Animations in pdfTEX-generated PDF

Jan Holeček
Faculty of Informatics, Masaryk University
Botanická 68a
602 00 Brno
Czech Republic
holecek@fi.muni.cz
http://www.fi.muni.cz/~xholecek

Petr Sojka
Faculty of Informatics, Masaryk University
Botanická 68a
602 00 Brno
Czech Republic
sojka@fi.muni.cz
http://www.fi.muni.cz/usr/sojka

Abstract

This paper presents a new approach for creating animations in Portable Document Format (PDF). The method of animation authoring described uses free software (pdfTEX) only. The animations are viewable by any viewer that supports at least some features of Acrobat JavaScript, particularly Adobe (Acrobat) Reader, which is available at no cost for a wide variety of platforms. Furthermore, the capabilities of PDF make it possible to have a single file with animations both for interactive viewing and printing.

The paper explains the principles of PDF, Acrobat JavaScript and pdfTEX needed to create animations for Adobe Reader using no other software except pdfTEX. We present a step by step explanation of animation preparation, together with sample code, using a literate programming style. Finally, we discuss other possibilities of embedding animations into documents using open standards (SVG) and free tools, and conclude with their strengths and weaknesses with respect to the method presented.

Introduction

Extensive use of electronic documents leads to new demands being made on their content. Developing specific document versions for different output devices is time consuming and costly. A very natural demand, especially when preparing educational materials, is embedding animations into a document.

A widely used open format for electronic documents is the Adobe PDF [2] format, which combines good typographic support with many interactive features. Even though it contains no programming language constructs such as those found in PostScript, the format allows for the inclusion of *Document Level JavaScript* (DLJS) [1]. Widely available PDF viewers such as Adobe Reader (formerly Acrobat Reader) benefit from this possibility, allowing interactive documents to be created.

One of the first applications showing the power of using JavaScript with PDF was Hans Hagen's calculator [5]. Further, the AcroTEX bundle [9] uses several LATEX packages and the full version of the Adobe Acrobat software for preparing PDF files with DLJS [10]; macro support for animations is rudimentary and it is stressed in the documentation that it works only with the full commercial version of Acrobat.

Our motivation is a need for PDF animations in a textbook [3] published both on paper and on CD. We have published it using Acrobat [7, 8], and eventually discovered a method to create animations using pdfTEX [11] only.

pdfTEX facilitates the PDF creation process in several ways. We can directly write the PDF code which is actually required to insert an animation. We can also utilise the TEX macro expansion power

to produce PDF code. And finally, we can write only the essential parts directly, leaving the rest to pdfTEX. pdfTEX introduces new primitives to take advantage of PDF features. The ones we are going to use will be described briefly as they appear.

In this paper, we present this new 'pdfTEX only' way of embedding animations. We require no previous knowledge either of the PDF language or of pdfTEX extensions to TEX. However, the basics of TEX macro definitions and JavaScript are assumed.

The structure of the paper is as follows. In the next section we start with the description of the PDF internal document structure with respect to animations. The core of the paper consists of commented code for the pdfTEX that generates a simple all-in-one animation. The examples are written in plain TEX [6], so that others can use it in elaborate macro packages, in a literate programming style. In the second example the animation is taken from an external file, allowing the modification of the animation without modifying the primary document. Finally, we compare this approach with the possibilities of other formats, including the new standard for Scalable Vector Graphics (SVG) [12] from the W3C.

The PDF Document Structure

A PDF file typically consists of a header, a body, a cross-reference table and a trailer. The body is the main part of the PDF document. The other parts provide meta-information and will not be discussed here. A PDF document is actually a graph of interconnected objects, each being of a certain type. There are basic data types (boolean, numeric, string) and some special and compound types which require some explanation.

A *name* object has the form /MYNAME. There is a set of names with predefined meanings when used as a dictionary key or value. Other names can be defined by the user as human readable references to indirect objects (dictionaries and indirect objects are treated below). An *array* object is a one-dimensional list, enclosed by square brackets, of objects not necessarily of the same type. A *dictionary* object is a hash, i.e., a set of key-value pairs where the keys are name objects and the values are arbitrary objects. A dictionary is enclosed by the << and >> delimiters. *Stream* objects are used to insert binary data into a PDF document. There is also a special *null* object used as an "undefined" value.

