Presentable Document Format: Improved On-demand PDF to HTML Conversion

Ralph Sommerer
Microsoft Research
7 J J Thomson Avenue
Cambridge CB3 0FB, United Kingdom
som@microsoft.com

ABSTRACT
Search engines such as Google and MSN Search crawl and index files in Adobe's Portable Document Format (PDF) alongside material in HTML. Google furthermore offers a *View as HTML* option for PDF that includes query term highlighting. The visual appearance of these HTML files converted from PDF is very poor. In this paper we claim that significant improvements to the quality of on-demand PDF to HTML conversion can be achieved at insignificant cost in terms of increased file size and processing time. We can show in particular, that a slightly more sophisticated HTML coding can easily compensate for the increase in file size when including line graphics and images.

Categories and Subject Descriptors
E.4 [**CODING AND INFORMATION THEORY**]: Data compaction and compression; H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval; H.3.5 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based services*; H.5.4 [**INFORMATION INTERFACES AND PRESENTATION**]: Hypertext/Hypermedia; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Optimization, Parsing*; I.7.2 [**DOCUMENT AND TEXT PROCESSING**]: Document Preparation—*Markup languages, Format and notation*; I.7.5 [**DOCUMENT AND TEXT PROCESSING**]: Document Capture—*Document analysis*

General Terms
Algorithms, Documentation, Performance, Design.

Keywords
Portable Document Format, HTML optimization, document conversion, Web services, search engines.

1. INTRODUCTION
Adobe's Portable Document Format (PDF) is a popular file format for the distribution of online publications. Because of the popularity and ubiquity of PDF, search engines such as Google [4], and more recently also MSN search [8] crawl and index PDF files alongside material in HTML (and other formats like that of Microsoft Word). Additionally, Google also offers a "view as HTML" option to preview PDF files in the result set in HTML form. Previewing in HTML has a particular advantage, because it allows Google to highlight query terms in the document, a feature that is only very difficult to achieve in PDF without actually changing the document itself.

The previewing-as-HTML facility in Google is a useful feature that allows users to assess the relevance of a PDF file before it is downloaded. However, the visual representation of the PDF file in HTML is usually very poor. Figure 1 shows a moderately complex page of a PDF file and its HTML representation as shown by Google. The two obviously share a faint resemblance regarding the arrangement of text. But otherwise it is hard to identify one as a conversion of the other.

This paper argues that a more sophisticated conversion of PDF files to HTML including images and line graphics can be accomplished at only moderate cost with regard to file size or processing time. In particular by employing a slightly more complex text flow reconstruction, the increase in file size due to the inclusion of figures in all but some pathological cases is easily compensated for by a more compact HTML coding of the text.

In the following section 2 we will briefly discuss a few other existing PDF to HTML converters, both standalone and online. Then, we will introduce the PDF file format in section 3 and discuss the basics of the conversion techniques that are common among the various HTML converters. Sections 4 and 5 will give an overview of the software architecture and discuss in some detail the conversion algorithms. A performance evaluation with regard to file size and conversion efficiency follows. Finally, our conclusions and a bibliography will complete this paper.

2. OTHER WORK

2.1 Standalone PDF converters
There are many proprietary standalone PDF converters available that extract text from PDF and represent the result either as plain text, or convert it to various document formats including HTML [3], [5], [9]. The purpose of these converters is to provide an alternative format for PDF files in order to make them available for Web users or search engines [5]. Under these circumstances PDF files usually need to be converted only once, for example before being published on the Web. These converters are therefore optimized for *quality*, and the result can be expected to be nearly identical to the original PDF and in fact may actually replace it.

In contrast to standalone offline converters ours is an online on-demand converter that is *not* optimized for quality. The result of the conversion is therefore still a far shot from that of the original PDF. Instead, our converter is optimized for compactness of the result, and for efficiency of the computation to enable an on-demand HTML conversion of reasonable quality and similarity to the original.

2.2 Online PDF converters

A quick Google query reveals that there are not that many online PDF converters available on the Web. The most notable is Adobe's own online converter [1]. Incidentally, it is also the one pointed to by many of the result sites in said Google query that mention online PDF conversion.

Adobe's converter is a simple text flow reconstruction utility that preserves text looks (font and styles) and column widths but disregards all other layout information. The result is a linear single-column HTML file with explicit line breaks where the PDF file has implicit line breaks at column margins. It does not mimic the layout of the original PDF file and, in particular, does not show page breaks.

Despite the simplicity of the result, Adobe's online conversion is quite slow, and takes tens of seconds to complete.

