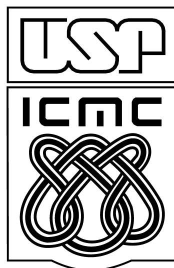
UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Movie support for PDF using LaTeX

Marco Aurélio Graciotto Silva
Ellen Francine Barbosa
José Carlos Maldonado

n. XXX

TECHNICAL REPORT



São Carlos - SP
November/2011

UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

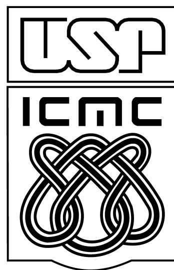
ISSN: 0103-2569

Movie support for PDF using LaTeX

Marco Aurélio Graciotto Silva
Ellen Francine Barbosa
José Carlos Maldonado

n. XXX

TECHNICAL REPORT



São Carlos - SP

November/2011

Contents

1 Introduction **1**

1.1 Text organization . . . 2

1.2 Text conventions . . . 2

2 PDF format **3**

2.1 Low-level structure . . . 3

2.2 High-level structure . . . 5

2.2.1 Header . . . 5

2.2.2 Body . . . 6

2.2.2.1 Number . . . 7

2.2.2.2 Indirect reference . . . 7

2.2.2.3 Name . . . 7

2.2.2.4 Dictionary . . . 7

2.2.2.5 Array . . . 8

2.2.2.6 String . . . 8

2.2.2.7 Stream . . . 9

2.2.2.8 Boolean . . . 10

2.2.2.9 Null object . . . 10

2.2.3 Cross-reference table . . . 10

2.2.4 Trailer . . . 10

2.3 Logical types . . . 10

2.3.1 Catalog . . . 11

2.3.2 Pages . . . 11

2.3.3 Page . . . 11

2.3.4 Font . . . 12

2.3.5 Vector graphic . . . 12

2.3.6 Raster graphics . . . 12

2.3.7 Text . . . 12

2.3.8 Encoding . . . 13

2.3.9 File attachment . . . 13

2.3.10 Movie . . . 13
2.4 Metadata . . . 15
2.5 Filters . . . 15

3 PDF and LaTeX **17**
3.1 Beamer and multimedia.sty . . . 17
3.2 Flashmovie . . . 18
3.3 Movie15 . . . 20
3.4 External command . . . 22
3.5 Concluding remarks . . . 23

4 PDF viewers **25**
4.1 Adobe Reader . . . 25
4.2 Okular . . . 26
4.3 Evince . . . 27
4.4 Concluding remarks . . . 27

5 Conclusions **29**

Chapter 1

Introduction

PDF (Portable Document Format) files are a widespread document format and thoroughly used for document sharing in the business and scientific world. The ability to efficiently store text, sound, pictures and movies in a single file, providing security features (due to its cryptography and signing capabilities) and fidelity in presentation, is key for its success.

Despite the success of PDF, some of its features are not used at its fullest. For instance, movies, supported since 2008[1], are not commonly found in PDF documents (although it is common to find documents with links to such artifacts).

Efforts to develop tools that are able to use such features are undergoing. Adobe itself develops Acrobat Professional (Adobe Systems 1993), which can insert movies in PDF files. However, Acrobat is a proprietary and closed software. Open source options are available, and should, whenever possible, be considered.

Unfortunately, there are few open source software available for viewing and creating such files. Viewing support for movies is only available and stable for Adobe Reader: open source options, like Evince (The Evince Team 2004) and Okular (Szymanski, Cid *et al.* 2005), provides crude support, failing to play most files. Even the support for such feature by Adobe Reader for Linux is flawed, as it requires Real Player One, which is not available in most Linux distributions.

A work around to this issue is to generate files for both platforms: Windows and Linux. This approach is straightforward and this technical report is built on this hypotheses. Nonetheless, efforts should be driven to address Linux support for movies (which is possible, given the open source nature of the main PDF viewers that support Linux), eventually removing the requirement to generate multiple versions (platform specific) of the same document.

The creation of multimedia PDF files for several platforms can be achieved by employing intermediate documents from which PDF documents can be compiled from. In the context of my PhD project, the LaTeX document typesetting language is used to create such documents. Some LaTeX packages allow the inclusion

[1] Acrobat supports movies and Flash animations since Acrobat Reader 9 (`http://blogs.adobe.com/adobereader/2008/06/adobe_reader_9_is_here_1.html`).

of movies: `movie15`, `flashmovie`, `multimedia`. However, the files generated using these packages are not fully compatible with Acrobat Reader (and, supposedly, with the PDF standard). Even such intermediate documents must be adapted. Fortunately, it is feasible (and pretty straightforward) to handle such situations using LaTeX.

Thus, the goal of this technical report is to describe the inherent difficulties regarding multimedia PDF support for open source (Linux) platforms, and assess the best solution for viewing and creation of cross-platform (Linux and Windows) PDF files.

1.1. Text organization

This technical report is organized as follows. Chapter 2 briefly describes PDF documents inner details related or required by movie support. Chapter 3 presents LaTeX solutions to use movies LaTeX documents. Chapter 4 builds up on the examples implemented using the techniques described in the previous chapter, and discusses the limitations and support for multimedia features of open source PDF viewers and Acrobat Reader. Finally, Chapter 5 draws some conclusions regarding the support for movies in PDF files.

1.2. Text conventions

The following conventions were adopted in this text: monospaced words represent system commands, short source code text extracts, PDF data types and LaTeX package's names.

Chapter 2

PDF format

PDF (Portable Document Format) is a file format suitable for reliable presentation of documents. It implements a subset of PostScript (but without the stateful features of the PostScript language), allows font-embedding and provides a font replacement system (thus allowing a high fidelity reproduction of documents, regardless of the system's configuration).

Since its inception, in 1993, it gained features that shifted its roots from static contents (text and figures), allowing animations, movies and even user interaction. Although interaction is usually restricted to link activation (that relies on events generated by input devices such as mouse and keyboard), it is possible to use ECMAScript within a PDF file, recovering the computability once lost after abandoning PostScript.

2.1. Low-level structure

A PDF document is a sequence of 8-bit bytes (Marchesi *et al.* 2008). Those bytes represents characters, which can be of the following types: delimiter characters, white-space characters, or regular characters.

- Delimiter character: These characters delimit syntactic entities (but are not part of the defined entity):
  - (and): begin and end of literal string;
  - < and >: begin and end of hexadecimal string;
  - << and >>: begin and end of dictionary;
  - [and]: : begin and end of array;
  - { and }: begin and end of procedure;
  - /: begin of a name;
  - %: begin of a comment .
- White-space character: The white-space characters are (followed by their byte value): null (0), tabulation mark (9), line feed (10), form feed (12), carriage return (13), space (32) and end of line (which can be a carriage return followed by a line feed, a carriage return or a line feed).

- Regular character: Any character that is not a white-space character.

