

Rendering Multimedia Clips via the Adobe PDF Library

By

Ernest R. Corl

Edited by

Andrew C. Green

August 2006

Rendering Multimedia Clips via the Adobe PDF Library

Introduction

With the release of PDF version 1.5, Adobe has greatly enhanced the PDF document viewing experience by including support for embedded multimedia movie clips. *Adobe Acrobat 7* and *Adobe Reader* support controlling and viewing many different media types. Both applications use newly-added multimedia-support features of the *Adobe PDF Library* to implement the rendering of multimedia clips.

In this paper, we will identify the set of objects in a PDF document that are used to render a multimedia clip. We will describe their purpose, their interrelation, and how to use them. Through the use of “C” language code examples, we will describe how you can use the *Adobe PDF Library* to extract a multimedia stream, and play the movie clip in your PDF viewer application.

We assume you are familiar with the Adobe PDF Reference manual, and the *Adobe Acrobat* and PDF Library API Reference. Note that we provide a description and code examples that narrowly focus on the task of PDF file data extraction. The broader exercise of creating a suitable, platform-specific viewing site is left to the reader. In order to clearly express the concepts surrounding the topic, we deliberately avoid discussions of content validation and error handling.

PDF Document Structure

A PDF file is organized as a hierarchical collection of data objects, which contain all the content and structure information that comprise a PDF document.

Page Objects

The basic recurring object in a PDF file is the *Page Object*. You can think of the collection of Page Objects in a document as being organized in a tree structure. Figure 1 depicts the logical structure of *Page Objects* and their sub-components:

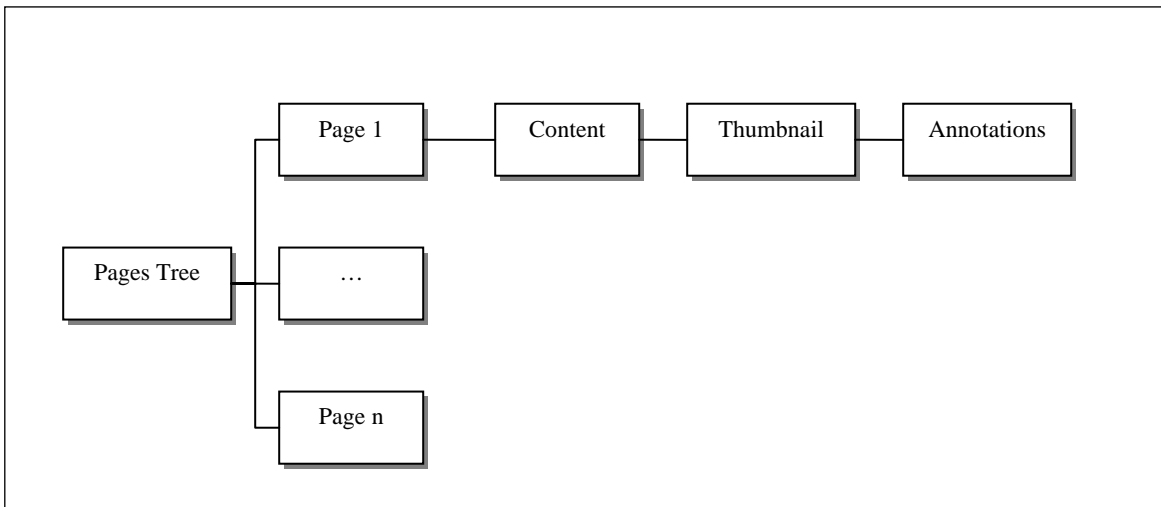


Fig. 1 Structure of a PDF document Page Tree

A Page Object either directly contains or indirectly references all the content and attributes needed to render the page (*i.e.* make it visible) or to otherwise process it. Each Page Object must contain or reference the physical dimensions of the page, the page's contents, and any additional resources such as fonts. Page Objects may also reference a thumbnail and an array of *Annotation Objects*.