2.3 Google's *View as HTML*

Google's online PDF-to-HTML conversion is more sophisticated than Adobe's because it preserves the original two-dimensional layout in a page-wise arrangement. This is accomplished by providing an absolutely positioned `<div>` element for each line of text. There does not appear to be any higher level layout structure beyond text lines, but to avoid excessive repetition of font attributes, several lines sharing the same font are sometimes grouped together. All in all, however, in addition to having a poor appearance the generated HTML coding is also inefficient and repetitive.

3. BACKGROUND

3.1 Portable Document Format

There are two common misperceptions regarding PDF. The first one is a consequence of the format's brand name, namely that PDF is a *document* format comparable to Microsoft Word's or HTML. PDF is not a document format, because it doesn't store a document, but rather a document's *rendering*. A PDF file contains instructions where to leave marks (ink) on a page, and therefore is actually a *graphics meta-file*, or more precisely a *print file*.

The opposite misperception is that, given that PDF is not a document format, its content is essentially a bitmap and therefore requires optical character recognition (OCR) techniques to extract text. The truth is somewhere in the middle: because PDF files are print files, the character information is usually explicit, but because they are *not* document files, the text flow information is lost. Therefore, text extraction is essentially text flow reconstruction. (Note that there are two exceptions to the above: first, there PDF files whose contents *are* bitmaps, e.g. scanned documents, but we regard them as image files embedded in PDF and don't discuss them any further here. Second, more recent versions of the format *do* define explicit text flow information.)

A (physical) PDF file consists of a collection of *objects* that are identified by a pair of numbers (id, version number). These translate via a cross-reference table into absolute file positions. Objects have either scalar form like numbers and strings, or compound form based on dictionaries (name-value pairs), arrays (of arbitrary element types), and streams (dictionaries with an attached data sequence of arbitrary length and content type, subjected to various different compression and encoding schemes). A *reference object* (similar to a pointer) in place of a value allows for a level of indirection. Different object types are

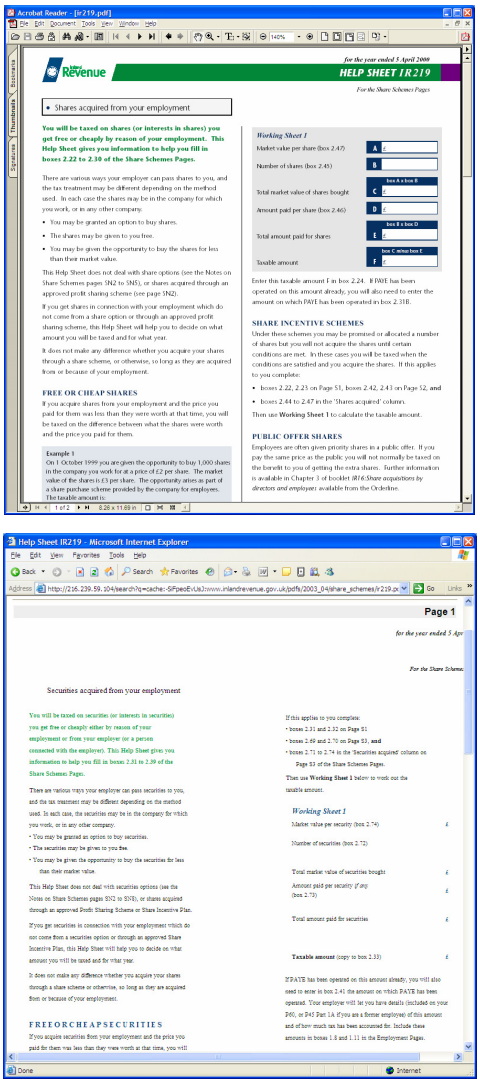

Figure 1. Example page: PDF (top), Google's *View as HTML* (bottom)

distinguished by a *Type* entry in the corresponding dictionary. Although PDF files are considered binary files, the file syntax and all values, names etc. are in ASCII form and therefore have to be parsed. Only the contents of streams are usually rendered unreadable because of compression filters applied to the content.

Logically, a PDF file consists of a number of page descriptions (dictionaries with *Type* "Page"). A page description contains general attributes of the page, such as its size, orientation, a resources dictionary that lists fonts, images etc. needed to render the page, and, most importantly, a *content stream* that contains the marking (drawing) commands to render the page. The content stream is a sequence of (ASCII) commands and their attributes in postfix notation. The drawing command vocabulary includes commands to set attributes (color, line thickness, fonts, etc.), to do line graphics (move-to, line-to, stroke/fill a path), to draw strings (with/without inter-word and/or line spacing etc.) and to draw images (taken from the resources dictionary attached to the page, or specified inline for small images).