The body of a PDF file consists of a sequence of labelled objects called *indirect objects*. An object of any other type which is given a unique *object identifier* can form an indirect object. When an object is required in some place (an array element, a value of a key in a dictionary), it can be given explicitly (a direct reference) or as an object identifier to an indirect object (an *indirect reference*). In this way objects are interconnected to form a graph. An indirect reference consists of two numbers. The first number is a unique object number. The second is an object version number and is always 0 in indirect objects newly created by pdfTEX — the first one therefore suffices to restore an indirect reference.

Various document elements are typically represented by dictionary objects. Each element has a given set of required and optional keys for its dictionary. For example, the document itself is represented by a Catalog dictionary, the root node of the graph. Its key-value pairs define the overall properties of the document. A brief description of concrete objects will be given when encountered for the first time. See [2] for more detailed information.

Insertion of the Animation Frames

We are not interested in constructing the animation frames themselves — any graphics program as METAPOST will do. Let us hence assume we have a PDF file, each page of which forms a single animation frame and the frames are in the order of appearance.

Every image is inserted into PDF as a so-called *form XObject* which is actually an indirect stream object. There are three primitives that deal with images in pdfTEX. The `\pdfximage` creates an indirect object for a given image. The image can be specified as a page of another PDF file. However, the indirect object is actually inserted only if referred to by the `\pdfrefximage` primitive or preceded by `\immediate`. `\pdfrefximage` takes an object number (the first number of indirect reference) as its argument and adds the image to the TEX list being currently built. The object number of the image most recently inserted by `\pdfximage` is stored in the `\pdflastximage` register.

A general PDF indirect object can be created similarly by `\pdfobj`, `\pdfrefobj` and `\pdflastobj`. `\pdfobj` takes the object content as its argument. TEX macro expansion can be used for generating PDF code in an ordinary manner.

In our example, we first define four macros for efficiency. The `\ximage` macro creates a form XObject for a given animation frame (as an image) and saves its object number under a given key. The `\insertobj` macro creates a general PDF object and saves its object number under a given key. The `\oref` macro expands to an indirect reference of an object given by the argument. The last "`R`" is an operator that creates the actual indirect reference from

two numbers. We are not going to use \pdfref* primitives, so \immediate must be present. References will be put directly into the PDF code by the \oref macro. The \image macro actually places an image given by its key onto the page.

```
% an image for further use
\def\ximage#1#2{%
  \immediate\pdfximage
     page #2 {frames-in.pdf}%
  \expandafter\edef
     \csname pdf:#1\endcsname
     {\the\pdflastximage}}

% a general object for further use
\def\insertobj#1#2{%
  \immediate\pdfobj{#2}%
  \expandafter\edef
     \csname pdf:#1\endcsname
     {\the\pdflastobj}}

% expands to an indirect ref. for a key
\def\oref#1{%
  \csname pdf:#1\endcsname\space 0 R}

% actually places an image
\def\image#1{%
  \expandafter\pdfrefximage
     \csname pdf:#1\endcsname}
```

Another new primitive introduced by pdfTEX is \pdfcatalog. Its argument is added to the document's Catalog dictionary every time it is expanded. The one below makes the document open at the first page and the viewer fit the page into the window. One more key will be described below.

```
% set up the document
\pdfcatalog{/OpenAction [ 0 /Fit ]}
```

Now we are going to insert animation frames into the document. We will use the \ximage macro defined above. Its first argument is the name to be bound with the resulting form XObject. The second one is the number of the frame (actually a page number in the PDF file with frames). One needs to be careful here because pdfTEX has one-based page numbering while PDF uses zero-based page numbering internally.

```
% all animation frames are inserted
\ximage{fr0}{1}  \ximage{fr1}{2}
\ximage{fr2}{3}  \ximage{fr3}{4}
\ximage{fr4}{5}  \ximage{fr5}{6}
\ximage{fr6}{7}  \ximage{fr7}{8}
\ximage{fr8}{9}
```

Setting up an AcroForm Dictionary

The interactive features are realized by annotation elements in PDF. These form a separate layer in addition to the regular document content. Each one denotes an area on the page to be interactive and binds some actions to various events that can happen for that area. Annotations are represented by Annot dictionaries. The way pdfTEX inserts annotations into PDF is discussed in the section "Animation Dynamics" below.

