Apache FOP Output Formats

$Revision: 681307 $

by Keiron Liddle, Art Welch

Table of contents

FOP supports multiple output formats by using a different renderer for each format. The renderers do not all have the same set of capabilities, sometimes because of the output format itself, sometimes because some renderers get more development attention than others.

1. General Information

1.1. Fonts

Most FOP renderers use a FOP-specific system for font registration. However, the Java2D/AWT and print renderers use the Java AWT package, which gets its font information from the operating system registration. This can result in several differences, including actually using different fonts, and having different font metrics for the same font. The net effect is that the layout of a given FO document can be quite different between renderers that do not use the same font information.

Theoretically, there's some potential to make the output of the PDF/PS renderers match the output of the Java2D-based renderers. If FOP used the font metrics from its own font subsystem but still used Java2D for text painting in the Java2D-based renderers, this could probably be achieved. However, this approach hasn't been implemented, yet.

With a work-around, it is possible to match the PDF/PS output in a Java2D-based renderer pretty closely. The clue is to use the intermediate format. The trick is to layout the document using FOP's own font subsystem but then render the document using Java2D. Here are the necessary steps (using the command-line):

1. Produce an IF file: `fop -fo myfile.fo -at application/pdf myfile.at.xml`
   Specifying "application/pdf" for the "-at" parameter causes FOP to use FOP's own font subsystem (which is used by the PDF renderer). Note that no PDF file is created in this step.
2. Render to a PDF file: `fop -atin myfile.at.xml -pdf myfile.pdf`
3. Render to a Java2D-based renderer:
   - `fop -atin myfile.at.xml -print`
   - `fop -atin myfile.at.xml -awt`
   - `fop -atin myfile.at.xml -tiff myfile.tiff`

1.2. Output to a Printer or Other Device

The most obvious way to print your document is to use the FOP print renderer, which uses the Java2D API (AWT). However, you can also send output from the Postscript renderer directly to a Postscript device, or output from the PCL renderer directly to a PCL device.

Here are Windows command-line examples for Postscript and PCL:

```
fop ... -ps \\computername\printer
fop ... -pcl \\computername\printer
```

Here is some Java code to accomplish the task in UNIX:

```
proc = Runtime.getRuntime().exec("lp -d" + print_queue + " -o -dp -");
out = proc.getOutputStream();
```

Set the output MIME type to "application/x-pcl" (MimeConstants.MIME_PCL) and it happily sends the PCL to the UNIX printer queue.

2. PDF

PDF is the best supported output format. It is also the most accurate with text and layout. This creates a PDF document that is streamed out as each page is rendered. This means that the internal page index information is stored near the end of the document. The PDF version supported is 1.4. PDF versions are forwards/backwards compatible.

Note that FOP does not currently support "tagged PDF" or PDF/A-1a. Support for PDF/A-1b and PDF/X has recently been added, however.

2.1. Fonts

PDF has a set of fonts that are always available to all PDF viewers; to quote from the PDF Specification: *"PDF prescribes a set of 14 standard fonts that can be used without prior definition. These include four faces each of three Latin text typefaces (Courier, Helvetica, and Times), as well as two symbolic fonts (Symbol and ITC Zapf Dingbats). These fonts, or suitable substitute fonts with the same metrics, are guaranteed to be available in all PDF viewer applications."*

2.2. Post-processing

FOP does not currently support several desirable PDF features: watermarks and signatures. One workaround is to use Adobe Acrobat (the full version, not the Reader) to process the file manually or with scripting that it supports.

Another popular post-processing tool is iText, which has tools for adding security features, document properties, watermarks, and many other features to PDF files.

> **Warning:**
> Caveat: iText may swallow PDF bookmarks. But Jens Stavnstrup tells us that this doesn't happen if you use iText's PDFStamper.

