount
\_def\.pdfcurrentpageattr#1#2{\.initpageattributes
   \_incr\.pageattrcount
   \.setpageof{pageattr:\_the\.pageattrcount}%
   \.sxdef{pdfpgattr:\.pageof{pageattr:\_the\.pageattrcount}:#1}{#2}%
}
\_def\.pdfdefaultpageattr#1#2{\.initpageattributes
   \.sxdef{pdfpgattr:#1}{#2}%
}
```

Finally, the macro `\pdfpageattributes` takes care of setting generating the contents of `\pdfpageattr`. For each attribute in `\pageattributes` it first checks its current page value, only then the default value. If neither is set, nothing is added.

pdfextra.opm

```
\_def\.pdfpageattributes{\_ea\.pdfpageattributesA\.pageattributes\.end}
\_def\.pdfpageattributesA#1{\_ifx\.end#1\_else
   % use current page override or "default"
   % don't emit anything if the value is empty
   \.attrorempty{#1}{%
      \.trycs{pdfpgattr:\_the\_gpageno:#1}{\.trycs{pdfpgattr:#1}{}}%
   }%
   \_ea\.pdfpageattributesA\_fi
}
```

Each attributes then has two switches for the respective default and current values. For defining a few of them a helper is introduced:

`\.pdfpageattributesetters` ⟨*attribute*⟩ \⟨*default setter*⟩ \⟨*current setter*⟩ {⟨*value*⟩},

where ⟨*attribute*⟩ is name of the attribute without the slash (e.g. `MediaBox`), the two control sequences name the future user setters, which will take single argument in brackets (e.g. `\mediabox` and `\thismediabox`) and the ⟨*value*⟩ can use the argument.

pdfextra.opm

```
\_def\.pdfpageattributesetters#1 #2#3#4{%
   \.sdef{\_csstring#2}[##1]{\.pdfdefaultpageattr{#1}{#4}}%
   \.sdef{\_csstring#3}[##1]{\.pdfcurrentpageattr{#1}{#4}}%
   \_nspublic #2 #3 ;
}
```

Some of the useful attributes are `/Rotate`, which rotates the pages visually (can be set with `\defaultpagerotate` and `\pagerotate`), and the additional actions (`/AA`, see section 2.4.1, set using `\defaultpageactions` `\pageactions`).

pdfextra.opm

```
\.pdfpageattributesetters Rotate \defaultpagerotate \pagerotate {#1}

\.pdfpageattributesetters AA \defaultpageactions \pageactions {\.pdfaactions{#1}}
```

2.5.1 Transitions, page durations

There are predefined types of transitions, like `/Wipe`, `/Box`, `/Split`, etc. Most have other customizible attributes – usually directions set in different ways depending on the animation type at hand, but the most important attribute is the duration of the animation. Parsing friendly user notation in the form of `[`⟨*animation type*⟩`:`⟨*duration*⟩`:`⟨*other raw attributes*⟩`]`, where fields from the right may be omitted to produce the default value, is handled by `\.maketrans`. This macro is also used by transition actions (see 2.4.4). The defaults are simply those defined by PDF standard (no transition, 1 second duration and the respective default directions).

pdfextra.opm

```
\_def\.maketrans[#1]{\.maketransA#1:::\.end}
\_def\.maketransA#1:#2:#3:#4\.end{%
   \.emptyor{#1}{<</S /\.nonempty \.attrorempty{D}{#2} #3>>}
}
```

The attribute setters for transitions (`\transitions`, `\transition`) are a simple wrappers. Similiar is the setting of page duration in seconds after which PDF viewer automatically advances to the next page (`\defaultpageduration`, `\pageduration`).

pdfextra.opm

```
\.pdfpageattributesetters Trans \transitions \transition {\.maketrans[#1]}

\.pdfpageattributesetters Dur \defaultpageduration \pageduration {#1}
```

2.6 Name trees – attachments and document level JavaScript

These don't have any last place to be in, so they are documented separately, here. Attaching files using `/FileAttachment` annotations:

1. is intended more towards viewers of the document for extra additions and
2. doesn't work in the viewers as well as one would like.

That is why instead embed files using normal `\filedef` and then allow them to be added to the document level `/EmbeddedFiles` entry, which means they will be shown in the user interface by PDF viewers. `/EmbeddedFiles` is a document level name tree (contained inside `/Names` entry of `/Catalog`) that maps names of files to their objects. Although we simplify matters by constructing more of an array.

What works very similiarly is document level JavaScript. It is a name tree within `/JavaScript` field. It maps names of JavaScript actions to their object numbers. The names aren't very useful, but the actions have their purpose. They are executed in turn after the document is opened. Hence they can be used to predefine JavaScript functions in the global context, to be used later within actions explicitly activated by the user.

The user level commands are `\attach[`⟨*name*⟩`]` (to attach a previously `\filedef`'d name with fallback to embedding now if it is a valid path) and `\dljavascript[`⟨*name*⟩`]{`⟨*script*⟩`}` (adds action that executes ⟨*script*⟩ after document is opened, ⟨*name*⟩ is more or less meaningless).

Internally the commands construct lists of what ends up in the resulting name array, i.e. pairs `(`⟨*name*⟩`)`␣⟨*object*␣*number*⟩␣`0`␣`R`␣. Intermediate macros `\.embeddedfiles` and `\.dljavascripts` are used for this.

In the case of file attachments, nothing happens if file is defined and not found by the fallback.

pdfextra.opm

```
% file attachment
\_def\.embeddedfiles{}
\_def\.attach[#1]{\.isfiledefined{#1}\_iftrue
   \.xaddto\.embeddedfiles{(#1) \.cs{filespec:#1} }\_fi
}
\_nspublic \attach ;