3.2 Text Extraction

Extracting text from a PDF file requires executing the commands of the pages' content streams to establish the proper graphics state and to assemble the data of text drawing commands (character codes and coordinates). The character codes are then merged to ever longer strings according to various properties of the graphics state (e.g. whether the characters are close enough to be part of the same word, and whether they align properly and thus form a text line). From the graphics state, color and font information can also be gathered. Because in PDF the text flow information is lost, all PDF-to-text converters have to employ this or a very similar technique to reconstruct the text flow.

Geometry

In PDF text is usually drawn in "natural order" e.g. left to right, top to bottom, and is therefore in the correct order to be assembled into ever longer strings, lines and paragraphs. A new string being printed is considered part of the line currently under construction if it overlaps vertically with the line (allowing for super-/subscripts) and is less than a certain fraction (e.g. 40%) of the font height away from the current right end of the line. A new line is started whenever a string does not continue the previous one.

After the strings are assembled into lines, a vertical arrangement of lines is assembled called *text box*. The following algorithm gives an initial estimate which subsequent lines are part of the same text box, i.e. a group of lines sharing the same inter-line space. This is the basis for computing higher level logical structures such as paragraphs (a box may contain several paragraphs and a paragraph my span several boxes):

1. Lines obtained using the above rules are enumerated in the natural order (i.e. as they are printed).
2. The distance between the first pair of lines is computed.
3. While the distance to the new line is within a small margin (e.g. 0.05 * line-height) of the computed distance, the line is added to the box, updating its bounds.
4. If the new distance is bigger, the new line is presumed to belong to a new box. If the new distance is smaller, both the previous and new lines are presumed to belong to a new box. The corresponding distance is set as the boxes' inter-box distance (the line space is the inner-box distance).

Above algorithm produces a list of boxes bounding all sequences of lines sharing the same inter-line space.

Styles

In the course of all of above text assembling steps, style information (in particular font face and size) is collected and, if it is shared among all elements, inherited "upwards" to the next higher level. Thus, if all strings in a line share the same font, the whole line also assumes the font. Similarly, if all lines in a box share the same font, the whole box assumes the font as well. This factoring out of font information to higher structural levels is the key to a more efficient i.e. more compact conversion to HTML by avoiding unnecessary repetition of font and position information [12]. Font *styles* (bold/italic) are not included in the font information shared between lines and boxes because they tend to change more frequently.

4. ARCHITECTURE

The overall architecture of our prototype PDF extraction and HTML conversion service, and the algorithms and processes that drive it, are based on the design decision to avoid *any duplication of data*. We regard a PDF file as an *archive* of objects (text, graphics, and images), therefore there is no need to extract and store separately from it any information contained therein. As a consequence, we compute the HTML from a PDF file *on demand* and *upon request*. Also, we extract all secondary features such as images etc. on-the-fly directly from the file as well. Modern search engines usually have an associated file storage containing all crawled documents that they can deliver the "snippets" from i.e. the text excerpts with highlighted query terms that accompany the search results. Google even includes a link in the result set allowing users to actually pull the target page from the storage. This is especially useful when the originals are found to be temporarily inaccessible, or if they have been removed since the engine has indexed them. Considering the amount of data that this storage is required to hold, it is clearly undesirable to duplicate unnecessarily any information that can be extracted from an existing file without undue effort at the time it is requested. By extracting the data directly out of the PDF file we avoid any unnecessary duplication of data.

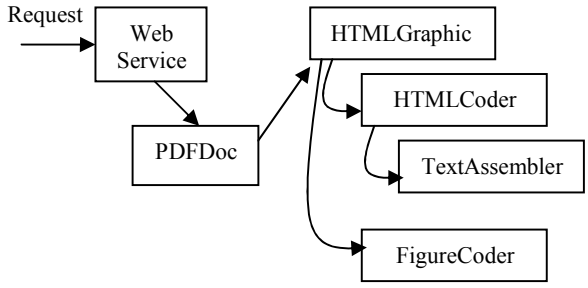

Figure 2. Software architecture of the PDF to HTML conversion service.