Reading a PDF file is rather simple. White-space characters are ignored. Delimiter characters signal the existence of a PDF syntactic entity (literal string, hexadecimal string, array, procedure, name). If the demiliter character for a comment is found, the rest of the line is ignored. A sequence of consecutive regular characters comprises a single token.

Consider Listing 2.1. If you tokenize and analyze it, the following syntactic entities will be found: a comment (`%PDF-1.7`), three tokens (`1 0 obj`), a comment (`% entry code`), a dictionary start (`<<`), three beginnings of a name and respective tokens (that will be, at the logical level, a name), three regular tokens (`2 0 R`), a dictionary end (`>>`), and so on.

```
%PDF-1.7

1 0 obj  % entry point
<<
  /Type /Catalog
  /Pages 2 0 R
>>
endobj

2 0 obj
<<
  /Type /Pages
  /MediaBox [ 0 0 200 200 ]
  /Count 1
  /Kids [ 3 0 R ]
>>
endobj

3 0 obj
<<
  /Type /Page
  /Parent 2 0 R
  /Resources
  <<
    /Font
    <<
      /F1 4 0 R
    >>
  >>
  /Contents 5 0 R
>>
endobj

4 0 obj
<<
  /Type /Font
  /Subtype /Type1
  /BaseFont /Times-Roman
>>
endobj

5 0 obj  % page content
<<
  /Length 44
>>
stream
BT
70 50 TD
/F1 12 Tf
(Hello, world!) Tj
ET
endstream
```

```
53 endobj
54
55 xref
56 0 6
57 0000000000 65535 f
58 0000000010 00000 n
59 0000000079 00000 n
60 0000000173 00000 n
61 0000000301 00000 n
62 0000000380 00000 n
63 trailer
64 <<
65   /Size 6
66   /Root 1 0 R
67 >>
68 startxref
69 492
70 %%EOF
```

Listing 2.1: Sample PDF document (extracted from GNU PDF project). This is the famous “Hello World” (the very first example for any programming language), but implemented using PDF.)

2.2. High-level structure

A simple PDF contains 4 parts: header, body, cross-reference table, and trailer. After the trailer, the document still has three lines, as shown in Listing 2.2. The last one contains the word `%%EOF`, that marks the end of the document. Above this word, there are two lines: the first with the word `startxref` and the second with a number: this number is the offset of the cross-reference table within the input file.

```
66 startxref
67 492
68 %%EOF
```

Listing 2.2: End section the Hello World example.

Besides those parts, the file can contain comments. They start with the percent symbol (`%`) character and end at the next new line. Technically, comments are equivalent to a whitespace character, and do not include the ending new line.

2.2.1. Header

The header contains the magic word `PDF-` and the version of the PDF format used for the file. For example, for a PDF 1.7 compatible file, the header would be as defined in the Listing 2.3.

```
1 %PDF-1.7
```

Listing 2.3: Header of PDF file.

The current available (valid) magic words and the version of Acrobat that implements the given version of the PDF specification are:

- `PDF-1.0`: Acrobat 1.0 (1993).
- `PDF-1.1`: Acrobat 2.0 (1994).
- `PDF-1.2`: Acrobat 3.0 (1996).

- `PDF-1.3`: Acrobat 4.0 (1999).
- `PDF-1.4`: Acrobat 5.0 (2001).
- `PDF-1.5`: Acrobat 6.0 (2003).
- `PDF-1.6`: Acrobat 7.0 (2005).
- `PDF-1.7`: This corresponds to ISO 32000, the standardized version of PDF (specified by ISO). The first Acrobat version that supports PDF 1.7 and ISO 32000 is Acrobat 8.0 (2006). Besides that, Adobe has developed three extensions for the ISO 32000: Adobe Extension Level 3, supported by Acrobat 9.0 (2008), and Adobe Extension Level 5, supported by Acrobat 9.1 (2009).

The header can be followed by an optional line that must be specified if the PDF contains binary data[1].This line must have at least 4 commented binary characters ($ASCIIvalue \geq 128$), so that generic tools have a chance to detect it as a binary file (the best would be to use characters in the range $[128, 159]$, which are not part of ASCII nor Latin-1, and the NUL – 0 – character, since some programs consider a file as binary only if it contains the NUL character).

2.2.2. Body

The body contains a collection of objects that are used in the document. In the body (the object list), we see different kind of objects definitions, as shown in the Listing 2.4.

```
3  1 0 obj  % entry point
4  <<
5    /Type /Catalog
6    /Pages 2 0 R
7  >>
8  endobj
9
10 2 0 obj
11 <<
12   /Type /Pages
13   /MediaBox [ 0 0 200 200 ]
14   /Count 1
15   /Kids [ 3 0 R ]
16 >>
17 endobj
```

Listing 2.4: Body declaration of a PDF document.

Objects may be either direct (embedded in another object) or indirect. Indirect objects are identified by an object number (ON), which is a positive integer, and a generation number (GN), which is a non-negative integer. The definition of the object starts with the object identification and its contents. The content is prefix by the word `obj` and suffixed by `endobj`. The Listing 2.5 contains the basic structure of an object.

```
ON GN obj
  <object contents>
endobj
```

Listing 2.5: General structure of an PDF object.

Using Listing 2.4 as an example, it defines two indirect objects (`1 0` and `2 0`) and each indirect object defines direct objects (in the example, a `dictionary` object). Note that content streams (the object's definition within the words `obj` and `endobj`) may contain any object but `indirect reference` objects.

[1] A PDF file can be seen as a general text-based structure, but as it usually contain non-ASCII (binary) data, it should always be considered a binary file.

There are nine types of objects defined in the PDF specification:

- Number: An integer or floating point number.
- Indirect reference: A reference to another object.
- Name: Identifier. It usually represents a property name for an object.
- Dictionary: unordered list of tuples $< name, object >$.
- Array: Ordered list of objects.
- String: Text (either literal or hexadecimal text).
- Stream: Embedded data.
- Boolean: Logical value.
- Null object: The NULL value.

This was a brief description of the object types. The subsections below describe them in detail.

2.2.2.1. Number

A number is a scalar value of an integer or real numbers.

2.2.2.2. Indirect reference

An indirect reference is a reference to an indirect object. It is specified by referencing the object identifier, followed by the word `R`. (e.g., `5 0 R`).

If the object referenced does not exist, it is equivalent to the `Null` object.

2.2.2.3. Name

`Name` objects are made out of a sequence of regular characters which is prefixed by the symbol `/`. They are finished by a null octet (e.g., `/Pages`).

This type of object is usually used to represent a property name for an object (if you imagine an object, in the sense of object-oriented programming, as a dictionary where each tuple is the property name and its value, respectively). Although any name can be defined, the PDF Specification defines expected names for each object defined in the PDF document. For example, most objects defines a `/Type` name that represents the its type (semantically speaking, not regarding its syntactic function).

2.2.2.4. Dictionary

A `dictionary` object is an associative table containing pairs of objects: the key is a `Name` object and the value is any object (that represents the value associated to the given `Name` object).