\_def\.dljavascripts{}
\_def\.dljavascript[#1]#2{%
   \_immediate\_pdfobj{<< \.jsaction[js:{#2}] >>}%
   \.xaddto\.dljavascripts{(#1) \_the\_pdflastobj \_space 0 R }%
}
\_nspublic \dljavascript ;
```

Renditions (see 2.8.1) also need their name tree. This package mostly doesn't play well with Unicode filenames, that is why they are forbidden. However, Renditions that are accessed from JavaScript have

to be named/present in a `/Renditions` name tree, with the names encoded in the PDF encoding (UTF-16BE).
The names and object references are collected in `\.renditions`. Unicode encoding is hacked with `\.pdfstringtounicode`.

pdfextra.opm

```
\_def\.pdfstringstrip(#1){#1}%
\_def\.pdfstringtounicode#1#2{%
   \_ea\_pdfunidef\_ea#1\_ea{\_ea\.pdfstringstrip\_expanded{\.cs{filename:#2}}}%
}
\_def\.renditions{}
```

Object creation, which is common to all name trees, is handled by
`\.makenamearray`⟨*name tree name*⟩⟨*name tree content*⟩.
It doesn't do anything for empty lists, to not bloat PDF files when this mechanism isn't used.

pdfextra.opm

```
\_def\.makenamearray#1#2{\_ifx#2\_empty\_else
   \_immediate\_pdfobj {<< /Names [ #2 ] >>}%
   \_pdfnames{/#1 \_the\_pdflastobj \_space 0 R }\_fi
}
```

The lists themselves can only be written out to the PDF file at the very end of the run. We use OpTeX's `\_byehook`, which is run in `\_bye`. But `\bye` itself may be predefined by the user, for example when using some of the OpTeX tricks. We just hope that the user keeps `\_byehook`.

pdfextra.opm

```
\_addto\_byehook{%
   \.makenamearray{EmbeddedFiles}\.embeddedfiles
   \.makenamearray{JavaScript}\.dljavascripts
   \.makenamearray{Renditions}\.renditions
}
```

2.7 Viewer preferences

There are a few customizations of display (and other preferences of PDF viewers) possible in the document catalog or its subdictionary `/ViewerPreferences`. Most are not that useful. The interesting ones are implemented by `\fullscreen`, `\showoutlines`, `\showattached`. They all set the page mode using `\.setpagemode`. We don't handle respecting the last setting (using `\_byehook`). To prevent invalid PDF files, we set `\.setpagemode` to `\_relax` after use.

pdfextra.opm

```
\_def\.setpagemode#1{\_pdfcatalog{/PageMode /#1}\_glet\.setpagemode=\_relax}

\_def\.fullscreen{\.setpagemode{FullScreen}}
\_def\.showoutlines{\.setpagemode{UseOutlines}}
\_def\.showattached{\.setpagemode{UseAttachments}}

\_nspublic \fullscreen \showoutlines \showattached ;
```

Only the setting of duplex / double sided printing and display is in the nested dictionary. It is handled by `\duplexdisplay`. The simplistic version does not handle more attributes in `/ViewerPreferences`. We also set the meaning to `\_relax` to prevent more (erroneous) uses.

pdfextra.opm

```
\_def\.duplexdisplay{\_pdfcatalog{%
  /PageLayout /TwoPageRight
  /ViewerPreferences <<
    /Duplex /DuplexFlipLongEdge
  >>}%
  \_glet\.duplexdisplay=\_relax
}

\_def\duplexdisplay{\.duplexdisplay}
```

2.8 Multimedia

PDF essentially allows insertion of different types of multimedia:

- images,
- audio/video,
- 3D art.

The first is pretty standard and handled normally by the engine (LuaTEX). Others are possible, but have to be done manually according to one of the mechanisms specified by PDF standard:

- Sounds (audio only),
- Movies (video and/or audio),
- Renditions (video and/or audio),
- 3D annotations (3D art),
- Rich Media (video and/or audio, 3D art)

Sadly all these mechanisms are badly flawed, each in different ways. At least we try to use the one that works in the viewers.

For audio/video "Movies" are the simplest mechanism, but they have been deprecated in PDF 2.0 and no longer work in Acrobat/Foxit (same for "Sounds").

"Renditions" are complicated, partly dependant on JavaScript, but at least supported by Acrobat, Foxit, Evince and Okular.

"Rich media" annotations were designed for Flash. This use case is no longer possible today, but the obscurities remain. They are unnecessiraly complicated, but can be used without Flash too. Although the result is very plain for audio/video – no controls can be displayed and there are no associated actions.

"3D annotations" are reasonably simple, but also flawed. They cannot reuse embedded file as a source for 3D data. Hence it is better and more consistent to use Rich Media for 3D annotations. It even has additional benefits, like the possibility of using multiple initialization scripts.

In the end, this package exposes two user commands corresponding to two mechanisms – first are Renditions (`\render`) for audio/video that works in most browsers and Rich Media (`\RM`) mainly for 3D art, but also for audio/video with limited possibilities.

Both mechanisms have an annotation at their core. Annotations is essentially a rectangular area on page. The area corresponds to where the multimedium will show up. After activating the area somehow (by user click, or action) the multimedium will start playing. Before annotations the rectangular area will show something that is called "normal appearance". This appearance is of type form XObject. Those are really similiar to pages – they have dimensions, contents made up of PDF graphics operators, . . . , but they are reusable. Not that useful for annotations where we will need the form only once, but nice anyways. pdfTEX has primitives for creating them – `\pdfxform` and friends. They essentially do the same code like `\shipout` does, but instead of page, they make this reusable object. One can then either use this reusable object in another page/form, but we will indirectly refer to it for the appearance.

Important aspect of annotations is that they are really only rectangular areas on the page, but they are not really part of the page. They sort of sit on another level and are not influenced by PDF graphic operators which make the page. In pdfTEX annotations are handled by *whatsit* nodes. While most nodes map to known primitive TEX concepts (like typeset characters, boxes, rules, etc.) Whatsits are essentially commands for TEX that are delayed until page is being shipped out (written to PDF file). `\write`, `\special`, and most pdfTEX commands create whatsits. For annotationos this is important, because this means that the engine only stores the information about annotation that we specify, but creates it at due time, when it should be written to PDF.

Because whatsits are essentially dimensionless and we want it to be a part of normal TEX typesetting material we create the annotation (whatsit) in `\hbox`. This box will be otherwise empty, because the apperance of the rectangular area is determined by the normal appearance field (`/N` in `/AP`). We set the dimensions of the box to the dimensions of normal appearance. Everything will line up nicely, because when processed, the annotation will take dimensions from the box.

All of these concepts are implemented in:

`\.boxedannot[`⟨*type*⟩`:`⟨*name*⟩`]{`⟨*appearance*⟩`}{`⟨*special text*⟩`}{`⟨*annotation attributes*⟩`}`
⟨*type*⟩ is used to determine the annotation border (same principle as with Link annotations, section 2.4.2), ⟨*name*⟩ will be used as the annotation name (`/NM`), ⟨*special text*⟩ is used for influencing the `\pdfannot` primitive, and ⟨*annotation attributes*⟩ will become the body of the annotation.

pdfextra.opm