The PDF converter whose software architecture is depicted in figure 2 is a *Web service* that provides HTML versions of PDF files upon request. It accepts `GET` requests to convert files from the local archive, and `POST` requests to convert PDF data that is uploaded as part of the request. The PDF file is then opened and processed i.e. "printed" if it is found to be valid. *PDFDoc* contains the PDF parser and the logic to print pages, i.e. to execute the commands of the pages' content streams. For each graphics command encountered while executing the content stream a method of an interface named *Graphics* is called. This interface is supplied as an argument to the *PDFDoc's* print method. The component *HTMLGraphic* in figure 2 implements that interface. *HTMLGraphic* then uses the services of *HTMLCoder* and *FigureCoder* to create HTML and figures, respectively. The text flow is reconstructed with the help of the *TextAssembler* component. It employs the methods described in section 3.2. *HTMLCoder* and *FigureCoder* write the resulting code directly onto the wire. The inner workings of the latter two components are explained in more detail in the following sections.

5. HTML CONVERSION

5.1 Text

The result of the text extraction phase is a data structure that integrates text flow information (text and style) with placement information (text boxes grouping text lines that share the same inter-line spacing). For the sake of simplicity we have ignored the higher level text flow information such as paragraphs etc. and rely only on the box and line structure for conversion to HTML.

The layout of the text is encoded in HTML using a nested structure of absolutely positioned HTML `<div>` elements, with *Cascading Style Sheet* (CSS) [14] properties to store position and look (absolute positioning of `<div>` elements is set globally in the HTML's header section). The top level of the structure is a sequence of `<div>` elements, one for each page. Nested within these are text boxes and lines. The advantage of this arrangement is that (vertical) coordinates can be relative to the "outer" page element and therefore limited in range and hence number of digits used to represent them. The following example shows the top 5 lines of a text box and below its representation in HTML:

Lorem ipsum dolor sit
amet, consectetur
adipisicing elit*, sed*
do eiusmod tempor
incididunt ut labore
…

```
<div style='left:68;top:261;font:9pt serif'>
 <i>
  <div style='top:0'>Lorem ipsum dolor sit</div>
  <div style='top:14'>amet, consectetur</div>
 </i>
 <div style='top:29'>
  <i>adipisicing, </i>elit<i>, sed</i>
 </div>
 <i>
  <div style='top:44'>do eiusmod tempor</div>
  <div style='top:59'>incididunt ut labore</div>
  …
 </i>
</div>
```

In HTML the box is represented using a `<div>` element, and for each text line of the box there's a nested `<div>` element. Lines require only a vertical coordinate (offset from the top of the surrounding box) because the left margin is shared among all lines within a box. Indented lines do have a left coordinate which is an offset from the left margin of the box. Note that the font face and size are set at the box level and hence shared by all lines. Variations of the font style such as *italics* are applied to lines and line fragments as appropriate.

5.2 Figures

Unfortunately, there is still not yet a simple, agreed-upon standard format for representing simple line graphics on the Web. Page authors usually revert to bitmap images for representing graphics, at a cost of lower figure quality, and both higher complexity and bandwidth requirements. The closest to what we can consider a simple yet powerful enough format for line graphics on the Web is Microsoft's proposal for the Vector Markup Language (VML) [13]. The competing standard called Scalable Vector Graphics (SVG, [11]) is more complex due to an attempt to emulate more elaborate formats such as Macromedia's Flash [7]. To represent line graphics in our prototype, we chose to employ the simpler VML (which incidentally happens to be natively supported in Microsoft's Internet Explorer).

Transformation of PDF line drawing commands to VML is more or less trivial, because PDF's line-drawing objects (paths) can be translated directly into VML elements such as `<polyline>`, `<line>` or `<shape>`. Without any optimization, however, the size of the resulting VML code will be prohibitive, because VML is a rather verbose way of representing line graphics. Also, there will be a lot of repetition because graphics properties such as stroke and fill colors that are usually shared across subsequent graphics operations must be included as an attribute separately with every VML element (unless they are equal to the default values for these colors). To obtain a more compact representation, it is therefore necessary to fold as many compatible graphics operations as possible into a single VML `<shape>` element. The following observations guide these optimizations.

Outlined Shapes

In PDF the same outline cannot be filled using one color and outlined using another in a single step. Rather, two distinct operations are required. Consequently, two (or more) graphics operations using the same path are quite common. Because VML shape elements contain a stroke *and* a fill color, we can merge subsequent filling and drawing operations on the same path into a single VML shape. This is even true if the poly-line is not closed (i.e. a polygon *a-b-c-a* can be merged with a poly-line *a-b-c*).

Sequence of shapes

If there's a sequence of shapes sharing the same stroke and fill colors we can fold them into a single `<shape>` element's path attribute. While this does not reduce the number of VML operations required to represent the graphic, it allows for a more compact representation by avoiding repetition of the element's tag name and its color attributes. Furthermore, in a shape element's path shorter relative coordinates can be used for the vertices. Nevertheless, figures add significantly to the size of a converted document.