The dictionary object definition is prefixed by `<<` and suffixed by `>>`, as presented in the Listing 2.6.

```
<<
    KEY1  VAL1
    KEY2  VAL2
     . . .
    KEYn  VALn
>>
```

Listing 2.6: General structure of a PDF dictionary declaration.

`Dictionary` objects are usually used to represent objects (as in object-oriented programming): each key is a property/field name and the value is its content. For example, a `Catalog` (semantic) object, as shown in Listing 2.7, is defined as a `dictionary` object: its type is defined by the `name` object `/Type` (which is associated to the value `/Catalog`, that also is a name); its contents are the pages that a catalog must have, which are defined in the `name` object `/Pages` (which is associated to the value `2 0 R`, that is a indirect reference).

```
3  1 0 obj  % entry point
4  <<
5    /Type /Catalog
6    /Pages 2 0 R
7  >>
8  endobj
```

Listing 2.7: Catalog declaration of a PDF document.

By convention, `/Type` and `/Subtype` entries may be present to identify the kind of information a dictionary contains. The type entry is not required except when it is stated in PDF specification and the actual type may be deduced from the context.

2.2.2.5. Array

A `array` object is a one-dimensional collection of objects arranged sequentially. It is prefixed by `[` and sufixed by `]` (e.g., `[ 0 0 200 200 ]`).

2.2.2.6. String

A `string` object is a sequence of bytes, which values ranges from 0 to 255. It can be written in two ways: sequence of literal characters or hexadecimal data.

Strings of literal characters can contain any character except unbalanced parentheses. The backslash character (\), white-space characters and parentheses (if unbalanced) must be specially handled, as defined in Section 2.2.2.6

Table 2.1: Escape codes for characters in a PDF string.

`\n`	Line Feed (LF)
`\r`	Carriage Return (CR)
`\t`	Horizontal Tab (HT)
`\b`	Backspace (BS)
`\f`	FormFeed (FF)
`\(`	Left parenthesis
`\)`	Right parenthesis
`\\`	Backslash
`\EOL`	Ignore
`\ddd`	Character code in hexadecimal. It can be used one, two or three 0-7 digits. If a digit follows, three octals digits are mandatory.

Any backslash occurrence inside a string not found on the above table should be ignored. Inner balanced parenthesis are taken literally and does not make use of backslash escaping.

Strings of hexadecimal data are composed of pairs of heximal digits, each defined by the following expression: `<[0-9A-Fa-f[WSC]]*>`. White-space characters are ignored. If there is an odd number of digits inside a hexadecimal string, the final digit is assumed to be 0.

2.2.2.7. Stream

A stream is a sequence of bytes that represents embedded data. Unlike strings, a stream can be read incrementally and can be of unlimited length. It starts with a dictionary that describes the stream (such as its length or the encoding is uses) and the sequence of bytes, which are prefixed by the word `stream` and suffixed by the word `endstream`. For instance, Listing 2.8 defines a stream with the text of the "Hello World" example (Listing 2.1).

```
42 5 0 obj  % page content
43 <<
44   /Length 44
45 >>
46 stream
47 BT
48 70 50 TD
49 /F1 12 Tf
50 (Hello, world!) Tj
51 ET
52 endstream
53 endobj
```

Listing 2.8: Example of a stream object in a PDF document.

As a stream can handle any kind of data, it requires the specification of several properties. The common entries for stream dictionaries are described at Line 53:

Table 2.2: Properties of a PDF stream object.

Name	Required	Type	Description
/Length	required	integer	Length of the unfiltered stream (in octets)
/Filter	optional	name or array of names	One or more names of filters that should be applied to the stream content in order to get the original content
/DecodeParms	optional	dictionary or array of dictionaries	One or more dictionaries containing the parameters of the decoding filters
/F	optional	string	String with the name of the external file that holds the stream contents
/FFilter	optional	name or array of names	One or more names of filters that should be applied to external files in order to get the original content
/FDecodeParms	optional	dictionary or array of dictionaries	One or more dictionaries containing the parameters of the decoding filters for external files

Besides the specification of the properties listed in Line 53, every streams must be defined as an indirect object and the stream dictionary must be a direct object. If the stream content is stored in an external file, any byte enclosed between `stream` and `endstream` is ignored.

2.2.2.8. Boolean

A `boolean` object is a logical value that is either `true` or `false`.

2.2.2.9. Null object

The `null` object has a type and a value that are different to those of any other object. There is only one possible value for this object type: the `null` keyword.

2.2.3. Cross-reference table

The cross-reference table is a sequential list of indirect object offsets (the position in bytes from the beginning of the file). The cross-reference table allows access to any given object by its number.

The table is identified by the word `xref`. The next line defines the start index number for the objects described by the table and the quantity of objects stores. For example, Listing 2.9 describes a cross-reference table that contains 6 objects offsets, counting from 0.

```
55 xref
56 0 6
57 0000000000 65535 f
58 0000000010 00000 n
59 0000000079 00000 n
60 0000000173 00000 n
61 0000000301 00000 n
62 0000000380 00000 n
```

Listing 2.9: Cross-reference table of “Hello World” example.

Each line, following the table description, contains the offset of an direct object definition, a generation number, and a marker: f (free) or n (in use).

2.2.4. Trailer

The trailer contains information about where the document starts, *i.e.*, the root object of the PDF document. This information is defined in a dictionary, which has the key `/Root` that is set as an indirect reference (usually to an object of the type `Catalog`, as defined in Listing 2.10).

```
63 trailer
64 <<
65   /Size 6
66   /Root 1 0 R
67 >>
```

Listing 2.10: Trailer of “Hello world” example.

2.3. Logical types

A PDF document is a hierarchy of objects arranged following some conventions. Most objects are defined as dictionaries. As such, it must define its data type. This is more like a logical data type (instead of the object types described in the previous section). The dictionary at the top of the hierarchy is known as the root dictionary and is called of `Catalog`.

2.3.1. Catalog

A PDF document has a root, which is defined by the `Catalog` object. This can be identified by the `/Type` name object, as defined at Listing 2.11.

```
3  1 0 obj
4  <<
5    /Type /Catalog
6    /Pages 2 0 R
7  >>
8  endobj
```

Listing 2.11: Catalog definition.

Any other object in the document shall be accessible from the root dictionary, expect for the information dictionary. For most PDF documents, the `Catalog` defines the property `/Pages`, which represents the pages of the document.

2.3.2. Pages

A `Catalog` mostly have just references to `Pages` objects (Listing 2.12). The `Pages` object is a tree-like data structure: it is a node that can reference either leaves (`Page` object) or other `Pages` nodes (which themselves can references leaves and nodes).

```
10 2 0 obj
11 <<
12   /Type /Pages
13   /MediaBox [ 0 0 200 200 ]
14   /Count 1
15   /Kids [ 3 0 R ]
16 >>
17 endobj
```

Listing 2.12: Pages definition.