```
\_def\.boxedannot[#1:#2]#3#4#5{%
   \_setbox0=\_hbox{#3}\_setbox2=\_null
   \_ht2=\_ht0 \_wd2=\_wd0 \_dp2=\_dp0
   \_preshipout0 \box0
   \_immediate\_pdfxform0
   % box with annotation both stretching to dimensions of appearance
   \_hbox{\.setpageof{#1:#2}%
      \_pdfannot #4 {#5
```

```
         /AP <</N \_the\_pdflastxform \_space 0 R>>
         \_pdfborder{#1}
         /NM (#2)
         /Contents (#1 '#2')
      }%
      \_copy2
   }%
}
```

There is another weird thing common to both multimedia mechanisms – the redefinition of `\.name`. It is initially set by `\.secondoptdef` to ⟨*name*⟩, but may be redefined by user supplied `name` key-value parameter. This should be used when there are multiple uses of the same content. Otherwise samely named annotations would be indistinguishable both for PDF viewer and our handling of actions (which would all refer only to the first instance).

To somewhat overcome this, trying to use the same ⟨*name*⟩ (within the same type of annotaiton) will use dummy name from `\.unnamedannotcount` (for uniqueness). This means that ⟨*name*⟩ will always refer to the first instance. `\.redefinename` handles this.

pdfextra.opm

```
\_newcount\.unnamedannotcount
\_def\.redefinename#1{%
   \.isdefined{#1:\.name}\_iftrue
      \_incr\.unnamedannotcount
      \_edef\.name{\_the\.unnamedannotcount}%
   \_else
      \_edef\.name{\_kv{name}}%
   \_fi
}
```

2.8.1 Renditions (audio/video)

There are three main types of PDF objects involved in the Renditions ("Multimedia") mechanism:

- Screen annotations define the area for playing multimedia.
- Rendition objects define the multimedia to play.
- Rendition actions associate Rendition objects with Screen annotations.

You can theoretically arbitrarily mix and match rendition objects and screen annotations by invoking different actions. In practice Evince and Okular do really simplistic parsing and don't fully support the actions fully. But by keeping it simple it is possible to make it work almost the same in all viewers that support renditions.

Different sources of audio/video should be possible. In fact all three file specifications (embedded files, files specified by URL/path) could work. Again in practice embedded file is the safest bet, that works in all viewers that support renditions.

The user facing command is:

`\render`[⟨*name*⟩][⟨*optional key-value paramers*⟩]{⟨*horizontal material*⟩}

⟨*name*⟩ is the friendly name set using `\filedef` or file path if ⟨*name*⟩ isn't `\filedef`d and is to be embedded. The key-value parameters in brackets can be entirely omitted. They can influence the playback (except for `controls` most are not well supported). Default values are taken from `\.renderdefaults`.

`\render` doesn't do anything (except print warning) if file ⟨*name*⟩ isn't defined and ⟨*name*⟩ isn't a path to file that can be embedded.

The first PDF object it defines is Rendition, which specifies information about the multimedium (name, file specification, MIME type and options from key-value parameters). Some of the fields are in `/BE` ("best effort") dictionaries. This is due to the very general design of Renditions, which theoretically allows the PDF viewer to choose from multiple Renditions if they know they can't support some of the requested features. But that is not much useful in practice, so we just don't complicate it.

Next defined object is Screen annotation, which complicates thing by requiring (`/P`) reference to the page where the annotation is (handled by `\setpageof` and `\pageof` pair). Important field is `/A` which specifies actions that shall be executed when the screen area is clicked. We let the user change the action, but the sensible default of starting to play the multimedium is used (and this is the only thing that works in some viewers anyways). Additional actions `/AA` may be used to react to events like mouse over or page open/close – the most probable use case is autoplay on page open, for which shortcut of `\renditionautoplay` is defined.

The code is slightly complicated by the fact, that actions need to reference the Rendition and Screen objects. In the case of the action contained in Screen annotation this essentialy involves a self reference. Hence it is needed to first reserve an object number and later use it for the annotation. Because the object numbers may also be needed by actions defined later, we need to save them to `\_pdfextra_rendition:`⟨*name*⟩ and `\_pdfextra_screen:`⟨*name*⟩ respectively, but also define aliases with empty names, so users can easily reference the latest rendition.

pdfextra.opm