Annotation Objects

Annotations provide the way for interactive components to be placed on a PDF page, and provide the means for the user to control the interaction. There are numerous types of Annotation Objects defined in the PDF reference, and the *Screen* annotation is used to enable display of multimedia clips.

Figure 2 is an abridged view of a PDF file. It includes only the objects associated with storing and rendering Screen Annotations. We will refer to this figure in the following section as we describe the purpose of the PDF structures that constitute a *Multimedia Annotation*.

```

%PDF-1.6
37 0 obj
<<
  /Type/Catalog
  /Pages 5 0 R
>>
endobj

5 0 obj
<<
  /Kids [38 0 R]
  /Count 1
>>
endobj

38 0 obj
<<
  /Type/Page
  /CropBox [0 0 612 792]
  /Annots 39 0 R
>>
endobj

39 0 obj
<<
  [40 0 R]
>>
endobj

40 0 obj
<<
  /Type /Annot
  /Subtype /Screen
  /A 55 0 R
  /AP<</N 41 0 R>>
  /Rect [179 591 558 752]
>>
endobj

41 0 obj
<<
  /Fm1 42 0 R
  /BBox [0 0 379 161]
>>
endobj

42 0 obj
<<
  /Matrix [1 0 0 1 -254
    -114]
>>
endobj

55 0 obj
<<
  /OP 0
  /R 56 0 R
  /S /Rendition
>>
endobj

56 0 obj
<<
  /C 57 0 R
  /S /MR
>>
endobj

57 0 obj
<<
  /D 58 0 R
  /CT (application/x-
    shockwave-flash)
  /P<</TF (TEMPACCESS)>>
  /S/MCD
>>
endobj

58 0 obj
<<
  /EF<</F 2 0 R>>
  /Type/Filespec
>>
endobj

2 0 obj
<<
  /Subtype/application#2Fx-
    shockwave-flash
  /Length 196100
  /DL 254018
>>
Stream ... Endstream
endobj

Trailer
<<
  /Root 37 0 R
>>
%%EOF

```

Fig. 2 Structure of a PDF file

PDF Basics

First, let's review some PDF document syntax. In Figure 2, we see a set of PDF objects, each beginning with "obj" and ending with "endobj", plus some additional header and footer material.

The "%PDF-1.6" at the beginning identifies the file as Portable Document Format, and that it adheres to the v1.6 (*Acrobat v7.0*) specification. This is the required first line of the file; individual object definitions follow it.

Each object begins with an *Object Number*, followed by (usually) a zero. That first number is used as the object's identifier, a positive integer which can be thought of as a sequence number (though their numbering is arbitrary and may be in any order). The zero following it is its *Generation Number*, which can be thought of as a revision counter; zero denotes an original. The combination of Object Number and Generation Number uniquely identifies that object within the document. Note that it isn't necessary for objects to appear in sequential order.

In an object, names begin with a forward slash ("/"), and strings of bytes are enclosed in parentheses ("(" and ")"). Numbers can be integers or rational. Objects reference other objects by their numbers (Object and Generation), followed by the letter "R" to indicate a reference.

There are two compound objects: *Arrays* and *Dictionaries*. Arrays are lists of objects and are enclosed in square brackets ("[" and "]"). Dictionaries are sequences of key/value pairs, and are denoted by double angle brackets or guillemets ("<<" and ">>"). Dictionaries can be nested to any arbitrary depth.

Components of a Multimedia Annotation

As we see in Figure 2, a PDF page annotation can be described as a tree of component object nodes. Each component object contains a dictionary of keys and key values, along with references to its siblings and dependents. In this section, we will describe the meaning and purpose of the most relevant dictionary values, and we will describe the referencing scheme that links the objects together so that the tree can be traversed programatically.