In this case, the `Pages` object references only one object, as defined by the tuple `/Count, 1`, and defined by the tuple `/Kids, [3 0 R]`. which associates the key `Kids` to a `array` of `indirect references` (in this case, an array with just one element: an indirect reference with the value `3 0 R`), and the size of the medium (defined in the example as a 200x200 box).

2.3.3. Page

The `Page` object (Listing 2.13) defines a set of resources necessary to render the page. For instance, to render text, it should define a `Font` object and the text itself (`object 5`). It also defines its parent page (`/Parent 2 0 R`).

```
19 3 0 obj
20 <<
21   /Type /Page
22   /Parent 2 0 R
23   /Resources <<
24     /Font <<
25       /F1 4 0 R
26     >>
27   >>
```

```
28     /Contents 5 0 R
29  >>
30  endobj
```

Listing 2.13: Page definition.

The content of a `Page` object is defined using a language similar to PostScript. It uses postfixed operators. Further information on the supported operators is available in the PDF Specification or ISO 32000 (Adobe Systems Incorporated 2009).

2.3.4. Font

A font object in PDF is a description of a digital typeface. The font used to render a text is specified by a `Font` object. For example, a 14 base font (Times-Roman) would be defined as in Listing 2.14

```
34  4 0 obj
35  <<
36     /Type /Font
37     /Subtype /Type1
38     /BaseFont /Times-Roman
39  >>
40  endobj
```

Listing 2.14: Font definition.

A font may either describe the characteristics of a typeface, or it may include an embedded font file. The latter case is called an embedded font while the former is called an unembedded font.

The font files that may be embedded are based on widely used standard digital font formats: Type 1 (and its compressed variant CFF), TrueType, and OpenType. Additionally PDF supports the Type 3 variant in which the components of the font are described by PDF graphic operators.

2.3.5. Vector graphic

Vector graphics in PDF, as in PostScript, are constructed with paths. Paths are usually composed of lines and cubic Bézier curves, but can also be constructed from the outlines of text.

Paths can be stroked, filled, or used for clipping. Strokes and fills can use any color set in the graphics state, including patterns. Several types of patterns are supported (e.g., tiling – artwork drawn repeatedly – and shading – artwork drawn repeatedly varying colors).

2.3.6. Raster graphics

A raster image in PDF can be represented as an inline image, defined in the page description, or as a dictionary with an associated stream. The commonest form is the dictionary one. The dictionary describes properties of the image, and the stream contains the image data (which is usually filtered for compression purposes).

2.3.7. Text

Text in PDF is represented by text elements in page description. A text element specifies that characters should be drawn at certain positions. The characters are specified using the encoding of a selected font resource.

2.3.8. Encoding

Within text strings, characters are defined as bytes that map to glyphs in the current font using an encoding. There are a number of predefined encodings, including WinAnsi, MacRoman, and a large number of encodings for East Asian languages, and a font can have its own built-in encoding.

PDF files can specify a predefined encoding to use, the font's built-in encoding or provide a lookup table of differences to a predefined or built-in encoding (which is not recommended for TrueType fonts, as the encoding mechanisms in PDF is designed for Type 1 fonts, and the rules for applying them to TrueType fonts are complex).

2.3.9. File attachment

PDF files can have document-level and page-level file attachments, which the reader can access and open or save to their local filesystem.

2.3.10. Movie

The PDF file format is able to display movies, either by referencing an external file or by embedding it in the PDF file itself. PDF specifications, in its version 1.2, introduced the `Movie Annotation` object type. It consists of:

- A `Movie` dictionary with the following attributes:
  - F: File specification.
  - Aspect: Aspect ratio.
  - Rotate: Rotation of the movie (multiple of 90).
  - Poster: Thumbnail of the movie.
- An `A` object (Activation) as a `dictionary` or as a `boolean`. This object controls how the movie will be played. If defined by a boolean type object, the value of `true` will enable the default values to play the video, while the value `false` disables it.
  - Start: Starting time.
  - Duration: Duration.
  - Rate: Frame rate.
  - Volume: Sound volume.
  - ShowControls: Boolean of whether or not to show controls.
  - Mode: Play mode (once, repeat, palindrome).
  - Synchronous: Boolean of whether or not to wait for the video player.
  - FWScale: Floating window scale.
  - FWPosition: Floating window position.

Movies can also be controlled by means of a `Movie action`. Actions describe things to do when an annotation is activated, or when a page is opened, when the document is closed, etc. Contents of the `Movie action` dictionary are identical to those of a `Movie activation` dictionary. It adds the following fields:

- S: Name of the action, must be Movie.
- Annotation: Indirect reference to the `Movie annotation`.
- T: (optional) title of the movie.

- Operation: (optional) operation on the movie (play/pause/stop/resume).

PDF 1.2 has some shortcomings, such as no support for embedding movies. It also is specific for movies, and there was a demand for other multimedia objects types' support, such as animations and 3D objects. Thus, PDF 1.5 introduces a new object, `Multimedia Object`, that provides a generic framework for multimedia content. Files can also be embedded directly in the PDF file itself. Multimedia objects are classified into the following objects:

- Screen Annotation: Object that represents an area on the rendering screen linked to an action to be triggered when activated.
- Rendition Action: Object that details what operation to run on a `Rendition` object.
- Rendition Object: Object that describes the media (movie).

Finally, PDF 1.7, Extension Level 3 (thus, not an official standard, but an Adobe-specific specification) defines the `RichMedia` annotation. It defines a dictionary, whose `Subtype` is `RichMedia` and defines two dictionaries: `RichMediaSettings` and `RichMediaContent`.

`RichMediaSettings` stores the conditions and responses that occur in response to activation and deactivation events. Each of these events are defined using dictionaries:

- Activation: The dictionary type is `RichMediaActivation`. The key `Condition` defines the circumstances under which the media should be activated (`XA` if explicitly activated by an user action or script, `PO` if activated when the page that contains the media receives focus; or `PV`, if activated when the media becomes visible).
  The `Animation` key controls the method the media is played (speed and repetition). This feature is highly implementation dependent (*i.e.*, it is not assured the parameters will work the same way in every PDF reader). It is defined by a dictionary. Its type is `RichMediaAnimation`. The key `PlayCount` defines how many times the media will be played (and a negative number will cause the media to be played infinitely in loop until it is deactivated). The key `Speed` defines the speed the media will be played (if set to 1, the original speed will be used).
- Deactivation: The dictionary type is `RichMediaDeactivation`. The key `Condition` defines the circumstances under which the media should be deactivated (`XD` if explicitly deactivated by an user action or script, `PC` if deactivated when the page that contains the media loses focus; or `PI`, if deactivated when the media is no longer visible).