Annotations are transparent by default, i.e., the page appearance is left unchanged when adding an annotation. It is up to the regular content to provide the user with the information that some areas are interactive.

We will be interested in a subtype of annotations called *interactive form fields*. They are represented by a Widget subtype of the Annot dictionary. Widgets can be rendered on top of the regular content. However, some resources have to be set. The document's Catalog refers to an AcroForm dictionary in which this can be accomplished.

The next part of the example first defines the name Helv to represent the Helvetica base-font (built in font). This is not necessary but it allows us to have a smooth control button. Next we insert the AcroForm dictionary. The DR stands for "resource dictionary". We only define the Font resource with one font. The DA stands for "default appearance" string. The /Helv sets the font, the 7 Tf sets the font size scale factor to 7 and the 0 g sets the color to be 0 % white (i.e., black). The most important entry in the AcroForm dictionary is NeedAppearances. Setting it to true (line 43) makes the Widget annotations visible. Finally, we add the AcroForm dictionary to the document's Catalog.

```
% the Helvetica basefont object
\insertobj{Helv}{
<< /Type /Font  /Subtype /Type1
   /Name /Helv
   /BaseFont /Helvetica  >> }

% the AcroForm dictionary
\insertobj{AcroForm}{
<< /DR << /Font <<
       /Helv \oref{Helv} >> >>
   /DA (/Helv 7 Tf 0 g )
   /NeedAppearances true >> }

% add a reference to the Catalog
\pdfcatalog{/AcroForm \oref{AcroForm}}
```

To make a form XObject with an animation frame accessible to JavaScript, it has to be assigned

a name. There are several namespaces in PDF in which this can be accomplished. The one searched for is determined from context. We are only interested in an AP namespace that maps names to annotation appearance streams. pdfTEX provides the `\pdfnames` primitive that behaves similarly to `\pdfcatalog`. Each time it is expanded it adds its argument to the Names dictionary referred from document's Catalog. The Names dictionary contains the name definitions for various namespaces. In our example we put definitions into a separate object AppearanceNames.

The name definitions may form a tree to make the lookup faster. Each node has to have Limits set to the lexically least and greatest names in its subtree. There is no extensive set of names in our example, so one node suffices. The names are defined in the array of pairs containing the name string and the indirect reference.

```
% defining names for frames
\insertobj{AppearanceNames}{
<< /Names
  [ (fr0) \oref{fr0}  (fr1) \oref{fr1}
    (fr2) \oref{fr2}  (fr3) \oref{fr3}
    (fr4) \oref{fr4}  (fr5) \oref{fr5}
    (fr6) \oref{fr6}  (fr7) \oref{fr7}
    (fr8) \oref{fr8} ]
  /Limits [ (fr0) (fr8) ] >> }

% edit the Names dictionary
\pdfnames{/AP \oref{AppearanceNames}}
```

Animation Dynamics

We have created all the data structures needed for the animation in the previous section. Here we introduce the code to play the animation. It uses Acrobat JavaScript [1], an essential element of interactive forms. Acrobat JavaScript is an extension of Netscape JavaScript targeted to PDF and Adobe Acrobat. Most of its features are supported by Adobe Reader. They can, however, be supported by any other viewer. Nevertheless, the Reader is the only one known to us that supports interactive forms and JavaScript.

The animation is based on interchanging frames in a single widget. Here we define the number of frames and the interchange timespan in milliseconds to demonstrate macro expansion in JavaScript.

```
% animation properties
\def\frames{8}
\def\timespan{550}
```

Every document has its own instance of a JavaScript interpreter in the Reader. Every JavaScript action is interpreted within this interpreter. This means that one action can set a variable to be used by another action triggered later. Document-level JavaScript code, e.g., function definitions and global variable declarations, can be placed into a JavaScript namespace. This code should be executed when opening the document.

Unfortunately, there is a bug in the Linux port of the Reader that renders this generally unusable. The document level JavaScript is not executed if the Reader is not running yet and the document is opened from a command line (e.g., '`acroread file.pdf`'). Neither the first page's nor the document's open action are executed, which means they cannot be used as a workaround. Binding a JavaScript code to another page's open action works well enough to suffice in most cases.