We found VML to be very simple yet powerful enough for our purposes, and we expect it to be so for most line graphic needs of the Web publishing industry. We therefore expect it to be implemented soon by various browser manufacturers, even if they intend to go for the more complex SVG as well. Nevertheless, a pluggable figure converter in our PDF service allows switching to a different coding scheme (e.g. SVG if that is what the browser supports), or a simple fall-back coding using pure HTML but with severely limited capabilities (only boxes and horizontal/vertical lines are shown).

5.3 Images

5.3.1 Addressing

Images in PDF are data streams. There are two kinds of images with respect to the way they are stored in PDF files: (1) in-line images that appear in the page's content stream, and (2) so called *XObjects* which are located in a page's resource dictionary. The latter can be accessed given the pair (page number, image name). The former require positioning the page's content stream to the exact position where the in-line image starts and then executing (i.e. parsing) the content stream from that location on. Note that positioning of the content stream may actually require scanning the stream linearly from the start up to that position because the stream may be subjected to a compression filter.

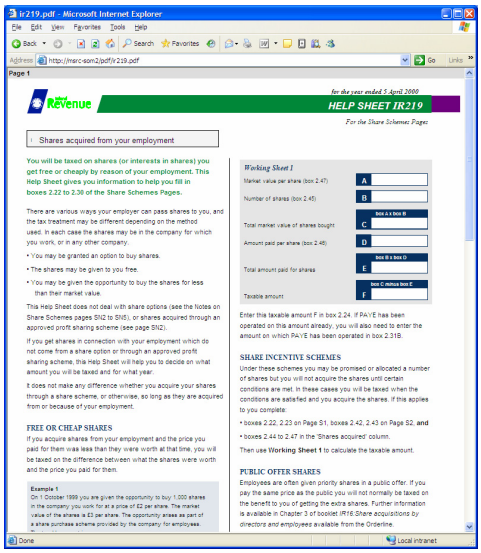

Figure 3. PDF file from figure 1 run through our converter.

5.3.2 *Coding*

Photographic images are usually stored in PDF using a *discrete cosine transformation* filter that is familiar from the JPEG image compression [10]. In fact, the image format is identical to the one employed in JPEG. Extracting such an image thus requires only copying the stream to the wire without touching it.

Other non-photographic images are usually bitmaps with either indexed (via color lookup table) or direct color coding (using RGB or various other color schemes). These are stored as pixel-maps without any further formatting but possibly subjected to a compression filter. Before transmitting them to the client they need to be repackaged as a bitmap stream in a common bitmap format. In our prototype we package the image data as BMP [16] streams because this requires only prefixing the pixel data with a BMP header and possibly realigning the pixel data on 4-byte boundaries (BMP files are not compressed).

5.3.3 *Extraction*

Images represented as XObjects can be extracted and served directly from the PDF. If during the text extraction process, an image is encountered that is of XObject type, an image tag is inserted into the HTML page with a URL that encodes the page number, the image name and the expected coding of the image (JPEG or BMP).

```
<img src='pageNo/ImageName.jpg'/>
```

If the browser later resolves the URL the server can extract the image from the PDF using the page number and image name, and *directly* stream the data back to the client. In-line image are handled differently using an URL-coding scheme.

As explained above, extracting inline images is expensive because they require scanning the content stream up to the image's starting position (decompression is a process where any code x_i is a function of all previous codes x_0 to x_{i-1}). However, extracting inline images is cheap at the time they are encountered while extracting text. There is no way to "skip" past an inline image during text extraction; therefore, the content stream must be positioned behind an inline image by parsing the image data. It is of course possible to extract such images to a cache for later delivery but we chose a different approach: we "inline" the images into the HTML. There is, however, no direct way of inlining image data into an HTML page in a way that could be processed by browsers, but we have devised an indirect way that places the bulk of the work when it is cheap and defers the rest to when it is easy. We do it by encoding the image data into the image *URL*.

This is done by (1) compressing the image data and then (2) Base64-encoding [6] the compressed image string. The resulting character string is then set as the `src` attribute of an `<img>` tag to be added to the HTML document. Note that the image URL *is* the image and doesn't merely link to it. Nevertheless, a server interaction is still required because the image URL needs to be resolved. However, resolution is essentially confined to re-coding the image URL (Base64-decoding and decompressing) and sending the resulting data back. The image Web service thus acts as a reflector.

While this technique has the appearance of a "hack" it actually mirrors a common technique in computing to improve efficiency: a potentially complex operation is done optimistically when it is cheap to do so, irrespective of whether it may be needed at all. Later, when the task proves to be necessary, the expensive work is already done, and the remaining processing is relatively cheap.