Catalog Objects

In Figure 2, the *Trailer Object* near the end of the file points back to the *Root Object* (i.e. “/Root 37 0 R” points to “37 0 obj”). The dictionary of the Root identifies it as /Type/Catalog, the *Catalog Object*, and also contains a reference to the *Page Tree Object* (“/Pages 5 0 R”). In the page tree dictionary, we find an array containing a reference to a single *Page Object* (/Kids [38 0 R]). Finally, at object 38, our page definition begins.

Page Objects

The Page Object dictionary has three entries: a Type entry (“/Type/Page”), which identifies it as a page; a crop box definition (“/CropBox [0 0 612 792]”), which denotes the page’s physical dimensions in user space coordinates; and a reference to an array of *Annotation Objects* (“Annots 39 0 R”). The Annots object array has a single entry, a reference to object 40 (“[40 0 R]”).

Multimedia Annotations

The `/Subtype` entry in the Annotation Object's dictionary is `/Screen`, the keyword that identifies a multimedia annotation. This object has three dictionary entries of interest:

- `/AP` – The *Appearance Stream* (“41 0 R”);
- `/Rect` – A rectangle describing the location where the multimedia clip is to be played (“[179 591 558 752]”), and
- `/A` – A reference to the *Action Object* (“55 0 R”).

Action Dictionary

An *Action Dictionary* specifies an action to be performed when the annotation is activated, and contains references to objects that define the particular characteristics of the action. In Figure 2, we can see in object 55 that the *Action Type*, specified by the `/S` key, is `/Rendition`.

Action Dictionaries that are Rendition subtypes control the playing of multimedia content, using the method hinted at by the Action command. This takes one of two forms, a JavaScript or one of five discrete operations, and is denoted by either a `/JS` entry or an `/OP` entry (JavaScript or Operation respectively). One or the other is required:

- If a `/JS` entry is present, it references a JavaScript to be executed when the action is triggered.
- If an `/OP` entry is present instead, it will contain a value from 0 to 4, defining one of five operations to perform when the action is triggered. (See Table 8.60 in the PDF Specification, “Additional entries specific to a rendition action,” for full details.)

An `/R` entry is required when `/OP` is present, providing a reference to the *Rendition Object* (“56 0 R”). In this example, the `/OP` value of zero (in object 55) is a command to play the rendition specified by `/R`.

Media Rendition objects

A *Media Rendition Object* (object 56 of our example in Figure 2) is a basic media object that specifies what to play, how to play it, and where to play it. The */C* entry in the object 56 dictionary is a reference to the *Media Clip Object*.

Media Clip objects

In the *Media Clip Object* (object 57 of our example) we find three important pieces of information:

- The */D* key is a reference to the *Data Stream Object*.
- The */CT* entry identifies the type of data in the data stream.
- The */P* entry is a dictionary containing the permissions needed to write a temporary file, in order to play a media clip.

The format of the */CT* string must conform to the content type specification described in Internet RFC 2045, Multipurpose Internet Mail Extensions (MIME). Our sample's content type is a Shockwave Flash movie clip.

The permissions value ("TEMPACCESS") indicates that we should create a temporary file for the media clip only if the document permissions allow content extraction. A document's permission setting is stored in the *Encryption Dictionary*, located in the Trailer Object. Documents with restricted access have encrypted content, and access permission flags are set in the Encryption Dictionary. (How to access the content of an encrypted document is beyond the scope of this discussion.) In our example, the absence of an Encryption Dictionary in the Trailer Object indicates there are no access restrictions on its content.

File Specification dictionary

The *File Specification* (*Filespec*) dictionary (object 58 of our example) may describe a platform-specific *File System File Stream* (*/FS*) or a simple *Embedded File Stream* (*/F*). (See Table 3.40 in the PDF Specification, "Entries in a file specification

dictionary,” for full details.) In this example, we are referencing an Embedded File Stream (“/F 2 0 R”).