Here is some sample code that uses iText to encrypt a FOP-generated PDF. (Note that FOP now supports PDF encryption. However the principles for using iText for other PDF features are similar.)

```
public static void main(String args[]) {
  try {
    ByteArrayOutputStream fopout = new ByteArrayOutputStream();
    FileOutputStream outfile = new FileOutputStream(args[2]);
    FopFactory fopFactory = FopFactory.newInstance();
    Fop fop = fopFactory.newFop(MimeConstants.MIME_PDF, fopout);

    Transformer transformer = TransformerFactory.newInstance().newTransformer(
        new StreamSource(new File(args[1])));
    transformer.transform(new StreamSource(new File(args[0])),
        new SAXResult(fop.getDefaultHandler()));
    PdfReader reader = new PdfReader(fopout.toByteArray());
    int n = reader.getNumberOfPages();
```

```
    Document document = new Document(reader.getPageSizeWithRotation(1));
    PdfWriter writer = PdfWriter.getInstance(document, outfile);
    writer.setEncryption(PdfWriter.STRENGTH40BITS, "pdf", null,
      PdfWriter.AllowCopy);
    document.open();
    PdfContentByte cb = writer.getDirectContent();
    PdfImportedPage page;
    int rotation;
    int i = 0;
    while (i < n) {
      i++;
      document.setPageSize(reader.getPageSizeWithRotation(i));
      document.newPage();
      page = writer.getImportedPage(reader, i);
      rotation = reader.getPageRotation(i);
      if (rotation == 90 || rotation == 270) {
        cb.addTemplate(page, 0, -1f, 1f, 0, 0,
        reader.getPageSizeWithRotation(i).height());
      } else {
        cb.addTemplate(page, 1f, 0, 0, 1f, 0, 0);
      }
      System.out.println("Processed page " + i);
    }
    document.close();
  } catch( Exception e) {
    e.printStackTrace();
  }
}
```

Check the iText tutorial and documentation for setting access flags, password, encryption strength and other parameters.

2.3. Watermarks

In addition to the PDF Post-processing options, consider the following workarounds:

- Use a background image for the body region.
- (submitted by Trevor Campbell) Place an image in a region that overlaps the flowing text. For example, make region-before large enough to contain your image. Then include a block (if necessary, use an absolutely positioned block-container) containing the watermark image in the static-content for the region-before. Note that the image will be drawn on top of the normal content.

3. PostScript

The PostScript renderer has been brought up to a similar quality as the PDF renderer, but may still be missing certain features. It provides good support for most text and layout. Images and SVG are not fully supported, yet. Currently, the PostScript renderer generates PostScript Level 3 with most DSC comments. Actually, the only Level 3 features used are the FlateDecode and DCTDecode filter (the latter is used for 1:1 embedding of JPEG images), everything else is Level 2.

3.1. Configuration

The PostScript renderer configuration currently allows the following settings:

```
<renderer mime="application/postscript">
  <auto-rotate-landscape>false</auto-rotate-landscape>
  <language-level>3</language-level>
  <optimize-resources>false</optimize-resources>
  <safe-set-page-device>false</safe-set-page-device>
  <dsc-compliant>true</dsc-compliant>
</renderer>
```

The default value for the "auto-rotate-landscape" setting is "false". Setting it to "true" will automatically rotate landscape pages and will mark them as landscape.

The default value for the "language-level" setting is "3". This setting specifies the PostScript language level which should be used by FOP. Set this to "2" only if you don't have a Level 3 capable interpreter.

The default value for the "optimize-resources" setting is "false". Setting it to "true" will produce the PostScript file in two steps. A temporary file will be written first which will then be processed to add only the fonts which were really used and images are added to the stream only once as PostScript forms. This will reduce file size but can potentially increase the memory needed in the interpreter to process.

The default value for the "safe-set-page-device" setting is "false". Setting it to "true" will cause the renderer to invoke a postscript macro which guards against the possibility of invalid/unsupported postscript key/values being issued to the implementing postscript page device.

The default value for the "dsc-compliant" setting is "true". Setting it to "false" will break DSC compliance by minimizing the number of setpagedevice calls in the postscript document output. This feature may be useful when unwanted blank pages are experienced in your postscript output. This problem is caused by the particular postscript implementation issuing unwanted postscript subsystem initgraphics/erasepage calls on each setpagedevice call.

3.2. Limitations

- Images and SVG may not be displayed correctly. SVG support is far from being complete. No image transparency is available.
- Only Type 1 fonts are supported.
- Multibyte characters are not supported.
- PPD support is still missing.

4. PCL

This format is for the Hewlett-Packard PCL printers and other printers supporting PCL. It should produce output as close to identical as possible to the printed output of the PDFRenderer within the limitations of the renderer, and output device.