We redeclare everything each time an action is triggered so as to make the code as robust as possible. First we define the `Next` function, which takes a frame index from a global variable, increases it modulo the number of frames and shows the frame with the resulting index. The global variable is modified.

The animation actually starts at line 78 where the frame index is initialized. The frames are displayed on an interactive form's widget that we name `"animation"` — see "Placing the Animation" below. A reference to this widget's object is obtained at line 79. Finally, line 80 says that from now on, the `Next` function should be called every `\timespan` milliseconds.

```
% play the animation
\insertobj{actionPlay}{
<< /S /JavaScript /JS (
function Next() {
  g.delay = true;
  if (cntr == \frames) {
    cntr = 0;
    try { app.clearInterval(arun); }
        catch(except) {}
  } else {  cntr++; }
  g.buttonSetIcon(
      this.getIcon("fr" + cntr));
  g.delay=false;
}
try { app.clearInterval(arun); }
    catch(except) {}
var cntr = 0 ;
var g = this.getField("animation");
var arun = app.setInterval("Next()",
                           \timespan);
) >> }
```

Now, let us describe the `Next` function in more detail. Line 66 suspends widget's redrawing until line 74. Then the global variable containing the current frame index is tested. If the index reaches the number of frames, it is set back to zero and the periodic calling of the function is interrupted. The function would be aborted on error, but because we catch exceptions this is avoided. The `getIcon` function takes a name as its argument and returns the reference to the appearance stream object according to the AP names dictionary. This explains our approach of binding the names to animation frames — here we use the names for retrieving them. The `buttonSetIcon` method sets the object's appearance to the given icon.

Line 76 uses the same construct as line 69 to handle situations in which the action is relaunched even if the animation is not finished yet. It aborts the previous action. It would have been an error had the animation not been running, hence we must use the exception catching approach.

Placing the Animation

The animation is placed on an interactive form field — a special type of annotation. There are two primitives in pdfTEX, `\pdfstartlink` and `\pdfendlink`, to produce annotations. They are intended to insert hyperlink annotations but can be used for creating other annotations as well. The corresponding `\pdfstartlink` and `\pdfendlink` must reside at the same box nesting level. The resulting annotation is given the dimensions of the box that is enclosed by the primitives. We first create a box to contain the annotation. Note that both box and annotation size are determined by the frame itself — see line 91 where the basic frame is placed into the regular page content.

We will turn now to the respective entries in the annotation dictionary. The annotation is to be an interactive form field (`/Subtype /Widget`). There are many field types (FT). The only one that can take any appearance and change it is the *pushbutton*. It is a special kind of *button* field type (`/FT /Btn`). The type of button is given in an array of field bit flags Ff. The pushbutton has to have bit flag 17 set (`/Ff 65536`). To be able to address the field from JavaScript it has to be assigned a name. We have assigned the name `animation` to it as mentioned above (`/T (animation)`). Finally, we define the appearance characteristics dictionary MK. The only entry `/TP 1` sets the button's appearance to consist only of an icon and no caption.

```
83  % an animation widget
84  \centerline{\hbox{%
85    \pdfstartlink user{
86    /Subtype /Widget   /FT /Btn
87    /Ff 65536          /T (animation)
88    /BS << /W 0 >>
89    /MK << /TP 1 >> }%
90    \image{fr0}%
91    \pdfendlink}}
```

For the sake of brevity and clarity we are going to introduce only one control button in our example. However, we have defined a macro for creating control buttons to show a very simple way of including multiple control buttons. The `\controlbutton` macro takes one argument: the caption of the button it is to produce. The macro creates a pushbutton and binds it to an action defined like `actionPlay`.