`RichMediaContent` is a dictionary that defines the objects that will be used. Its main attributes are the assets and the configuration:

- Assets: It is a tree of named objects. The objects represented in the tree are the media files (Flash applications, movies, etc).
- Configuration: It is defined with a dictionary (which type is `RichMediaConfiguration`). The key `Instances` describes the media content.
  `Instances` is defined in a dictionary, which `Type` is `RichMediaInstance`. The key `Subtype` defines the type of the object (which can be `3D`, `Flash`, `Sound` or `Video`). The key `Asset` defines the actual media to be played (*i.e.*, it is a object reference to an object defined in the `Assets` tree.

This should be enough for understanding PDF structures used by current LaTeX classes to use RichMedia elements. However, it is not complete and, for a comprehensive definition of these elements, the Adobe complement to ISO 32000 should be used (**??**).

2.4. Metadata

PDF files can contain two types of metadata. The first is the Document Information Dictionary, a set of tuples key/value such as author, title, subject, creation and update dates. This is stored in the optional Info trailer of the file.

In PDF 1.4, support was added for Metadata Streams, using the Extensible Metadata Platform (XMP) to add XML standards-based extensible metadata as used in other file formats. This allows metadata to be attached to any stream in the document, such as information about embedded illustrations, as well as the whole document (attaching to the document catalog), using an extensible schema.

2.5. Filters

Streams can be represented directly, as clear text, or as binary data (usually compressed text). This is accomplished by using filters.

A filter is specified by the tuple `/Filter, <filter>`. For instance in Listing 2.15 it is used a filter to compress (`/FlateDecode`) a stream.

```
5 0 obj
<<
  /Length 52
  /Filter /FlateDecode
>>
stream
...
endstream
endobj
```

Listing 2.15: Example of use of an filter for compression purposes.

PDF implements general purpose filters (mostly lossless compression filters) and specific purpose filter (e.g., image-specific filters):

- ASCII85Decode: Convert a stream into 7-bit ASCII.
- ASCIIHexDecode: Similar to ASCII85Decode but less compact.
- FlateDecode: Compression filter based on the DEFLATE or Zip algorithm.
- LZWDecode: Compression filter based on LZW algorithm.
- RunLengthDecode: Compression filter suitable for streams with repetitive data that uses the Run-length encoding (RLE) algorithm.

- DCTDecode: Lossy filter based on the JPEG standard.
- CCITTFaxDecode: Lossless filter based on the CCITT fax compression standard.
- JBIG2Decode: Lossy or lossless filter based on the JBIG2 standard.
- JPXDecode: Lossy or lossless filter based on the JPEG 2000 standard.

If you want to study an existing PDF, the pdftk tool (Steward 2004) can come in handy: the `uncompress` command will convert all compressed streams to clear text (Listing 2.16).

```
pdftk hello-stream.pdf output hello-clear.pdf uncompress
```

Listing 2.16: Command to uncompress the streams of a PDF document.

Chapter 3

PDF and LaTeX

Although PDF files can be created by Adobe Acrobat, this technical report considers only PDF files generated by LaTeX (Lamport *et al.* 1986), more specifically by pdfTeX (Thành *et al.* 1998). The latter supports some PDF special commands, such as `pdfmark` and `pdfannot`. Those commands, a heritage from PostScript, can be used by LaTeX packages to define any kind of object for a PDF document.

Three LaTeX packages were analyzed: `multimedia.sty`, provided as part of Beamer (Tantau *et al.* 2003); `movie15` (Grahn); and `flashmovie` (Hartmann 2009). For each of them, the support for SWF (Flash animations and movies), AVI (movies encoded in MPEG-4 and stored in an AVI container) and MPEG-4 (movies encoded in MPEG-4 and archived in an MPEG-4 container) was evaluated.

3.1. Beamer and multimedia.sty

The `beamer` package supports the compilation of presentations in PDF and provides an extra package, `multimedia.sty` that enables support for the inclusion of movies. It defines a new command, `movie`, which has two arguments: the text to used when no movie is played, and the movie itself (Listing 3.1, Lines 9, 13 and 17). As for the text to be shown before the movie start, it is highly recommended to insert an image of the movie (usually the first frame, but this is a design decision).

```
1  \documentclass{article}
2
3  \usepackage{multimedia}
4  \usepackage{graphicx}
5
6  \begin{document}
7
8  \section{AVI}
9  \movie{\includegraphics{saturn5.jpg}}{saturn5.avi}
10 \clearpage
11
12 \section{MPEG}
```

```
13 \movie{\includegraphics{saturn5.jpg}}{saturn5.mp4}
14 \clearpage
15
16 \section{SWF}
17 \movie{\includegraphics{saturn5.jpg}}{saturn5.swf}
18 \clearpage
19
20 \end{document}
```

Listing 3.1: Example of movies in PDF documents using `multimedia.sty`.

Besides these parameters, the `movie` command have several options. Three are rather important (information regarding other options can be read in the package documentation):

- autostart: Automatically starts the movie when the page it belongs too is shown.
- poster: Displays the first frame of the movie if it is not playing. However, this did not work for our example, thus we opted out for not using the `poster` option. Instead we used the `includegraphics` statement as the first argument of `movie` command (and achieving the same result).
- showcontrols: Displays movie control (pause, stop, play, etc).

Usage instructions aside, the implementation of the movie support by the `multimedia` package is defined as follows. The command `movie` creates a PDF annotation of the type `Movie` (Listing 3.2, Lines 2 and 4). The parameter `/F` defines the filename for the movie to be played (Listing 3.2, Line 9).

```
1  \pdfmark[{\box\@tempboxa}]{%
2      \pdfannot width \@tempdima height \@tempdimb depth \@tempdimc
3      {
4          /Subtype /Movie
5          /T (\mm@label)
6          /Border [0 0 \mm@bw]
7          /Movie
8          <<
9              /F (#3) \mm@poster\space
10         >>
11         /A
12         <<
13             \mm@start\space \mm@duration\space \mm@playmode\space \mm@controls\space
14         >>
15     }%
16 }%
```

Listing 3.2: Code related to movie definition in the package `multimedia.sty`.

The solution adopted by `multimedia.sty` is compatible with PDF specification 1.2. This option has the drawback that the file cannot be embedded into the PDF document. Thus, the movie must be distributed with the document.

3.2. Flashmovie

The package `flashmovie` allows the embedding of movies into PDF documents. Actually it supports only the inclusion of SWF files. The trick here is to include a video player as a SWF file, to embed the set of movies to be played into the PDF document as streams and configure the video player to use them.

The command provided by the package to insert movies is `flashmovie` (Listing 3.3, Lines 8, 12 and 16). It has just one parameter: the name of the file (AVI, MPEG or SWF) of the movie.

```
1 \documentclass{article}
```

```
2
3   \usepackage{flashmovie}
4
5   \begin{document}
6
7   \section{AVI}
8   \flashmovie[image=saturn5.jpg, engine=flv-player]{saturn5.avi}
9   \clearpage
10
11  \section{MPEG}
12  \flashmovie[image=saturn5.jpg, engine=flv-player]{saturn5.mp4}
13  \clearpage
14
15  \section{SWF}
16  \flashmovie[image=saturn5.jpg]{saturn5.swf}
17  \clearpage
18
19  \end{document}
```

Listing 3.3: Example of movies in PDF documents using `flashmovie`.