The output created by the PCLRenderer is generic PCL 5, HP GL/2 and PJL. This should allow any device fully supporting PCL 5 to be able to print the output generated by the PCLRenderer. PJL is used to control the print job and switch to the PCL language. PCL 5 is used for text, raster graphics and rectangular fill graphics. HP GL/2 is used for more complex painting operations. Certain painting operations are done off-screen and rendered to PCL as bitmaps because of limitations in PCL 5.

4.1. References

- [WikiPedia entry on PCL](#)
- [Technical reference documents on PCL from Hewlett-Packard](#)

4.2. Limitations

- Text or graphics outside the left or top of the printable area are not rendered properly. This is a limitation of PCL, not FOP. In general, things that should print to the left of the printable area are shifted to the right so that they start at the left edge of the printable area.
- The Helvetica and Times fonts are not well supported among PCL printers so Helvetica is mapped to Arial and Times is mapped to Times New. This is done in the PCLRenderer, no changes are required in the FO's. The metrics and appearance for Helvetica/Arial and Times/Times New are nearly identical, so this has not been a problem so far.
- For the non-symbol fonts, the ISO 8859-1 symbol set is used (PCL set "0N").
- All fonts available to the Java2D subsystem are usable. The texts are painted as bitmap much like the Windows PCL drivers do.
- Multibyte characters are not supported.
- At the moment, only monochrome output is supported. PCL5c color extensions will only be implemented on demand. Color and grayscale images are converted to monochrome bitmaps (1-bit). Dithering only occurs if the JAI image library is available.
- Images are scaled up to the next resolution level supported by PCL (75, 100, 150, 200, 300, 600 dpi). For color and grayscale images an even higher PCL resolution is selected to give the dithering algorithm a chance to improve the bitmap quality.
- Currently, there's no support for clipping and image transparency, largely because PCL 5 has certain limitations.

4.3. Configuration

The PCL renderer configuration currently allows the following settings:

```
<renderer mime="application/vnd.hp-PCL">
  <rendering>quality</rendering>
  <text-rendering>bitmap</text-rendering>
  <disable-pjl>false</disable-pjl>
</renderer>
```

The default value for the "rendering" setting is "speed" which causes borders to be painted as plain rectangles. In this mode, no special borders (dotted, dashed etc.) are available. If you want support for all border modes, set the value to "quality" as indicated above. This will cause the borders to be painted as bitmaps.

The default value for the "text-rendering" setting is "auto" which paints the base fonts using PCL fonts. Non-base fonts are painted as bitmaps through Java2D. If the mix of painting methods results in unwelcome output, you can set this to "bitmap" which causes all text to be rendered as bitmaps.

The default value for the "disable-pjl" setting is "false". This means that the PCL renderer usually generates PJL commands before and after the document in order to switch a printer into PCL language.

7

PJL commands can be disabled if you set this value to "true".

You can control the output resolution for the PCL using the "target resolution" setting on the FOUserAgent. The actual value will be rounded up to the next supported PCL resolution. Currently, only 300 and 600 dpi are supported which should be enough for most use cases. Note that this setting directly affects the size of the output file and the print quality.

4.4. Extensions

The PCL Renderer supports some PCL specific extensions which can be embedded into the input FO document. To use the extensions the appropriate namespace must be declared in the fo:root element like this:

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
         xmlns:pcl="http://xmlgraphics.apache.org/fop/extensions/pcl">
```

4.4.1. Page Source (Tray selection)

The page-source extension attribute on fo:simple-page-master allows to select the paper tray the sheet for a particular simple-page-master is to be taken from. Example:

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="simple" pcl:paper-source="2">
    ...
  </fo:simple-page-master>
</fo:layout-master-set>
```

Note: the tray number is a positive integer and the value depends on the target printer. Not all PCL printers support the same paper trays. Usually, "1" is the default tray, "2" is the manual paper feed, "3" is the manual envelope feed, "4" is the "lower" tray and "7" is "auto-select". Consult the technical reference for your printer for all available values.

5. AFP

> **Warning:**
> The AFP Renderer is a new addition (27-Apr-2006) to the sandbox and as such not yet fully tested or feature complete.