```
\.secondoptdef\.render#1{\.isfiledefined{\.name}\_iftrue\_bgroup
   \_ea\_readkv\_ea{\_ea\.renderdefaults\_ea,\_the\_opt}%
   \.colortorgbdef\.bgcolor{\_kv{background}}%
   % rendition object ("media specifaction")
   \.pdfstringtounicode\.uiname\.name
   \_pdfobj {<</Type /Rendition
      /S /MR
      /N \.uiname
      /C <<%/Type /MediaClip
        /S /MCD % subtype MediaClipData
        /D \.cs{filespec:\.name}
        /CT (\.cs{filemime:\.name})
        /P << /TF (TEMPALWAYS) >> % allow creating temporary files
      >>
      /P <<%/Type /MediaPlayParams
        /BE << /C \_kv{controls} /V \_kv{volume} /RC \_kv{repeat} >>
      >>
      /SP <<%/Type /MediaScreenParams
        /BE << /O \_kv{opacity} /B [\.bgcolor] >>
      >>
   >>}\_pdfrefobj\_pdflastobj
   \.xaddto\.renditions{\.uiname \_the\_pdflastobj \_space 0 R }%
   \.redefinename{rendition}%
   \.sxdef{rendition:\.name}{\_the\_pdflastobj}%
   % screen annotation ("screen space allocation")
   \_pdfannot reserveobjnum% "self" reference will be needed inside screen annot.
   \.sxdef{screen:\.name}{\_the\_pdflastannot}%
   % aliases to latest rendition/screen with empty name
   \_global\.slet{rendition:}{rendition:\.name}%
   \_global\.slet{screen:}{screen:\.name}%
   \_edef\.action{\_kv{action}}\_edef\.aactions{\_kv{aactions}}%
   \.boxedannot[rendition:\.name]{#1}{useobjnum\_the\_pdflastannot}{%
      /Subtype /Screen
      % reference to page of the rendition (\setpageof done by \.boxedannot)
      % the spaces are weird, but \pdfpageref eats them
      /P \_pdfpageref\.pageof{rendition:\.name} \_space 0 R
      /A \_ea\.pdfaction\_ea[\.action]
      /AA \_ea\.pdfaactions\_ea{\.aactions}
   }%
   \_egroup\_fi
}
\_nspublic \render ;
```

Here are the defaults used for `\render` – `\.renderdefaults`. Users can redefine them all together or override as needed with key-value parameters. The defaults correspond to values specified by PDF standard. Other values may not be respected by all viewers.

pdfextra.opm

```
\_def\.renderdefaults{%
   name=\.name,
   controls=false,
   volume=100,
   repeat=1,
   opacity=1.0,
   background=1 1 1,
   action=rendition::play,
   aactions={},
}
```

Most probable use of additional actions is to start auto-start playing of the multimedium. For this purpose `\renditionautoplay` is defined as a shorthand for action to play the lastly defined rendition on page visible event.

pdfextra.opm

```
\_def\.renditionautoplay{{PV}{rendition::play}}
\_nspublic \renditionautoplay ;
```

Rendition actions

Rendition actions unfortunately use cryptic symbolic numbers (`0`, `1`, `2` and `3`) for actions that could be called `play`, `stop`, `pause` and `resume` respectively. Except for these predefined actions (that use `/OP`) running of JavaScript is possible using `/JS (⟨script⟩)` with potential fallback to `/OP`. This is dangerous teritory, because support of the right API in the viewer is very low. Although it is possible to define such action type by:

```
\.sdef{renditionaction:myaction}{/JS (app.alert("something useful");) /OP 0}
```

The use of rendition action is: `\.renditionaction[rendition:⟨name⟩:⟨action type⟩]`. Empty name refers to last rendition, so e.g.`\.renditionaction[rendition::pause]` is possible.

pdfextra.opm

```
\.sdef{renditionaction:play}{/OP 0}
\.sdef{renditionaction:stop}{/OP 1}
\.sdef{renditionaction:pause}{/OP 2}
\.sdef{renditionaction:resume}{/OP 3}
\_def\.renditionaction[#1:#2:#3]{/S /Rendition
   \.cs{renditionaction:#3}
   /R \.cs{rendition:#2} 0 R
   /AN \.cs{screen:#2} 0 R%
}
```

2.8.2 Rich Media (3D/audio/video)

Some principles seen with Renditions (section 2.8.1) apply here too. But additionally we deal with 3D specifics and unfortunate Flash leftovers.

Unlike Renditions both page area and multimedium specfaction are handled in a single annotation – the Rich Media annotation. The code is unfortunately obscured due to the weird requirements, but this is essentially what we are trying to create with `\RM`:

```
/Type /Annot
/Subtype /RichMedia
/RichMediaSettings <<
  /Activation <<
    /Condition /PV
    /Scripts [ 14 0 R ]
  >>
  /Deactivation << /Condition /XD >>
>>
/RichMediaContent <<
  /Assets << /Names [ (kladka.prc) 2 0 R (wireframe.js) 14 0 R ] >>
  /Configurations [ <<
      /Type /RichMediaConfiguration
      /Subtype /3D
      /Instances [ <<
          /Type /RichMediaInstance
          /Subtype /3D
          /Asset 2 0 R
        >> ]
    >> ]
>>
```

The activation/deactivation will be dealt with later. But we see that to insert a simple 3D file, we have to pack it inside a file specification (indirect reference to object `2 0 R`), then in “instance”, inside a “configuration” inside “content”. As if it wasn’t enough the names (normally contained in the file specification) have to be specified again in `Assets` name tree that uselessly maps names to file specifications. Because this is a 3D Rich Media annotation there are other files at play – initialization scripts. These are specified in `/Scripts` and are executed in turn when the annotation is activated. Not shown is, that some “configurations” and “instances” actually have to be specified indirectly.

If it wasn't for Flash we could do with something like:

```
/Type /Annot
/Subtype /RichMedia
/Activation /PV
/Scripts [ 14 0 R ]
/Deactivation /XD
/Content 2 0 R
```

Which contains equivalent information. But unfortunately here we are…

pdfextra.opm

```
\.secondoptdef\.RM#1{\.isfiledefined{\.name}\_iftrue
      \_edef\.tmp{\.cs{filemime:\.name}}%
      \_edef\.subtype{\_ea\.mimetormsubtype\_ea[\.tmp]\_space}%
      \_ifx\.subtype\_space
         \_opwarning{unknown rich media type for '\.name', ignored}\_else
   \_bgroup
   \_ea\_readkv\_ea{\_ea\.RMdefaults\_ea,\_the\_opt}%
   % Instance that has the media file as an asset
   \_pdfobj {<</Type /RichMediaInstance
     /Subtype /\.subtype
     /Asset \.cs{filespec:\.name}
   >>}\_pdfrefobj\_pdflastobj
   % Configuration with one single instance (the above)
   \_pdfobj {<</Type /RichMediaConfiguration
     /Subtype /\.subtype
     /Instances [ \_the\_pdflastobj \_space 0 R ]
   >>}\_pdfrefobj\_pdflastobj \_edef\.configuration{\_the\_pdflastobj}%
   \_edef\.names{\.cs{filename:\.name} \.cs{filespec:\.name} }% initial asset
   \.redefinename{rm}%
   \_def\.scriptfilespecs{}%
   \_edef\.views{\_kv{views}}\_edef\.scripts{\_kv{scripts}}%
   \_ifx\.views\_empty \_edef\.views{\.name}\_fi
   \_ea\.DDDscripts\_ea{\.scripts}%
   % annotation in hbox
   \.boxedannot[rm:\.name]{#1}{}{%
      /Subtype /RichMedia
      /RichMediaSettings <<
        /Activation <<
          /Condition \.cs{activation:\_kv{activation}}
          \.emptyor{\.scriptfilespecs}{/Scripts [ \.nonempty ]}
          /Presentation << /Toolbar \_kv{toolbar} \.RMpresentationextra >>
        >>
        /Deactivation << /Condition \.cs{deactivation:\_kv{deactivation}} >>
      >>
      /RichMediaContent <<
        /Assets << /Names [ \.names ] >>
        /Configurations [ \.configuration \_space 0 R ]
        \.emptyor{\_ea\.DDDviews\_ea{\.views}}{/Views [ \.nonempty ]}
      >>
   }%
   \.sxdef{rm:\.name}{\_the\_pdflastannot}%
   \_global\.slet{rm:}{rm:\.name}%
   \_egroup\_fi\_fi
}
\_nspublic \RM ;
```

The code is similiar to `\render`, but we also ignore everything if we don't recognize the type of media (`Video`, `Sound` or `3D`). For that we use a simple mapping from MIME types with `\.mimetormsubtype`. This means that although we aim Rich Media mostly for 3D art it may also be used for Video and Sound.

pdfextra.opm

```
\_def\.mimetormsubtype[#1/#2]{\.cs{rmtype:#1}}

\.sdef{rmtype:model}{3D}
\.sdef{rmtype:video}{Video}
\.sdef{rmtype:audio}{Sound}
```

Then we also need to construct the weird name "tree" (essentialy an array in our case) and script array. `\.DDDscripts` and `\.DDDviews` do this. Name tree is accumulated in `\.names`, and starts with the media file. After that each script is added to `\.names` and `\.scriptfilespecs`. The scripts are passed as a comma separated array. Ignoring initial spaces is done using undelimited-delimited argument pair trick.

pdfextra.opm

```
\_def\.DDDscripts#1{\.DDDscriptsA#1,,,\.end}
\_def\.DDDscriptsA#1#2,{\_ifx,#1\_ea\.untilend\_else
   \.isfiledefined{#1#2}\_iftrue%
     \_addto\.scriptfilespecs{\.cs{filespec:#1#2} }%
     \_addto\.names{\.cs{filename:#1#2} \.cs{filespec:#1#2} }%
   \_fi
   \_ea\.DDDscriptsA\_fi
}
```

For 3D views we need to process yet another comma separated list, this time with `\.DDDviews`. The result has to be separated by spaces and we also don't want to emit something if the specified view was invalid. Unfortunately this is expansion only context, so we can't issue a warning.

As a user convenience, before `\.DDDview` is executed, view with the name of `\.name` is tried instead of empty view array. This means that for simple 3D art with one view, one can create view with the same name as the 3D object and not have to specify anything. We also take the name only after it is redefined from optional key-value parameters – this is so we can support even the case of e.g. `screw` 3D model used twice, once with `name=screw1`, another time with `name=screw2` (with the corresponding `screw1` and `screw2` views). This is probably less useful, but…

pdfextra.opm

```
\_def\.DDDviews#1{\.DDDviewsA#1,,,\.end}
\_def\.DDDviewsA#1#2,{\_ifx,#1\_ea\.untilend\_else
   \.isdefined{3dview:#1#2}\_iftrue
      \_lastnamedcs\_space \_fi
   \_ea\.DDDviewsA\_fi
}
```

The activation/deactivation names are kind of cryptic, so we give them descriptive names. Default is explicit (de)activation. Instead of `/PV` (page visible) and `/PI` (page invisible) it would be possible to use "page open" and "page close". These are slightly different in cases when more pages are shown on screen at once, because only one page is "open", while multiple are "visible".

pdfextra.opm