We have chosen control buttons to be pushbuttons again. They are little different from the animation widget — they are supposed to look like buttons. The BS dictionary (i.e., border style) sets the border width to 1 point and style to 3D button look. The MK dictionary (appearance characteristics dictionary) sets the background color to 60% white and the caption (line 98). The `/H /P` entry tells the button to push down when clicked on. Finally, an action is bound to the button by setting the value of the A key.

```
92  % control button for a given action
93  \def\controlbutton#1{%
94    \hbox to 1cm{\pdfstartlink user{
95    /Subtype /Widget   /FT /Btn
96    /Ff 65536          /T (Button#1)
97    /BS << /W 1 /S /B >>
98    /MK << /BG [0.6] /CA (#1) >>
99    /H /P  /A \oref{action#1}
100   }\hfil\strut\pdfendlink}}
```

And finally, we add a control button that plays the animation just below the animation widget.

```
101 % control button
102 \centerline{\hfil
103   \controlbutton{Play}\hfil}
104
105 \bye
```

External Animation

Let us modify the example a little so that the animation frames will be taken from an external file. This has several consequences which will be discussed at the relevant points in the code.

We are going to completely detach the animation frames from the document. As a result, we will need only the `\insertobj` and `\oref` macros from

lines 1–23 from the previous example. Lines 26–31 are no longer required.

A problem arises here: the basic frame should be displayed in the animation widget when the document is opened for the first time. This can be accomplished by modifying the OpenAction dictionary at line 25 as follows.

```
\pdfcatalog{ /OpenAction <<
  /S /JavaScript /JS (
    var g = this.getField("animation");
    g.buttonImportIcon(
        "frames-ex.pdf",0);
    this.pageNum = 0;
    this.zoomType = zoomtype.fitP;
  ) >> }
```

This solution suffers from the bug mentioned in the "Animation Dynamics" section. The animation widget will be empty until a user performs an action every time the bug comes into play.

We still do need an AcroForm dictionary, so lines 32–46 are left without a change. Lines 47–58 must be omitted on the other hand, as we have nothing to name. We are going to use the same animation as in the previous example, so lines 59–61 are left untouched. There is one modification of the JavaScript code to be done. The `buttonSetIcon` function call is to be replaced by

```
g.buttonImportIcon(
    "frames-ex.pdf", cntr);
```

We have used the basic frame to determine a size of the widget in the previous example. This is impossible now because it has to be done at compile time. The replacement for lines 83–91 is as follows

```
% an animation widget
\centerline{\hbox to 6cm{%
  \vrule height 6cm depth 0pt width 0pt
  \pdfstartlink user{
    /Subtype /Widget  /FT /Btn
    /Ff 65536         /T (animation)
    /BS << /W 0 >>
    /MK << /TP 1
           /IF << /SW /A   /S /P
               /A [0.5 0.5] >> >> }%
  \hfil\pdfendlink}}
```

Dimensions of the widget are specified explicitly and an IF (icon fit) dictionary is added to attributes of the pushbutton so that the frames would be always (`/SW /A`) proportionally (`/S /P`) scaled to fit the widget. Moreover, frames are to be centered in the widget (`/A [0.5 0.5]`) which would be the default behavior anyway. The basic frame is not placed into the document — there is only glue instead.

Lines 92–105 need not be modified.

Two Notes on Animation Frames

The examples with full TEX source files can be found at `http://www.fi.muni.cz/~xholecek/animations/`. As one can see in these examples, the all-in-one approach allows all frames to share a single background which is formed by the frame actually inserted into the page. However, it is possible to overlay pushbuttons. Elaborate constructions, the simplest of which is to use a common background frame in the example with external animations, can be achieved in conjunction with transparency.

One must ensure the proper size of all frames when fitting them into the widget. We have encountered situations (the given example being one of them) where the bounding box of METAPOST generated graphics with TEX label was not set properly using `\convertMPtoPDF` and a white line had to be drawn around the frames to force the proper bounding box as a workaround.

Animations in Other Formats

It is fair to list and compare other possible ways of creating animations. In this section we give a brief overview of a dozen other formats and technologies capable of handling animations.

GIF One of the versions of the GIF format is the GIF89a format, which allows multi-image support, with bitmap only animations to be encoded within a single GIF file. GIF format supports transparency, interlacing and plain text blocks. It is widely supported in Internet browsers. However, there are licensing problems due to the compression methods used, and the format is not supported in freely available TEXware.

SWF The SWF format by Macromedia allows storing frame-based animations, created e.g., by Macromedia's Flash authoring tool. The SWF authoring tools have to compute all the animation frames at export time. As proprietary Flash plug-ins for a wide range of Internet browsers are available, animations in SWF are relatively portable. The power of SWF can be enriched with scripting by ActionScript. At the time of writing, we are not aware of any TEXware supporting SWF.