6. EXAMPLES

A few examples may serve to illustrate the improvement obtained by including figures and images with converted PDF files. Figure 3 depicts the PDF file seen in figure 1, but this time converted via our converter. It goes without saying that it resembles the original PDF file more closely than Google's version. Figure 4 shows the details of a two further PDF files, one in each row, with the original in the left column, Google's *View as HTML* in the middle and our conversion on the right.

7. PERFORMANCE

The following tables and figures show the results of an experiment to measure the time required for the various phases of the conversion, and to establish the resulting file sizes. We measured the times and sizes of a selection of PDF files that were chosen to cover a wide range of situations: from no or little graphics to extensive graphics.

Table 1. Document properties

File	PDF size	Pages	Shapes[1]	Shapes/Page
1	479232	130	945	7.27
2	224349	16	200	12.5
3	291599	14	262	18.71
4	109705	9	225	25
5	196818	18	515	28.61
6	238592	16	843	52.69
7	103424	2	185	92.5
8	236048	7	1744	249.14
9	909413	108	10261	95.01

1. Graphics primitive: a polyline with ≥ 2 vertices, outlined and/or filled

The table 1 shows the file sizes and graphics properties of a selection of PDF documents. The first few are scientific papers with only a few figures each, and the last one is a data sheet with a

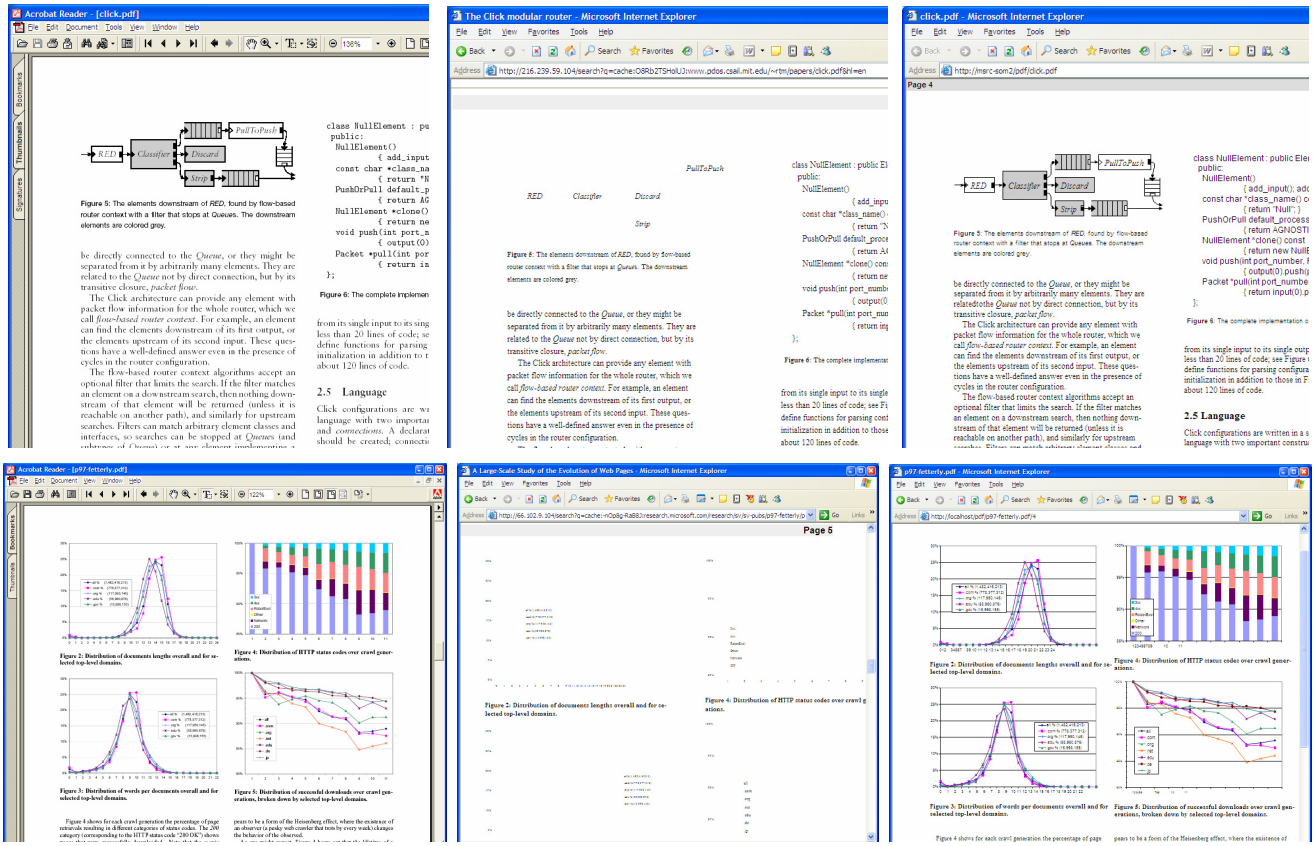

Figure 4. Details of two PDF files. Original (left column), Google's *View as HTML* (middle), our conversion (right).