```
\.sdef{activation:explicit}{/XA}
\.sdef{activation:auto}{/PV}
\.sdef{deactivation:explicit}{/XD}
\.sdef{deactivation:auto}{/PI}
```

Additional means of customization are here. `\.RMdefaults` contains the default key-value parameters. `\.RMpresentationextra` can be used to set more attributes in `/RichMediaPresentation` dictionary (although those are more specific and not generally useful).

pdfextra.opm

```
\_def\.RMdefaults{%
   name=\.name,
   activation=explicit,
   deactivation=explicit,
   toolbar=true,
   views=,
   scripts=,
}
\_def\.RMpresentationextra{}
```

For scripting using JavaScript actions one needs to access the 3D context of the 3D / Rich Media annotation. This requires the page number. We can't use `this.pageNum`, because the script strictly doesn't have to be on the same page. We use `\.pageof` (`\.setpageof` was done in `\.boxedannot`) to retrieve the page number in next run. Also PDF indexes page numbers from 0. `\DDDannot{`⟨*name*⟩`}`. and `\DDDcontext{`⟨*name*⟩`}` allow this.

pdfextra.opm

```
\_def\.DDDannot#1{%
   this.getAnnotRichMedia(\_the\_numexpr\.pageof{rm:#1}-1\_relax, '#1')%
}
\_def\.DDDcontext#1{\.DDDannot{#1}.context3D}

\_nspublic \DDDannot \DDDcontext ;
```

2.8.3 3D views

This is the interesting part about 3D art. They can have a set of predefined views – although a user may start from one, they can interactively change all the aspects by dragging with mouse or messing with the settings shown by right click menu.

There are several transformations that have to be done before it is possible to display 3D scene on a computer screen:

1. 3D transformation from the coordinate system of 3D artwork ("model") to the "world coordinate system".
2. 3D transformation from the world coordinate system to camera coordinate system.
3. projection to 2D (3D to 2D transformation).

When talking about PDF, positive x goes to the right, positive y up, and positive z "away" from us ("into the page"). This means we are working with a left handed coordinate system. In camera space, the camera sits at $(0, 0, 0)$ facing towards positive z with positive x and y going right and up respectively. Projection (one way or another) discards the z coordinate.

Although the transformations are not strictly linear, they are essentially done using multiplication by *transformation matrices*. The matrix for "model to world" (or "model") transformation is part of the 3D art file and can't be changed. However, we can make it up, because we can fully control the second transformation ("world to camera" or "view" transformation) – although we don't specify the "world to camera" matrix but rather its inverse, the "camera to world" matrix (`/C2W`). This matrix has the 4×4 form, which also allows *linear transformation* and *translation*:

$$M_{c2w} = \begin{pmatrix} a & d & g & t_x \\ b & e & h & t_y \\ c & f & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Here we use the column major convention, which is also the order how we would write the matrix to PDF file, where it is an array of 12 elements:

`/C2W [`$a\ b\ c\ d\ e\ f\ g\ h\ i\ t_x\ t_y\ t_z$`]`

In the rendering pipeline everything is transformed from world coordinates to camera space coordinates. We can think about the process also in the other way. Using M_{c2w} we specify camera's position and orientation in the world coordinate system. Due to how transformation using matrix multiplication works, the first column in the M_{c2w} matrix (vector $(a, b, c)^T$) specifies how "positive x direcetion" ("right") from camera space ends up in world coordinate system. Similiarly for $(d, e, f)^T$ being the image of positive y ("up") and $(g, h, i)^T$ being the image of positive z ("forward"). The last column, $(t_x, t_y, t_z)^T$ represents translation from camera space to world. Translation of origin (camera position) will leave it in the point with coordinates (t_x, t_y, t_z). Because of these associations with the intuitive meanings of x, y, z in camera space we also sometimes call the vectors in the first three columns of M_{c2w} "right", "up" and "forward" and the last one "eye":

$$\vec{R} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \quad \vec{U} = \begin{pmatrix} d \\ e \\ f \end{pmatrix}, \quad \vec{F} = \begin{pmatrix} g \\ h \\ i \end{pmatrix}, \quad \vec{E} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}.$$

We usually want $\vec{R}$, $\vec{U}$ and $\vec{F}$ to form an orthonormal set of vectors, i.e. all of unit length and each pair is orthogonal. The orthoganility will come from the way we calculate them, but the normality has to be ensured by normalizing the vectors after computing them, which will not be explicitly written out in the following text. $\vec{E}$ is a positional, not directional, vector and it's length will be preserved.

Now we only need a convenient way to calculate all four vectors. A wide spread method is sometimes called "look at". It essentially involves having two points: "eye" (E, position of the camera) and "target" (T, the point where the camera is pointing at). The camera position is already provided:

$$\vec{E} = E$$

From these two points alone we can easily calculate the forward vector, which corresponds to the direction of the camera:

$$\vec{F} = \overrightarrow{ET} = T - E$$

When we now imagine the point E and vector $\vec{F}$ looking towards T we can see that there is a degree of freedom – the camera can rotate about the forward vector. There is no other way than to arbitrarily choose either up or right vector. Usually we choose an arbitrary "global up" vector U_G, which will influence the general direction of the final up vector. This is because we use it to calculate the right vector:

$$\vec{R} = \vec{U}_G \times \vec{F}$$

The cross product makes it so that:

1. $\vec{R}$ is perpendicular to $\vec{F}$
2. it is also perpendicular to global up vector ($\vec{U}_G$) which we used to get rid of remaining degree of freedom.

Now that we have two orthonormal vectors (with the normalization not being explicit) we can calculate the remaining up vector:

$$\vec{U} = \vec{F} \times \vec{R}$$

The last mysterious part about the calculation are the cross products. They are of course not commutitative, so why e.g. $\vec{U}_G \times \vec{F}$ and not the other way around? This is because we have to preserve the relations these vectors had as directions of positive axes of the original camera space. There we had positive x going right, positive y up and positive z forward in a left handed coordinate system. This means that following holds (according the left hand rule):

$$\begin{aligned}\vec{R} &= \vec{U} \times \vec{F}\\ \vec{U} &= \vec{F} \times \vec{R}\\ \vec{F} &= \vec{R} \times \vec{U}\end{aligned}$$

The scheme has one flaw though. When the directions of global up vector $\vec{U}_G$ and forward vector $\vec{F}$ are linearly dependent the computed right vector will be $(0, 0, 0)$. Hence some handling of this special case is needed.

The "look at" method is essentially what is used in Alexander Grahn's package movie15[1]. Although the input aren't two points, but rather a "center of orbit" point (COO, our "target"), "center of orbit to camera vector" ($\overrightarrow{C2C}$, default is $(0, -1, 0)$) and distance of camera from the center of orbit (ROO). The value used for the arbitrary "global up" vector is $(0, 0, 1)$. When forward vector is $(0, 0, z)$, then global up is chosen to be $(0, -1, 0)$ or $(0, 1, 0)$ to handle the "0 right vector" issue.

Because the movie15 method of providing the parameters is used in essentially all packages that handle PDF 3D art (movie15, media9, rmannot, ConTEXt) we also follow the suite.

`\DDDview`[⟨*view name*⟩][⟨*key-value parameters*⟩] is the command for defining 3D views. These have to be saved into separate PDF objects anyways, using this interface we allow their reuse. If ⟨*view name*⟩ is same as ⟨*name*⟩ of `\RM` argument and no other views are specified ⟨*view name*⟩ view is automatically used (see `\RM` for details).

Key-value parameters are not optional this time, because rarely one suffices with default values – different 3D views are about customization. Handling of them is not as straightforward as before. We initially read the key-value parameters only to determine the `method` used for calculating the `/C2W` matrix. Then we reread key-value parameters again, this time with also with the default values for this particular method. Not that the general 3d views details are changed, but the methods themselves have key-value parameters of their own, and we support specifying them in this "flat" way.

Additionally we allow the different methods used to compute `/C2W` to not be expandable. Hence they are executed outside of expansion only context and are fully processed – the text they add to 3D view PDF object is temporarily stored in `\.viewparams`.

The rest is simply setting sensible defaults (or user overrides) for internal/external name of the view (`/IN` and `/XN`, one is used for scripting, one is shown by the PDF viewer), background color, rendering mode, and lighting. Cross sections and nodes are currently not supported, although users can hook in their own code using `\.DDDviewextra`, `\.DDDrendermodeextra` or `\.DDDprojectionextra`.

We have to be careful about setting rendering mode and lighting scheme, because they normally fall back to the values specified in 3D art file, which we can't access, so better not set them to anything if they are empty.

---

[1] https://www.ctan.org/pkg/movie15

pdfextra.opm