The FOP AFP Renderer deals with creating documents conforming to the IBM AFP document architecture also refered to as MO:DCA (Mixed Object Document Content Architecture).

5.1. References

- AFP (Advanced Function Presentation)
- AFP Resources on the FOP WIKI

5.2. Limitations

This list is most likely badly incomplete.

- Clipping of text and graphics is not supported.
- Only IBM outline and raster fonts and to a limited extend the original fonts built into FOP are supported. Support for TrueType fonts may be added later.

5.3. Configuration

5.3.1. Fonts

The AFP Renderer requires special configuration particularly related to fonts. AFP Render configuration is done through the normal FOP configuration file. The MIME type for the AFP Renderer is application/x-afp which means the AFP Renderer section in the FOP configuration file looks like:

```
<renderer mime="application/x-afp">
  <!-- AFP Renderer -->
  ...
</renderer>
```

There are 3 font configuration variants supported:

1. IBM Raster fonts
2. IBM Outline fonts
3. FOP built-in Base14 fonts

A typical raster font configuration looks like:

```
      <!-- This is an example of mapping actual IBM raster fonts / code pages to a
FOP font -->
      <font>
      <!-- The afp-font element defines the IBM code page, the matching Java
encoding and the
          path to the font -->
      <afp-font type="raster" codepage="T1V10500" encoding="Cp500"
path="fonts/ibm">
        <!-- For a raster font a separate element for each font size is required
providing
            the font size and the corresponding IBM Character set name -->
        <afp-raster-font size="7" characterset="C0N20070"/>
        <afp-raster-font size="8" characterset="C0N20080"/>
        <afp-raster-font size="10" characterset="C0N20000"/>
        <afp-raster-font size="11" characterset="C0N200A0"/>
        <afp-raster-font size="12" characterset="C0N200B0"/>
        <afp-raster-font size="14" characterset="C0N200D0"/>
        <afp-raster-font size="16" characterset="C0N200F0"/>
        <afp-raster-font size="18" characterset="C0N200H0"/>
        <afp-raster-font size="20" characterset="C0N200J0"/>
        <afp-raster-font size="24" characterset="C0N200N0"/>
        <afp-raster-font size="30" characterset="C0N200T0"/>
        <afp-raster-font size="36" characterset="C0N200Z0"/>
      </afp-font>
      <!-- These are the usual FOP font triplets as they apply to this font -->
      <font-triplet name="serif" style="normal" weight="normal"/>
      <font-triplet name="Times" style="normal" weight="normal"/>
      <font-triplet name="Times-Roman" style="normal" weight="normal"/>
      <font-triplet name="TimesNewRoman" style="normal" weight="normal"/>
    </font>
```

An outline font configuration is simpler as the individual font size entries are not required. However, the characterset definition is now required within the afp-font element.

```
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZH200  "
            path="fonts/ibm" />
        <font-triplet name="sans-serif" style="normal" weight="normal"/>
        <font-triplet name="Helvetica" style="normal" weight="normal"/>
        <font-triplet name="any" style="normal" weight="normal"/>
        </font>
```