Embedded File Streams

The Embedded File Stream dictionary (object 2 of our example) specifies the data subtype (`/Subtype`), the length of the encoded stream (`/Length`), and the length of the decoded stream (`/DL`). The encoded data (not shown here) is located between the `Stream` and `EndStream` keywords of this object.

In this section we detailed the elements of a PDF file that make up a multimedia annotation, and explained how they are interrelated. Next, we will describe how to use these pieces of information to render a multimedia clip.

Media Player Algorithms

In the previous section, we described the objects in a PDF file that comprise a multimedia annotation. Using our understanding of those objects, we can now describe, at a high level, a method to access the data stream and play the embedded multimedia clip:

1. For each page in the PDF document, fetch the Annotations array object.
2. For each annotation in the Annotations array where the subtype is `Screen`, do the following:
 - a. Fetch the rectangle that defines the rendition site coordinates. Convert the values to device space coordinates and save it.
 - b. Get the Action Dictionary object.
 1. If the Action Dictionary has a `/JS` entry, execute the JavaScript script referenced by `/JS`.
 2. Else (for an `/OP` entry)
 1. Fetch the Rendition Object referenced by `/R` and
 2. Use the value of `/OP` as the action to execute when the clip is triggered.
 3. Use the `/C` entry in the Media Rendition Object to fetch the Media Clip Dictionary.
 4. Fetch the MIME type from the `/CT` entry in the Media Clip Dictionary. Determine if a media player to render the MIME type is available.

5. Compare the permission settings in /TF with the encryption settings in the Trailer object. If permissions are compatible create a temporary file to hold the media clip data.
6. Fetch the File Stream Specification object using the value of the /D entry in the Media Clip Dictionary.
7. Determine the platform-specific file type of the data stream from the /EF entry. Fetch the Embedded Stream object.
8. In the Embedded Stream object Dictionary, get the length of the uncompressed data stream from the /DL entry. Read that number of bytes from the Stream and write them to the temporary file.
9. Invoke the media player, and pass it the coordinates of the media clip site and an appropriate handle to the temporary file containing the clip.

Adobe PDF Library API

In the previous sections, we discussed the physical data structure of a PDF file, and presented a high-level description of an algorithm to play a multimedia clip. In this section we discuss the *Adobe PDF Library* (APDFL), and the API methods we can use to access PDF data and implement our algorithm.

The API methods in the APDFL are organized in layers, with higher layers providing more abstraction away from the actual physical document structure. Two of note are the *Portable Document Layer*, or PD Layer, the highest level, providing access to bookmarks, pages, thumbnail images and annotation objects; and the *Carousel Object System Layer*, or COS Layer, providing access to the smallest granular objects in PDF, such as text strings, numbers and dictionaries. Most of the programming work we will do to display a multimedia clip will be performed using the methods and data elements of the COS Layer API. With those methods, we can read and manipulate the values in object dictionaries and streams.

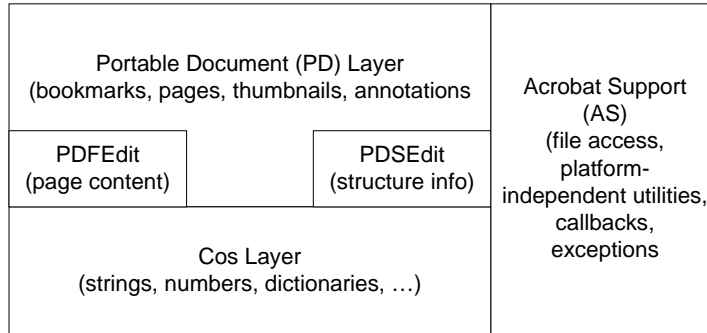


Figure 3. APDFL Core API Overview

We will need three components from the PDF to render a multimedia clip:

- a suitable media player
- the coordinates of the rendering site on the page
- the media clip data stream

In the following section, we will demonstrate how to gain access to each of these pieces. Note that for the sake of simplicity, we will not address testing for presence of optional dictionary objects, or validating the contents of a dictionary entry, nor do we perform error handling, but these should be accounted for in any production-quality application.