Java One can certainly program animations in a general programming language like Sun's Java. The drawback is that there are high demands on one's programming capabilities in Java when creating portable animations. With $\mathcal{NTS}$ (a TEX reimplementation in Java), one can possibly combine TEX documents with fully featured animations, at

the expense of studying numerous available classes, interfaces and methods.

DOM It is possible to reference every element in an HTML or XML document by means of the W3C's Document Object Model (DOM), a standard API for document structure.

DOM offers programmers the possibility of implementing animations with industry-standard languages such as Java, or scripting languages as ECMAScript, JavaScript or JScript.

SVG The most promising language for powerful vector graphic animation description seems to be Scalable Vector Graphics (SVG), a W3C recommendation [12]. It is being developed for XML graphical applications, and since SVG version 1.1 there is rich support for animations. The reader is invited to look at the freely available book chapter [13] about SVG animations on the publisher's web site, or reading [4] about the first steps of SVG integration into TEX world. There are freely available SVG viewers from Adobe (browser plug-in), Corel, and the Apache Foundation (Squiggle).

SVG offers even smaller file sizes than SWF or our method. The description of animations is time-based, using another W3C standard, SMIL, Synchronised Multimedia Integration Language. The author can change only one object or its attribute in the scene at a time, allowing detailed control of animated objects through the declarative XML manner. Compared to our approach, this means a much wider range of possibilities for creators of animations.

The SVG format is starting to be supported in TEXware. There are SVG backends in VTEX and BaKoMaTEX, and a program Dvi2Svg by Adrian Frischauf, available at `http://www.activemath.org/~adrianf/dvi2svg/`. Another implementation of a DVI to SVG converter in C is currently being developed by Rudolf Sabo at the Faculty of Informatics, Masaryk University in Brno.

Conclusions

We have shown a method of preparing both space-efficient and high-quality vector frame-based animations in PDF format using only freely available, TEX-integrated tools.

Acknowledgments

Authors thank Oleg Alexandrov and Karl Berry for comments on an early draft of the paper.

The work has been supported by VZ MSM 143300003.

References

[1] Adobe Systems Incorporated. Acrobat JavaScript Object Specification, Version 5.1, Technical Note #5186. Technical report, Adobe, 2003. `http://partners.adobe.com/asn/developer/pdfs/tn/5186AcroJS.pdf`.

[2] Adobe Systems Incorporated. *PDF Reference: Adobe Portable Document Format Version 1.5*. Addison-Wesley, Reading, MA, USA, fourth edition, August 2003.

[3] Zuzana Došlá, Roman Plch, and Petr Sojka. Mathematical Analysis with Maple: 2. Infinite Series. CD-ROM, `http://www.math.muni.cz/~plch/nkpm/`, December 2002.

[4] Michel Goossens and Vesa Sivunen. LATEX, SVG, Fonts. *TUGboat*, 22(4):269–280, Oct. 2001.

[5] Hans Hagen. The Calculator Demo, Integrating TEX, METAPOST, JavaScript and PDF. *TUGboat*, 19(3):304–310, September 1998.

[6] Petr Olšák. *TEXbook naruby (in Czech)*. Konvoj, Brno, 1997.

[7] Petr Sojka. Animations in PDF. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2003*, page 263, Thessaloniki, 2003. Association of Computing Machinery.

[8] Petr Sojka. Interactive Teaching Materials in PDF using JavaScript. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2003*, page 275, Thessaloniki, 2003. Association of Computing Machinery.

[9] Donald P. Story. AcroTEX: Acrobat and TEX team up. *TUGboat*, 20(3):196–201, Sep. 1999.

[10] Donald P. Story. Techniques of introducing document-level JavaScript into a PDF file from LATEX source. *TUGboat*, 22(3):161–167, September 2001.

[11] Hán Thế Thánh. Micro-typographic extensions to the TEX typesetting system. *TUGboat*, 21(4):317–434, December 2000.

[12] W3C. *Scalable Vector Graphics (SVG) 1.1 Specification*, January 2003.

[13] Andrew H. Watt. *Designing SVG Web Graphics*. New Riders Publishing, September 2001.