Experimentation has shown that the font metrics for the FOP built-in Base14 fonts are actually very similar to some of the IBM outline and raster fonts. In cases were the IBM font files are not available the path attribute in the afp-font element can be replaced by a base14-font attribute giving the name of the matching Base14 font. In this case the AFP Renderer will take the font metrics from the built-in font.

```
        <!-- The following are examples of defining outline fonts based on FOP
built-in
            font metrics for the Adobe Base14 fonts -->
        <!-- sans-serif fonts based on Helvetica -->
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZH200  "
            base14-font="Helvetica" />
        <font-triplet name="sans-serif" style="normal" weight="normal"/>
        <font-triplet name="Helvetica" style="normal" weight="normal"/>
        <font-triplet name="any" style="normal" weight="normal"/>
        </font>
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZH300  "
            base14-font="HelveticaOblique" />
        <font-triplet name="sans-serif" style="italic" weight="normal"/>
        <font-triplet name="Helvetica" style="italic" weight="normal"/>
        <font-triplet name="any" style="italic" weight="normal"/>
        </font>
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZH400  "
            base14-font="HelveticaBold" />
        <font-triplet name="sans-serif" style="normal" weight="bold"/>
        <font-triplet name="Helvetica" style="normal" weight="bold"/>
        <font-triplet name="any" style="normal" weight="bold"/>
        </font>
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZH500  "
            base14-font="HelveticaBoldOblique" />
        <font-triplet name="sans-serif" style="italic" weight="bold"/>
        <font-triplet name="Helvetica" style="italic" weight="bold"/>
        <font-triplet name="any" style="italic" weight="bold"/>
        </font>

        <!-- serif fonts based on Times Roman -->
        <font>
        <afp-font type="outline" codepage="T1V10500" encoding="Cp500"
characterset="CZN200  "
            base14-font="TimesRoman" />
        <font-triplet name="serif" style="normal" weight="normal"/>
        <font-triplet name="Times" style="normal" weight="normal"/>
```

```
         <font-triplet name="Times-Roman" style="normal" weight="normal"/>
      </font>

      <!-- The following are examples of defining raster fonts based on FOP built-in
           font metrics for the Adobe Base14 fonts -->
      <!-- monospaced fonts based on Courier -->
      <font>
        <afp-font type="raster" codepage="T1V10500" encoding="Cp500">
          <afp-raster-font size="7" characterset="C0420070" base14-font="Courier"/>
          <afp-raster-font size="8" characterset="C0420080" base14-font="Courier"/>
          <afp-raster-font size="10" characterset="C0420000" base14-font="Courier"/>
          <afp-raster-font size="12" characterset="C04200B0" base14-font="Courier"/>
          <afp-raster-font size="14" characterset="C04200D0" base14-font="Courier"/>
          <afp-raster-font size="20" characterset="C04200J0" base14-font="Courier"/>
        </afp-font>
        <font-triplet name="monospace" style="normal" weight="normal"/>
        <font-triplet name="Courier" style="normal" weight="normal"/>
      </font>
      <font>
        <afp-font type="raster" codepage="T1V10500" encoding="Cp500">
          <afp-raster-font size="7" characterset="C0440070"
base14-font="CourierBold"/>
          <afp-raster-font size="8" characterset="C0440080"
base14-font="CourierBold"/>
          <afp-raster-font size="10" characterset="C0440000"
base14-font="CourierBold"/>
          <afp-raster-font size="12" characterset="C04400B0"
base14-font="CourierBold"/>
          <afp-raster-font size="14" characterset="C04400D0"
base14-font="CourierBold"/>
          <afp-raster-font size="20" characterset="C04400J0"
base14-font="CourierBold"/>
        </afp-font>
        <font-triplet name="monospace" style="normal" weight="bold"/>
        <font-triplet name="Courier" style="normal" weight="bold"/>
      </font>
```

5.3.2. Output Resolution

By default the AFP Renderer creates output with a resolution of 240 dpi. This can be overridden by the <renderer-resolution/> configuration element. Example:

```
    <renderer-resolution>240</renderer-resolution>
```

5.3.3. Images

By default the AFP Renderer converts all images to 8 bit grey level. This can be overridden by the <images> configuration element. Example:

```
    <images mode="color" />
```
This will put images as RGB images into the AFP output stream. The default setting is:

```
    <images mode="b+w" bits-per-pixel="8" />
```
Only the values "color" and "b+w" are allowed for the mode attribute. The bits-per-pixel attribute is ignored if mode is "color". For "b+w" mode is must be 1, 4, or 8.

11

5.4. Extensions

The AFP Renderer supports some AFP specific extensions which can be embedded into the input fo document. To use the extensions the appropriate namespace must be declared in the fo:root element like this:

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
         xmlns:afp="http://xmlgraphics.apache.org/fop/extensions/afp">
```

5.4.1. Page Overlay Extension

The include-page-overlay extension element allows to define on a per simple-page-master basis a page overlay resource. Example:

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="simple">
    <afp:include-page-overlay name="O1SAMP1 " />
    ...
  </fo:simple-page-master>
</fo:layout-master-set>
```

The mandatory name attribute must refer to an 8 character (space padded) resource name that must be known in the AFP processing environment.