```
\_def\.DDDview[#1][#2]{\_bgroup
   \_readkv{\.DDDviewdefaults,#2}%
   \_edef\.tmp{\.DDDviewdefaults,\.cs{3dview:\_kv{method}:defaults}}%
   \_readkv{\.tmp,#2}%
   \.colortorgbdef\.bgcolor{\_kv{background}}%
   \.cs{3dview:\_kv{method}}% sets \.viewparams (/MS, /C2W, /CO)
   \_pdfobj {<</Type /3DView
     /XN (#1)
     /IN (#1)
     \.viewparams % /MS, /C2W, /CO
     /P <<
       \.cs{3dprojection:\_kv{projection}}
       \.DDDprojectionextra
     >>
     /BG <<%/Type /3DBG
       /Subtype /SC
       /C [\.bgcolor]
     >>
     \.emptyor{\_kv{rendermode}}{%
       /RM <<%/Type /3DRenderMode
         /Subtype /\.nonempty
         \.DDDrendermodeextra >> }%
     \.emptyor{\_kv{lighting}}{%
       /LS <<%/Type /3DLightingScheme
         /Subtype /\.nonempty >> }%
   >>}%
   \_pdfrefobj\_pdflastobj
   \.sxdef{3dview:#1}{\_the\_pdflastobj \_space 0 R}%
   \_egroup
}

\_nspublic \DDDview ;
```

`\.DDDviewdefaults` stores default key-value parameters for 3D views. They are mostly the PDF standard defaults or what movie15/media9 uses (for compatibility).`\.DDDviewextra`, `\.DDDrendermodeextra` or `\.DDDprojectionextra` can be used by the users to hook themself into 3D view object creation.

pdfextra.opm

```
\_def\.DDDviewdefaults{
  projection=perspective,
  scale=1,
  ps=Min,
  FOV=30,
  background=1 1 1,
  rendermode=,
  lighting=,
  method=media9,
}
\_def\.DDDprojectionextra{}
\_def\.DDDrendermodeextra{}
\_def\.DDDviewextra{}
```

There are two different projection methods:

- Orthographic: z coordinate is simply thrown away, `scale` is used for scaling the result. For technical parts where we want lines that are parallel stay parallel in the view.
- Perspective: is the way human eye sees. `FOV` can be used to set field of view. (`ps` parameter for additonal scaling to fit width/height is also available, but the default is fine for casual users).

pdfextra.opm

```
\.sdef{3dprojection:ortho}{/Subtype /O /OS \_kv{scale}}
\.sdef{3dprojection:perspective}{/Subtype /P /FOV \_kv{FOV} /PS /\_kv{ps}}
```

We offer the possibility of setting`/C2W` matrix and `/CO` (distance from camera to center of orbit) directly using `method=manual`.

pdfextra.opm

```
\.sdef{3dview:manual:defaults}{
  matrix=1 0 0 0 1 0 0 0 1 0 0 0 ,
  centeroforbit=0,
}
```