Besides the parameter with the movie filename, the following options can be set:

- image: File that should be used as first (static) image before commencing the movie.
- engine: Flash engine to be used to play the movies. If using a SWF file, no engine is required. However, for AVI and MPEG support, it is required to select one engine. The recommended one is `flv-player`.
- auto: Controls the automatic start of a movie. If set to `0` it will not automatically play the movie. If set to `1`, it will play the movie automatically. This option depends upon the engine choosen (when no engine is used, this option has no effect).

The code to embed the files (movie and video player engine) into the PDF document is shown at Listing 3.4.

```
\def\flashmovieembedfileinternal#1{
  \immediate
  \pdfobj stream
      attr {  /Type/EmbeddedFile  }
      file {#1}
  \immediate
  \pdfobj { <<
      /Type /Filespec
      /F  (#1)
      /UF (#1)
      /EF << /F \the\pdflastobj\space 0 R >>
    >>}
}
```

Listing 3.4: Code that embeds a file into a PDF file.

Embedding a file into a PDF document is sufficient for SWF-based movies, but not for AVI or MPEG ones. Thus, it is required to configure the Flash engine. The package `flashmovie` defines a `Rich Media Annotation` object, which is described in the Extension Level 3 for the ISO 32000, authored by Adobe (Listing 3.5).

```
\pdfobj{
<<
    /Activation
    <<
        /Type /RichMediaActivation
        /Condition /PV
```

```
        /Configuration \the\flashmovie@n@config\space 0 R
        /Animation
        <<
            /Subtype /Linear
            /Speed 1
            /Playcount 1
        >>
        /Presentation
        <<
            /PassContextClick false
            /Style /Embedded
            /Toolbar false
            /NavigationPane false
            /Transparent true
            /Window
            <<
                /Type /RichMediaWindow
                /Width
                <<
                    /Default 100
                    /Min 100
                    /Max 100
                >>
                /Height
                <<
                    /Default 100
                    /Min 100
                    /Max 100
                >>
                /Position
                <<
                    /Type /RichMediaPosition
                    /HAlign /Near
                    /VAlign /Near
                    /HOffset 0
                    /VOffset 0
                >>
            >>
        >>
    >>
    /Deactivation
    <<
        /Type /RichMediaDeactivation
        /Condition /PI
    >>
>>}
```

Listing 3.5: Code that configures a Rich Media Annotation object with the Flash movie player engine.

3.3. Movie15

The package `movie15` provides an interface to embed movies. It defines a new command, `includemovie`, that allows the inclusion of movies in the document. This command has three parameters: the width, the height, and the movie itself (Listing 3.6, Lines 11, 13, 19, 21, 26 and 28). The parameters related to size (width and height) can be left empty: if so, the size is the same as the one of the movie.

```
1 \documentclass{article}
2
3 \usepackage{hyperref}
4 \usepackage{movie15}
5 \usepackage{graphicx}
6
7 \begin{document}
```

```
8
9  \section{AVI}
10 \subsection{Embedded}
11 \includemovie[poster,inline=true,text={\includegraphics{saturn5}}]{}{}{saturn5.avi}
12 \subsection{External file}
13 \includemovie[poster,inline=false,text={\includegraphics{saturn5}}]{}{}{saturn5.avi}
14 \clearpage
15
16
17 \section{MPEG}
18 \subsection{Embedded}
19 \includemovie[poster,inline=true,text={\includegraphics{saturn5}}]{}{}{saturn5.mp4}
20 \subsection{External}
21 \includemovie[poster,inline=false,text={\includegraphics{saturn5}}]{}{}{saturn5.mp4}
22 \clearpage
23
24 \section{SWF}
25 \subsection{Embedded}
26 \includemovie[poster,inline=true,text={\includegraphics{saturn5}}]{}{}{saturn5.swf}
27 \subsection{External file}
28 \includemovie[poster,inline=false,text={\includegraphics{saturn5}}]{}{}{saturn5.swf}
29 \clearpage
30
31 \end{document}
```

Listing 3.6: Example of movies in PDF documents using `movie15`.

Besides such parameters, several options can be configured. The relevant ones are:

- inline: Controls whether a file will be embedded or not in a PDF file. If true, the movie in embedded in the PDF; otherwise, only the filename for the movie is used and the file itself must be available when viewing the document.
- attach: The file can be embedded as an attachment (thus, it can be easily copied from the PDF document by the user). The default value is `true`. Although it is usually a good idea to left this enabled (as the user can save the movie to file into his computer, if the player he is using does not support movies), some viewers notifies the user about attachments (and asks permission to access them), which is somewhat annoying. Thus, it is recommended to set this option to `false`.
- text: Defines the text to be used when the movie is stopped. This is usually configured with a image that represents the movie (usually the first frame).
- poster: Display the first frame of the movie when it is stopped (and no other alternative text is provided). This option is not supported by most Linux PDF viewers, so it is recommended to use the `text` option to configure the image to be displayed.
- control: Display a toolbar to control the movie (play, stop, pause, etc).
- autoplay: Automatically starts the movie if the page it belongs to is displayed.

The PDF support implemented by `movie15` is rather complete. It uses multimedia objects, which is supported by PDF 1.5 or greater. The respective source code is shown at Listing 3.7.

```
\pdfannot width #1 height #2 depth #3 {
    /Subtype /Screen
    /Border [0 0 0]
    /BS <</S/S/W 0>>
    /F 5
    /T (\@MXV@filename)%
    /Contents (Media File (\@MXV@mime))%
    /P \@MXV@pdfpageref{@MXV@annot\the\@MXV@includes.page} 0 R%
    /A \the\@MXV@adict\space 0 R%
    /AA \the\@MXV@aadict\space 0 R%
    /AP <</N<<>>/R<<>>/D<<>>>>%
```

```
}

% New media clip object of subtype 'Media Clip Data' (/S/MCD)
\pdfobj {
<<
    /D \@MXV@filespec\space 0 R
    /P
    <<
        /TF(TEMPACCESS)
    >>
    /S/MCD
    /CT (\@MXV@mime)
>>}
\pdfrefobj\pdflastobj%

% New rendition object of subtype 'Media Rendition' (/S/MR)
\pdfobj {
<<
    /C \the\pdflastobj\space 0 R
    /S/MR
    /SP
    <<
        /BE
        <<
            /O 0.0
        >>
    >>
    /P <<
        \@MXV@player
        /BE
        <<
            /F 2
            \@MXV@@repeat\@MXV@@volume\@MXV@@ctrls
            /D
            <<
                /S /F
            >>
        >>%
    >>%
>>
}
```

Listing 3.7: Code that configures a movie into a PDF using movie15.