Our initial goal is to recognize whether a Screen Annotation is present, and if so, acquire a handle to it. Initially, it is more productive to work at the API's PD layer, as it provides a higher degree of abstraction, and thus is easier to work with.

We begin by using the API methods `PDDocOpen()` and `PDDocAcquirePage()` to open a document and get a handle to a `PDPPage` object. We determine the number of annotations on the `PDPPage` by calling `PDPPageGetNumAnnots()`.

The PD Layer API defines the structure `PDAnnot` as a container for an annotation. All the `PDAnnots` attached to a `PDPPage` are held in an array that is part of the page. To gain access to a `PDAnnot`, call `PDPPageGetAnnot()` with the index number of the one you want.

To locate Screen annotations, we compare the contents of the PDAnnot subtype member with the string "Screen". PDAnnotGetAnnotSubtype() returns an ASAtom object, which we convert to a string by passing ASAtom to ASAtomGetString(). If the returned value is "Screen," our PDAnnot is a multimedia clip.

Once we have determined that our PDAnnot is a multimedia clip, we must descend into the COS Object underworld to process it further. A call to PDAnnotGetCosObj() returns the CosObj version.

Determining the Media Type

Before we get too far, it's best to ensure that we have a media player capable of rendering the multimedia type. We can use repeated calls to CosDictGetKeyString() to traverse the CosObj tree to the Media Clip Dictionary where the media clip's data type is defined. The following code snippet demonstrates this, walking the annotation object tree to the Media Clip Object, then gets the type value that was stored with the /CT key, and compares it to the szMediaType argument. (Note that this is a simplified snippet, with no validation or error checking.)

```
bool MediaIsType(PDAnnot annot, const char* szMediaType) {
    CosObj cosobj = PDAnnotGetCosObj(annot); // The Annotations COS object.
    cosobj = CosDictGetKeyString(cosobj, "A"); // Action dictionary.
    cosobj = CosDictGetKeyString(cosobj, "R"); // Rendition dictionary.
    cosobj = CosDictGetKeyString(cosobj, "C"); // Media Clip dictionary.
    cosobj = CosDictGetKeyString(cosobj, "CT"); // Media data type.
    CosType ct = CosObjGetType(cosobj);
    ASTCount cnt;
    char *psz = CosStringValue(cosobj, &cnt);
    if(strstr(psz, szMediaType))
        return true;
    else
        return false;
}
```

Obtaining the Rendering Site Bounding Box

We will use the BBox entry from the *Normal Appearance Stream* dictionary (object 41 of our sample, pointed to by object 40) as the location of the media clip's rendition site:

```
// Traverse to the normal appearance stream and extract the bounding box
CosObj cosAnnotDict = PDAnnotGetCosObj(annot);
CosObj cosAppearanceStream = CosDictGetKeyString(cosAnnotDict, "AP");
CosObj cosNormalAppearance = CosDictGetKeyString(cosAppearanceStream, "N");
CosObj cosBBox = CosDictGetKeyString(cosNormalAppearance, "BBox");

// Convert the CosArray to ASFixedRectangle.
ASFixedRect asfBBoxRect;
asfBBoxRect.left = CosFixedValue(CosArrayGet(cosBBox, 0));
asfBBoxRect.top = CosFixedValue(CosArrayGet(cosBBox, 1));
asfBBoxRect.right = CosFixedValue(CosArrayGet(cosBBox, 2));
asfBBoxRect.bottom = CosFixedValue(CosArrayGet(cosBBox, 3));
```

The BBox entry is an array of four values: the first two give the X-axis and Y-axis coordinates of the upper left corner; the second two denote the lower right corner (the origin of this coordinate system is at the upper left corner, with positive values to the right and descending). We must convert the array to a rectangle object. Note that the BBox values are in Default User Space units. After copying them to an ASFixed Rectangle they must still be converted to Device Space coordinate units.