5.4.2. Page Segment Extension

The include-page-segment extension element allows to define resource substitution for fo:external-graphics elements. Example:

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:afp="http://xmlgraphics.apache.org/fop/extensions/afp">
   <fo:layout-master-set>
     <fo:simple-page-master master-name="simple">
       <afp:include-page-segment name="S1ISLOGO"
src="../../resources/images/bgimg300dpi.jpg" />
       <fo:region-body/>
     </fo:simple-page-master>
   </fo:layout-master-set>
```

The include-page-segment extension element can only occur within a simple-page-master. Multiple include-page-segment extension elements within a simple-page-master are allowed. The mandatory name attribute must refer to an 8 character (space padded) resource name that must be known in the AFP processing environment. The value of the mandatory src attribute is compared against the value of the src attribute in fo:external-graphic elements and if it is identical (string matching is used) in the generated AFP the external graphic is replaced by a reference to the given resource.

5.4.3. Tag Logical Element Extension

The tag-logical-element extension element allows to injects TLEs into the AFP output stream. Example:

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
```

```
     xmlns:afp="http://xmlgraphics.apache.org/fop/extensions/afp">
      <fo:layout-master-set>
        <fo:simple-page-master master-name="simple">
          <afp:tag-logical-element name="The TLE Name" value="The TLE Value" />
          <fo:region-body/>
        </fo:simple-page-master>
      </fo:layout-master-set>
```

The tag-logical-element extension element can only occur within a simple-page-master. Multiple tag-logical-element extension elements within a simple-page-master are allowed. The name and value attributes are mandatory.

5.4.4. No Operation Extension

The no-operation extension provides the ability to carry up to 32K of comments or any other type of unarchitected data into the AFP output stream. Example:

```
    <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"
     xmlns:afp="http://xmlgraphics.apache.org/fop/extensions/afp">
      <fo:layout-master-set>
        <fo:simple-page-master master-name="simple">
          <afp:no-operation name="My NOP">insert up to 32k of character data
here!</afp:no-operation>
        </fo:simple-page-master>
      </fo:layout-master-set>
```

The no-operation extension element can only occur within a simple-page-master. Multiple no-operation extension elements within a simple-page-master are allowed. The name attribute is mandatory.

6. RTF

JFOR, an open source XSL-FO to RTF converter has been integrated into Apache FOP. This will create an RTF (rich text format) document that will attempt to contain as much information from the XSL-FO document as possible. It should be noted that is not possible (due to RTF's limitations) to map all XSL-FO features to RTF. For complex documents, the RTF output will never reach the feature level from PDF, for example. Thus, using RTF output is only recommended for simple documents such as letters.

The RTF output follows Microsoft's RTF specifications and produces best results on Microsoft Word.

> **Note:**
> RTF output is currently unmaintained and lacks many features compared to other output formats. Using other editable formats like Open Document Format, instead of producing XSL-FO then RTF through FOP, might give better results.

These are some known restrictions compared to other supported output formats (not a complete list):

- Not supported/implemented:
  - break-before/after (supported by the RTF library but not tied into the RTFHandler)
  - fo:page-number-citation-last
  - keeps (supported by the RTF library but not tied into the RTFHandler)
  - region-start/end (RTF limitation)

- multiple columns
- Only a single page-master is supported
- Not all variations of fo:leader are supported (RTF limitation)
- percentages are not supported everywhere

7. XML (Area Tree XML)

This is primarily for testing and verification. The XML created is simply a representation of the internal area tree put into XML. We use that to verify the functionality of FOP's layout engine.

The other use case of the Area Tree XML is as FOP's "intermediate format". More information on that can be found on the page dedicated to the Intermediate Format.

8. Java2D/AWT

The Java2DRenderer provides the basic functionality for all Java2D-based output formats (AWT viewer, direct print, PNG, TIFF).

The AWT viewer shows a window with the pages displayed inside a Java graphic. It displays one page at a time. The fonts used for the formatting and viewing depend on the fonts available to your JRE.

9. Print

It is possible to directly print the document from the command line. This is done with the same code that renders to the Java2D/AWT renderer.

9.1. Known issues

If you run into the problem that the printed output is incomplete on Windows: this often happens to users printing to a PCL printer. There seems to be an incompatibility between Java and certain PCL printer drivers on Windows. Since most network-enabled laser printers support PostScript, try switching to the PostScript printer driver for that printer model.