The package `movie15` is the most comprehensive solution for PDF support for LaTeX documents. Not only it provides several options, as it also supports global LaTeX options such as `draft`. This is valuable as movies are usually large and, using this option, the compilation of LaTeX documents can be sped up considerably.

3.4. External command

An option, useful for Linux viewers (which support for newer PDF specifications is poor), is to use an external program to show the video. This can be accomplished using a special LaTeX command that allows the execution of external programs from within a PDF document. Fortunately, Linux has several video players and supports most video encoding and containers.

The operation is straightforward – the external command must play the selected movie – except for one detail: the movie must be played in a given position of the screen. This requires several data: dimensions of the movie, dimensions of the screen, and placement of the movie in the screen (as explained at `http://greekbitches.blogspot.com/2008/07/embedded-movies-in-latex-pdf-workaround.html`).

The statement `href` is used to run an external application within a LaTeX document. The URL specified as argument for the `href` must use the `run` protocol, as exemplified at Section 3.4:

```
\href{run:movie-script.sh}{Click here}
```

As several parameters must be set to correctly play the movie, it is often required to use a script (thus the `movie-script.sh` in the example) that calls the application in charge of playing the movie and sets the required parameters. For instance, a script could be implemented as of Line 1.

```
mplayer -geometry 30%:100% -nofs -xy 2 -loop 0 movie.mpg
```

Normally, it is necessary to fiddle with the offset and width settings in the script until the movie window is exactly in place of the still image. Note that the settings change with screen resolution, so set your screen to the resolution of the projector you are going to use.

Optionally, it is possible to run an external program, while compiling the PDF document using LaTeX, to gather video dimensions, and automatically create the script. The script below, written in Python, can be used to accomplish the script defined in Line 1. Two options must be set in the script: the screen dimension (second line) and the PDF filename (third line).

```
#!/usr/bin/python
screen=[1024.0,768.0]
f=open("filename.pdf", "r")
read=f.read()
mb=read.find("/MediaBox") #look for page bounding box "[x,y]"
mb1=read.find("[", mb)
mb2=read.find("]", mb)
pgbb=read[mb1+1:mb2].split(" ")
#reverse search for movie placeholder image bounding box
a=read[::-1].find("moviename.sh"[::-1])
b=read[::-1].find("]",a)
c=read[::-1].find("[",a)
imgbb=read[::-1][b+1:c][::-1] .split(" ")
x0=float(imgbb[0])*screen[0]/float(pgbb[2])
xwidth=(float(imgbb[2])-float(imgbb[0]))*screen[0]/float(pgbb[2])
y0=screen[1]-float(imgbb[1])*screen[1]/float(pgbb[3]) # TODO: include black border!
shell=open("moviename.sh", "w")
shell.write("#!/bin/bash" + "\n"+
    "mplayer -geometry %(x)d:%(y)d -vf scale -xy %(xwidth)d -loop 0 moviename.avi"
    % "x":x0, "y":y0, "xwidth":xwidth})
shell.close()
```

3.5. Concluding remarks

Section 3.5 summarizes the writers analyzed considering the capabilities of inclusion of movies in different formats (AVI, MPEG, and SWF) as external files or embedded.

Table 3.1: Packages capabilities comparison (for embedded files and external files support).

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	N	Y	N	Y	N	Y
flashmovie	Y	N	Y	N	Y	N
movie15	Y	Y	Y	Y	Y	Y

The `multimedia` and `flashmovie` packages complements each other: the former supports only external files; the latter supports only embedded files.

The package `movie15` has a clear edge in the comparison, supporting all the required movie formats, either embedded or using external files. From our standpoint, it is the best choice. However, it should be considered the viewer capabilities regarding the generated files (which is described at Chapter 4).

The option to use an external application to play a movie has not been considered as it highly depends on the configuration of the machine to be use to view the PDF, which is unknown before hand.

Chapter 4

PDF viewers

The main PDF viewers for Linux are: Okular, Evince and Adobe Reader. For Windows and MacOS X, Adobe Reader is the best option and will be used as a benchmark for the purpose of evaluating movie support in PDF files.

The method adopted is rather simple: the PDF documents, used as examples in Chapter 3, are opened using each program and their capability to play each movie is assessed.

4.1. Adobe Reader

The standard for PDF viewers is the Adobe Reader. As Adobe was the creator of the PDF format, their software drives the evolution of the standard. Even the ISO 32000 follows the directives of the Adobe PDF Reference 6th edition, so it can be assumed that Adobe Reader is the reference implementation of the PDF standard.

Table 4.1: Adobe Reader for Linux movie-playing capabilities.

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	-	N	-	N	-	N
flashmovie	N	-	N	-	Y	-
movie15	N	N	N	N	N	N

Adobe Reader performance at Linux (Section 4.1) was disappointing: only SWF using `flashmovie` was supported.

Table 4.2: Adobe Reader for Windows movie-playing capabilities.

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	-	Y	-	N	-	N
flashmovie	N	-	Y	-	Y	-
movie15	Y	Y	Y	Y	Y	Y

Adobe Reader performance at Windows for files generated using `multimedia` and `flashmovie` was not good either: the former could play AVI files (and nothing else) while the latter could play everything but AVI files. The real winner was `movie15`-based documents: all movies could be played with it.

Table 4.3: Adobe Reader for MacOS X movie-playing capabilities.

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	-	Y	-	N	-	N
flashmovie	N	-	Y	-	Y	-
movie15	Y	Y	Y	Y	Y	Y

Adobe Reader performance at MacOS had similar results as its Windows counterpart. The only difference was that movies encoded with h.264 (MPEG-4 Part 10) and stored withing an AVI container were not supported on a recently installed MacOX 10.6.4. This is odd, as the same movie, but stored in a MPEG container, played correctly. Nonetheless, after updating the system, the video played correctly, so that we considered that MacOS X supported such AVI movies.

4.2. Okular

It is possible, although not easy, to play videos using Linux native and open source PDF viewers. The library used by most viewer, Poppler (Høgsberg, Cid *et al.* 2005), recently (2010) added support for movies. Following this move, some players are enabling movie support. The most advanced (actually, the only one currently working) is Okular (Szymanski, Cid *et al.* 2005), and the development version one.

To build an Okular version that can play movies, you should install the development version of Poppler and Okular. To install Poppler, the required steps are described at Listing 4.1

```
$ git clone git://git.freedesktop.org/git/poppler/poppler
$ cd poppler
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
```

Listing 4.1: Commands required to download, compile and install poppler.

After the `cmake` command, check if every dependency has been satisfied. A shortcut to install the required libraries is to run `apt-get build-dep poppler`. Even then, check if any other library is required.