Creating the Media Stream Temporary File

The final piece we need is the media clip itself. The code snippet below demonstrates traversing the CosObj tree from the *Annotation Object* to the *Embedded File Stream Object*. Next, it gets a handle to the open CosStream, and writes the contents to a temporary file.

```
// Extract the SWF from the embedded stream and write it to a temp file.
// Return the path name string. Assumes the annotation is of subtype Screen.
int CreateMediaTempFile(PDAnnot annot, std::string *strPath)
{
    // Traverse the CosObj tree to the movie stream object.
    CosObj cosobj = CosNewNull();
    cosobj = PDAnnotGetCosObj(annot); // The Annotations COS object.
    cosobj = CosDictGetKeyString(cosobj, "A"); // Action dict.
    cosobj = CosDictGetKeyString(cosobj, "R"); // Rendition dict.
    cosobj = CosDictGetKeyString(cosobj, "C"); // Media Clip dict.
    cosobj = CosDictGetKeyString(cosobj, "D"); // Data Stream dict.
    cosobj = CosDictGetKeyString(cosobj, "EF"); // Embedded File Stream dict.
```

```

cosobj = CosDictGetKeyString(cosobj, "F"); // Embedded File Stream object.

// Create an ASStream and a temp file.
// Read the ASStm and write to the temp file.
ASStm asstm = CosStreamOpenStm (cosobj, cosOpenFiltered);
char* tmpFileName = _tempnam("C:\\temp", "swf");
FILE *fp = NULL;
fp = fopen(tmpFileName, "wb");
char ch;
ASTCount cnt = ASStmRead (&ch, 1, 1, asstm);
while(cnt)
{
    fwrite(&ch, 1, 1, fp);
    cnt = ASStmRead(&ch, 1, 1, asstm);
}
fflush(fp);
fclose(fp);
ASStmClose(asstm);

*strPath = tmpFileName;

return 0;
}

```

We now have the three components we need to play the multimedia clip:

- the coordinates of the display site rectangle
- the name of the media player to use
- the data stream, isolated in an external file

These components, as described above, represent the minimum requirements for rendering the multimedia content. (We are deliberately not addressing the specifics of actually rendering or playing that content; those details are determined by the selected multimedia player.)

Conclusion

Adobe has extended the PDF specification to support the notion of compound documents that are capable of providing rich multimedia content. In this paper, we provided a description of the layout of a PDF file, and the major components of the PDF Page Annotation that is used for multimedia content. We presented an algorithm for determining the presence of multimedia content in a PDF and for extracting the multimedia stream. These guidelines may be used as the basis for adding your own multimedia annotations to your output documents.

References

“PDF Reference fifth edition”, Adobe Systems, Incorporated, November 2004
<http://www.adobe.com>

“Acrobat and PDF Library API Reference”, Adobe Systems Incorporated, January 2005
<http://www.adobe.com>

Copyright © 2006 Datalogics Incorporated. All Rights Reserved. Use of Datalogics software is subject to the applicable license agreement.

Adobe, *Adobe PDF Library*, PostScript, *Adobe Acrobat* and *Adobe Reader* are trademarks of Adobe Systems Incorporated. All other trademarks are the property of their respective owners. Reproduction of this document in whole or in part is prohibited without the express written consent of Datalogics, Inc.

This document, its related materials and information are provided “as is” with no warranties expressed or implied, including, but not limited to, any implied warranty of merchantability, fitness for a particular purpose, non-infringement of intellectual property rights, or any warranty otherwise arising out of any proposal, specification or sample. Datalogics, Inc. assumes no responsibility for any errors contained in this document and has no liabilities or obligations for any damages arising from or in connection with the use of this document, including but not limited to the source code cited within.