lot of complex line graphics. Graphics complexity is indicated by the number of graphics primitives (shapes) per page.

Note that the average number of shapes per page can give an indication of the overall graphics complexity of a file, but does not necessarily reflect the true distribution of graphical contents among the pages. However, this does not influence the performance measures (file sizes, conversion times) much because they are aggregates for the full files.

Table 2: File sizes

File	PDF	Google[1]	*HTML +VML*	*HTML only*	*limited*[2]
1	479232	525390[2]	602101	536251	444241
2	224349	222059	191718	170824	
3	291599	241142	181786	160697	
4	109705	130497	100399	79683	
5	196818	0[3]	104158	70194	
6	238592	196710	201291	133814	
7	103424	34673	31896	20088	
8	236048	149870	198366	114822	
9	909413	525370[2]	1356670	805634	389899

1. Google size includes a small page header
2. Google apparently limits the amount of PDF served; *limited* figures are HTML+VML sizes for the same number of pages
3. File unavailable for *View as HTML*

We compare three variants of each document (table 2 and figure 5): the original PDF file (column heading *PDF*), the version obtained by Google's View as HTML (*Google*), and our version (*italics*) in both "HTML plus graphics" and "HTML only". The comparison clearly shows that the inclusion of line graphics can be accomplished at no or only marginal additional cost. Where there are no or only very few simple figures, the more sophisticated HTML representation leads to a net reduction in the file size, and where the ratio is below about 50 shapes per page

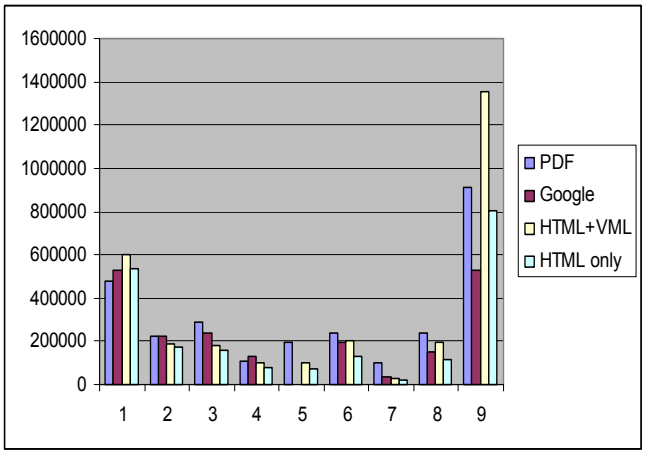

Figure 5. File sizes

the increase in size due to the figures is more or less compensated for by the more sophisticated HTML representation. In most cases the converted HTML including graphics are smaller than the PDF file they are converted from – notably before compression. This is significant because PDF files are usually compressed.

The datasheet (document number 9) makes extensive use of complex line graphics. It is therefore no surprise that the file size obtained by including line graphics by far exceeds that obtained via Google's *View as HTML* feature. However, Google limits PDF viewing to about 512kBytes of data (about 42 pages in the datasheet). The datasheet in question has 108 pages, and most figures are in the second half of the document. When compared with only the first 42 pages, our converter results in file size that is much smaller (see column 'limited').

Table 3 and figure 6 show the processing times of the various phases of the conversion for the same PDF files. We measured the time by running the each PDF file through different increasingly complex converters: a null conversion to measure PDF parsing time, a simple text extraction, a full HTML conversion with figures turned off, and finally full conversion including figures. Each phase is run 3 times, and the average taken. An initial run which is not included in the count is a full conversion to make sure that disk caches are filled, and all program objects loaded and initialized. Because we don't know what type of conversion Google employs we singled out the text extraction phase as it provides a fair lower bound to compare our converter with. Google's processing time is likely to be somewhere between those of text extraction and HTML conversion.