After building and installing Poppler, you can proceed to Okular installation. The same recommendations regarding Poppler are valid here (i.e., `apt-get build-dep okular`). The required commands to download and build the development version of Okular are specified in Listing 4.2.

```
$ svn co svn://anonsvn.kde.org/home/kde/trunk/KDE/kdegraphics -N
$ cd kdegraphics
$ svn up cmake
```

```
$ svn up libs
$ svn up okular
$ mkdir build
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=`kde4-config --prefix` ..
$ cd okular
$ make
$ sudo make install
```

Listing 4.2: Commands required to download, compile and install okular.

Although Okular can play videos (despite the aforementioned drawbacks), it has some limitations regarding SWF files. This is due to the fact that SWF actually is a Flash program, not limited to video. So, while supporting videos (AVI and MPEG), Okular does not support SWF files.

Table 4.4: Okular movie-playing capabilities.

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	-	Y	-	Y	-	N
flashmovie	N	-	N	-	N	-
movie15	N	N	N	N	N	N

4.3. Evince

Evince is the default PDF viewer for the Gnome environment. Although it uses the same library as Okular to handle PDF documents (poppler), it does not supports movies.

Table 4.5: Evince movie-playing capabilities.

Package	Emb.AVI	Ext.AVI	Emb.MPEG	Ext.MPEG	Emb.SWF	Ext.SWF
multimedia	-	N	-	N	-	N
flashmovie	N	-	N	-	N	-
movie15	N	N	N	N	N	N

4.4. Concluding remarks

PDF support for movies is lagging behind at Linux. Although efforts to implement it date from 2008 (`http://ten0k.free.fr/pdf/`), there is no capable viewer released as stable. Currently, the best choice is using Okular.

However, while testing the viewers using a set of PDF with movies, it was observed that files that played correctly using Acrobat Reader could not be handled by Okular, in the Linux operational system, and that some movies could be played by Okular, but not by Adobe Reader.

Another interesting result is regarding the software used to generate the PDF. Using always the same movie, a PDF file generated using one writer could be read by a viewer but not by other. For example, a MPEG within a PDF file created using `multimedia.sty` played perfectly with Okular, but not using `movie15`. The explanation for this is simple: while Okular supports movies, it does so by the mechanisms of PDF 1.2 (used by `multimedia.sty`), not by multimedia objects (used by `movie15`).

Another issue is the video container and compression method. The PDF specification recommends the use of AVI, MOV or MPEG container, but does not specify the compression method (codec) used for the video and audio streams stored within them. The supported codecs depends upon the player and the

available media player support of the operational system. For MacOS X, it can assumed that any video that Quicktime supports can be used, but for Windows the freedom of choice is restricted (and for Linux, even more). So it cannot be assured the portability of PDF files with movies, unless previously known the supported containers and codecs of the system (which is unknown before hand).

Considering portability and Windows and Linux support, the best choices are:

- If using AVI files and `multimedia.sty`, the PDF file is supported on Linux, using Okular, and Windows (using Adobe Reader). However, the movie file must be distributed along the PDF file, what can be an issue.
- If using SWF files, `flashmovie` works fine on Adobe Reader (for Windows, Linux, and MacOS).
- If the PDF is aimed at Windows and MacOS users, `movie15` is the best solution, as it supports any kind of video.

As for the video itself, the recommended container is MPEG. Although AVI is popular, its support in major PDF readers is worser than MPEG support. Regarding the video and audio codecs, MPEG-4 Part 10 (h.264) and MPEG-1 Layer 3 are a good choice (and usually support by most operational systems and PDF readers).

Chapter 5

Conclusions

Current LaTeX packages were conceived prior to 2008, when `RichMedia Annotation` was not a supported object type for PDF. Thus, the output of multimedia content using LaTeX is outdated and not supported by newer players and even deprecated (if using ISO 32000, the PDF 1.7 specification). An evidence of this is the package `movie15`, which uses newer PDF features, and does generate code that runs perfectly on Adobe Reader for Windows.

On the other hand, open source PDF viewers do not implement the complete PDF specification. Actually, they are driven by user's need, which is basically text display support. Just recently the need for annotation has arisen as an important feature. As movies within a PDF file are not a standard, such support does not receives sufficient developer's attention. Just recently the PDF library, `poppler`, gained support for such objects, and Okular is the only viewer that makes (some) use of them.

Given this scenario, the recommendations are:

- If the PDF document is aimed at Linux, and open-source viewers are preferred to closed-source ondes, the package `multimedia.sty` is the recommended option.
- If the PDF document is aimed at both Linux, Windows, and MacOS, and the Adobe Reader is available, the package `flashmovie` is recommended.
- If the PDF document is aimed at Windows or MacOS users, the package `movie15` is recommended.

In the context of the project this technical report is being written to, the LOD approach is used to create presentations using LaTeX. As such presentations are automatically generated for a given formal model, it could be compiled several versions of the presentation automatically for a given user. The default setting could be detected from the running system (e.g., Web user agent) or specified by the user and pre-compiled documents, for the commons configurations, could be readily available. Thus, all recommendations could be implemented, achieving the best possible user experience.

References

Adobe Systems 1993 Adobe Systems. *Adobe Acrobat*. jun. 1993. Disponível em: http://www.adobe.com/products/acrobat.

Adobe Systems Incorporated 2009 Adobe Systems Incorporated. *Adobe Supplement to the ISO 32000, Base Version 1.7, Extension Level 3*. jun. 2009. Specification. Disponível em: http://www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/adobe_supplement_iso32000.pdf.

Grahn GRAHN, Alexander. *movie15*. [s.d.]. Disponível em: http://www.ctan.org/tex-archive/macros/latex/contrib/movie15/.

Hartmann 2009 HARTMANN, Timo. *flashmovie*. dez. 2009. Disponível em: http://tug.ctan.org/tex-archive/macros/latex/contrib/flashmovie/.

Høgsberg, Cid *et al.* 2005 HøGSBERG, Kristian; CID, Albert Astals *et al. Poppler*. 2005. Disponível em: http://poppler.freedesktop.org/.

Lamport *et al.* 1986 LAMPORT, Leslie *et al. LaTeX*. 1986. Disponível em: http://www.latex-project.org.

Marchesi *et al.* 2008 MARCHESI, Jose E. *et al. GNU PDF*. 2008. Programa de computador. Disponível em: http://www.gnupdf.org/.

Steward 2004 STEWARD, Sid. *PDF Toolkit (pdftk)*. 2004. Programa de computador. Disponível em: http://www.accesspdf.com/pdftk/.

Szymanski, Cid *et al.* 2005 SZYMANSKI, Piotr; CID, Albert Astals *et al. Okular*. 2005. Disponível em: http://okular.kde.org/.

Tantau *et al.* 2003 TANTAU, Till *et al. LaTeX Beamer*. 2003. Disponível em: http://bitbucket.org/rivanvx/beamer/wiki/Home.

The Evince Team 2004 The Evince Team. *Evince*. dez. 2004. Disponível em: http://projects.gnome.org/evince/.

Thành *et al.* 1998 THàNH, Hàn Thé *et al. pdfTeX*. 1998. Disponível em: http://tug.org/applications/pdftex/.