10. Bitmap (TIFF/PNG)

It is possible to directly create bitmap images from the individual pages generated by the layout engine. This is done with the same code that renders to the Java2D/AWT renderer.

Currently, two output formats are supported: PNG and TIFF. TIFF produces one file with multiple pages, while PNG output produces one file per page. The quality of the bitmap depends on the target resolution setting on the FOUserAgent.

10.1. Configuration

14

The TIFF and PNG renderer configuration currently allows the following settings:

```
<renderer mime="image/png">
  <transparent-page-background>true</transparent-page-background>
  <fonts><!-- described elsewhere --></fonts>
</renderer>
```

The default value for the "transparent-page-background" setting is "false" which paints an opaque, white background for the whole image. If you set this to true, no such background will be painted and you will get a transparent image if an alpha channel is available in the output format.

10.2. TIFF-specific Configuration

In addition to the above values the TIFF renderer configuration allows some additional settings:

```
<renderer mime="image/tiff">
  <transparent-page-background>true</transparent-page-background>
  <compression>CCITT T.6</compression>
  <fonts><!-- described elsewhere --></fonts>
</renderer>
```

The default value for the "compression" setting is "PackBits" which which is a widely supported RLE compression scheme for TIFF. The set of compression names to be used here matches the set that the Image I/O API uses. Note that not all compression schemes may be available during runtime. This depends on the actual codecs being available. Here is a list of possible values:

- NONE (no compression)
- PackBits (RLE, run-length encoding)
- JPEG
- Deflate
- LZW
- ZLib
- CCITT T.4 (Fax Group 3)
- CCITT T.6 (Fax Group 4)

> **Note:**
> If you want to use CCITT compression, please make sure you've got a J2SE 1.4 or later and Java Advanced Imaging Image I/O Tools in your classpath. The Sun JRE doesn't come with a TIFF codec built in, so it has to be added separately. The internal TIFF codec from XML Graphics Commons only supports PackBits, Deflate and JPEG compression for writing.

11. TXT

The text renderer produces plain ASCII text output that attempts to match the output of the PDFRenderer as closely as possible. This was originally developed to accommodate an archive system that could only accept plain text files, and is primarily useful for getting a quick-and-dirty view of the document text. The renderer is very limited, so do not be surprised if it gives unsatisfactory results.

The Text renderer works with a fixed size page buffer. The size of this buffer is controlled with the textCPI and textLPI public variables. The textCPI is the effective horizontal characters per inch to use. The textLPI is the vertical lines per inch to use. From these values and the page width and height the

15

size of the buffer is calculated. The formatting objects to be rendered are then mapped to this grid. Graphic elements (lines, borders, etc) are assigned a lower priority than text, so text will overwrite any graphic element representations.

Because FOP lays the text onto a grid during layout, there are frequently extra or missing spaces between characters and lines, which is generally unsatisfactory. Users have reported that the optimal settings to avoid such spacing problems are:

- font-family="Courier"
- font-size="7.3pt"
- line-height="10.5pt"

12. Output Formats in the Sandbox

Due to the state of certain renderers we moved some of them to a "sandbox" area until they are ready for more serious use. The renderers and FOEventHandlers in the sandbox can be found under src/sandbox and are compiled into build/fop-sandbox.jar during the main build. The output formats in the sandbox are marked as such below.

12.1. MIF

> **Warning:**
> The MIF handler is in the sandbox and not yet functional in FOP Trunk!!! Please help us ressurrect this feature.

This format is the Maker Interchange Format which is used by Adobe Framemaker.

12.2. SVG

> **Warning:**
> The SVG renderer is in the sandbox and may not work as expected in FOP Trunk!!! Please help us improve this feature.

This format creates an SVG document that has links between the pages. This is primarily for slides and creating svg images of pages. Large documents will create SVG files that are far too large for an SVG viewer to handle. Since FO documents usually have text the SVG document will have a large number of text elements. The font information for the text is obtained from the JVM in the same way as for the AWT viewer. If the SVG is viewed on a system where the fonts are different, such as another platform, then the page may look wrong.

13. Wish list

Apache FOP is easily extensible and allows you to add new output formats to enhance FOP's functionality. There's a number of output formats which are on our wish list. We're looking for volunteers to help us implement them.

16

- [ODF (Open Document Format)](): The standardized successor to OpenOffice's file format.