Table 2: Processing times (msec)

File	Parse Time	Text Extraction	HTML Creation	VML Creation
1	524	197	170	30
2	240	107	36	17
3	364	73	40	23
4	157	53	13	4
5	240	47	26	11
6	233	67	73	14
7	36	10	4	10
8	300	64	29	44
9	1372	310	214	243

It can be seen in table 2 and figure 6 that while adding figures *does* contribute measurably to the overall processing time, the increase is modest and only a fraction of the time that is required to simply *parse* PDF. Even for documents with extensive use of graphics the additional processing time is only of the order of the time required to extract text or convert to HTML.

In our performance assessment we discovered a rare pathological case where the inclusion of figures results in an increase of the total file size by a factor of 20. This is due to a 3D plot painted back to front in order to obscure hidden surfaces. We found that the processing time is still within reasonable bounds and in particular *less* than the time required to merely parsing the PDF file (parsing 2136 msec, VML creation 1309 msec). However, the corresponding increase in size is clearly totally unacceptable (HTML 118K, HTML+VML 2.5M). It may come as a surprise that the problem rests not so much with the PDF service which is perfectly capable of creating and delivering the data in reasonable time, but with the *browser* that is completely overwhelmed by such amounts of VML graphics. Indeed, it freezes up completely when faced with a drawing of that sort. Some measures such as a limitation of the amount of data returned by the Web service (comparable to Google's 512kBytes limit) may therefore be required – to protect the integrity of browsers.

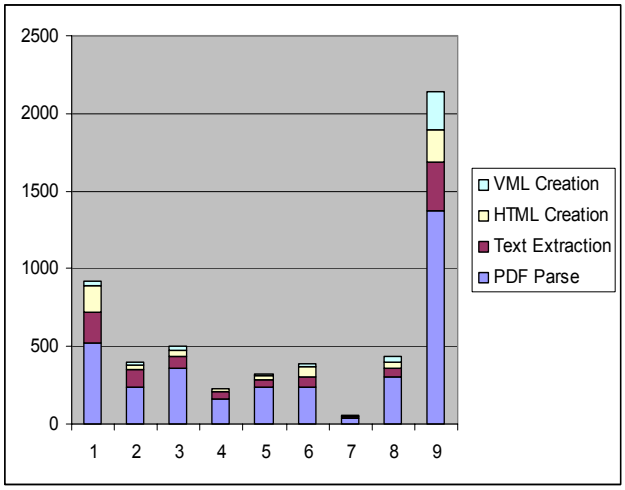

Figure 6. Processing times for different phases.

8. CONCLUSIONS

In this paper we have presented and evaluated a scheme to improve the appearance of PDF files that are "Viewed as HTML". In particular we have visual and numerical evidence to show that a significant improvement of the resulting HTML can be accomplished at almost no additional cost in terms of file sizes, and only moderate cost in terms of processing time. An increase in the file size due to the inclusion of graphics commands into the HTML file is easily compensated for by a slightly more sophisticated HTML synthesis and hence more compact HTML. Our prototype PDF server which is written in C# can convert a posted PDF file of a few dozen pages in a few seconds, including figures and images.

9. REFERENCES

[1] Adobe's PDF to HTML converter http://access.adobe.com/simple_form.html

[2] Adobe Systems Incorporated. Portable Document Format. Reference Manual, Version 1.3, March 11, 1999

[3] BCL, http://www.gohtm.com

[4] Google. http://www.google.com

[5] IntraPDF, http://www.intrapdf.com

[6] Josefsson, S. (Ed.), RFC 3548 - The Base16, Base32, and Base64 Data Encodings, http://www.faqs.org/rfcs/rfc3548.html

[7] Macromedia Flash, http://www.macromedia.com/software/flash/

[8] MSN search. http://search.msn.com

[9] pdftohtml http://www.ra.informatik.uni-stuttgart.de/~gosho/pdftohtml/

[10] Pennebaker, W. B., Mitchell, J. L., The JPEG Still Image Data Compression Standard, Van Nostrand Reinhold, 1993

[11] Scalable Vector Graphics (SVG), http://www.w3.org/Graphics/SVG/

[12] Spiesser, J., Kitchen, L., Optimization of HTML Automatically Generated by WYSIWYG Programs, Proc 13th World Wide Web Conf. (WWW2004), May 17–22, 2004, New York City, NY, USA, May 2004

[13] W3C. Vector Markup Language (VML), Draft specification http://www.w3.org/TR/NOTE-VML.html

[14] W3C. Cascading Style Sheets, Level 2 CSS2 Specification, http://www.w3.org/TR/REC-CSS2/

[15] W3C. XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), http://www.w3.org/TR/xhtml1

[16] Windows GDI - Bitmap Storage, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdi/bitmaps_4v1h.asp