```
\.sdef{3dview:manual}{\_edef\.viewparams{
  /MS /M
  /C2W [\_kv{matrix}]
  /CO \_kv{centeroforbit}
}}
```

Another simple way of specifying camera position/orientation is to use a named setting of U3D file using a U3D path with `method=u3d`.

pdfextra.opm

```
\.sdef{3dview:u3d:defaults}{
  u3dpath=,
}
\.sdef{3dview:u3d}{%
  \_pdfunidef\.tmp{\_kv{u3dpath}}%
  \_edef\.viewparams{
  /MS /U3D
  /U3DPath \.tmp \_space
}}
```

The most advanced method of setting `/C2W` matrix and `/CO` is `method=media9`. It is thoroughly explained above, the few differences are because the input values are not two points. Also for conciseness "x", "y" and "z" are used instead of right, up and forward. The calculations are done in Lua, for simplicity.

We expect the user to supply the numbers in the form "`1 2 3`", but in Lua we need them comma separated ("1, 2, 3"). `\.luatriplet` does this. Just in case the code is somehow adapted without ensuring that x and z are orthonormal, we normalize also y after the second cross product.

pdfextra.opm

```
\_def\.luatriplet#1 #2 #3 {#1, #2, #3}

\.sdef{3dview:media9:defaults}{%
  roo=0,
  coo=0 0 0,
  c2c=0 -1 0,
}
\.sdef{3dview:media9}{\_edef\.viewparams{
  /MS /M
  /C2W [\_directlua{
  local function normalize(x, y, z)
      local len = math.sqrt(x*x + y*y + z*z)
      if len \csstring\~= 0 then return x/len, y/len, z/len else return 0, 0, 0 end
  end
  local function cross(ux, uy, uz, vx, vy, vz)
      return uy*vz - uz*vy, uz*vx - ux*vz, ux*vy - uy*vx
  end
  local function printmat(...)
      local arr = table.pack(...)
      for k, v in ipairs(arr) do
          arr[k] = string.format("\_pcent.6f", v)
      end
      tex.print(table.concat(arr, " "))
  end

  local roo = \_kv{roo}
  local coo_x, coo_y, coo_z = \_ea\.luatriplet\_expanded{\_kv{coo}}
  local c2c_x, c2c_y, c2c_z = normalize(\_ea\.luatriplet\_expanded{\_kv{c2c}})

  local eye_x, eye_y, eye_z = coo_x + c2c_x*roo, coo_y + c2c_y*roo, coo_z + c2c_z*roo

  local z_x, z_y, z_z = -c2c_x, -c2c_y, -c2c_z

  local up_x, up_y, up_z = 0, 0, 1
  if math.abs(z_x) + math.abs(z_y) < 0.0000001 then % z_x == 0 and z_y == 0
      if z_z < 0.0000001 then % z_z <= 0
          up_x, up_y, up_z = 0, 1, 0
      else
          up_x, up_y, up_z = 0, -1, 0
      end
  end

  local x_x, x_y, x_z = normalize(cross(up_x, up_y, up_z, z_x, z_y, z_z))
  local y_x, y_y, y_z = normalize(cross(z_x, z_y, z_z, x_x, x_y, x_z))

  local eye_x, eye_y, eye_z = coo_x - z_x*roo, coo_y - z_y*roo, coo_z - z_z*roo

  printmat(x_x, x_y, x_z, y_x, y_y, y_z, z_x, z_y, z_z, eye_x, eye_y, eye_z)
  }]
  /CO \_kv{roo}
}}
```

Last, but not least, is an action for setting the 3D view of a 3D/RM annotation using an action. `\.goto3dviewaction[goto3dview:`⟨*name*⟩`:`⟨*view*⟩`]`. ⟨*name*⟩ is name of the annotation which will be influenced. ⟨*view*⟩ is passed directly to PDF. Therefore it can be either an index to the view array (starting at 0) or name of view in parentheses – "`(`⟨*view name*⟩`)`".

`pdfextra.opm`

```
\.sdef{goto3dviewaction}[#1:#2:#3]{/S /GoTo3DView
   /TA \.cs{rm:#2} 0 R
   /V #3
}
```

2.9 MIME type database

This is the uninteresting MIME type database teased in section 2.3. Ideally this would only be a subset of what IANA defines at `https://www.iana.org/assignments/media-types/media-types.xhtml`. But there are additions like `model/u3d` and `model/prc`, which don't seem to be official, yet. Other "unofficial" MIME types are taken from Mozilla's "common" lists:

- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types`.
- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types`

`\.mimetype{`⟨*extension*⟩`}{`⟨*MIME type*⟩`}` is a shortcut of mapping ⟨*extension*⟩ to ⟨*MIME type*⟩.

`pdfextra.opm`

```
\_def\.mimetype#1#2{\_sdef{_pdfextra_mimetype:#1}{#2}}

\.mimetype{js}{application/javascript}
\.mimetype{pdf}{application/pdf}

\.mimetype{prc}{model/prc}
\.mimetype{u3d}{model/u3d}

\.mimetype{wav}{audio/x-wav}
\.mimetype{mp3}{audio/mpeg}
\.mimetype{opus}{audio/opus}

\.mimetype{avi}{video/x-msvideo}
\.mimetype{mp4}{video/mp4}
\.mimetype{webm}{video/webm}

\.mimetype{tex}{application/x-tex}


% more or less legacy formats
\.mimetype{au}{audio/basic}
\.mimetype{aiff}{audio/x-aiff}
\.mimetype{mov}{video/quicktime}
\.mimetype{mpg}{video/mpeg}
\.mimetype{wmv}{video/x-ms-wmv}

\_endnamespace
```