Table of Contents

Journal feedback: suggestions for articles, questions, etc.

PDF and PostScript: a Comparison

Acumen PDF Classes

Acumen Training teaches two courses on the PDF file format: *PDF 1: File Content & Structure* and *PDF 2: Advanced File Content.*

Details of the courses' contents and schedule are available on the Acumen Training website.

A common misconception among students coming into my *PDF 1* class is that PDF is a direct derivative of PostScript and, in particular, that you should be able to embed a PDF file directly into a PostScript file, using it in the same manner as an EPS file.

PDF is a conceptual descendant of PostScript; its approach to drawing on the page is nearly identical to PostScript's and there is an exact equivalent in PDF for most of PostScript's drawing commands: the PostScript *moveto, lineto,* and *curveto* operators are matched by the PDF *m, l,* and *c* operators.

However, the differences between PDF and PostScript are greater than the similarities. In this issue, we shall examine those differences. This discussion will be necessarily brief; you can *(should! must!)* take the *PDF 1* and *PDF 2* classes for full details.

Fundamentals

Let's start by comparing the basic conceptual and structural characteristics of PostScript and PDF.

Programming language

Firstly, and perhaps most importantly, PostScript is a full programming language, PDF is not.

As you know, PostScript has all the features necessary to any programming language: procedures and variables, loops, conditional execution, etc., etc. Anything you can do in, say, C you can do in PostScript (though the code will look decidedly different).

PDF, by contrast, is not a programming language. In fact, strictly speaking, PDF is not a "language." The "code" within a PDF file is not in any sense executable, but rather is text that a PDF viewer reads and then constructs a page. This sometimes makes PDF difficult to discuss to in class, because it is too easy to think of, for example, the *obj* and *endobj* keywords as commands that together create a PDF object. They are not; *obj* and *endobj* simply delimit a section of the PDF file that describes an object.

As a programming language, PostScript is an incredibly powerful tool for constructing pages; in principle, there is no such thing as a page you cannot construct in PostScript one way or another. With PDF, you are limited to the commands and capabilities native to the file format; anything PDF cannot do, it simply cannot do. This has made it important for Adobe to make the set of PDF keywords extremely rich.

On the other hand, PDF files are much easier to examine and debug. PostScript output is typically an intertwisted mass of procedures that call other procedures that call other procedures to an indefinite depth; untangling those procedures when you are tracking down a bug can drive you quite crazy. PDF files can contain only native PDF commands. Regardless of a document's complexity, a PDF file will contain nothing that is not documented (somewhere) in the PDF reference manual. This makes PDF files relatively easy to debug, once you know the file format.

PDF is not a stream

A PostScript file contains a single stream of executable PostScript code. The PostScript interpreter consumes this stream sequentially, executing the PostScript program as it goes.

A PDF file, on the other hand, consists of a series of discrete PDF *objects,* each representing a single piece of data, such as a font dictionary, an image, or a stream of drawing code. To draw a page, a PDF viewer must traverse the "database" of PDF objects associated with that page in order to find the drawing commands, the data used by images, the details of the color spaces used, the fonts, etc.

If you examine a PDF file with a text editor, you will see many blocks of text bound by the *obj* and *endobj* keywords.

```
2 0 obj
<<   /Type    /Pages
     /Kids    [ 3 0 R  4 0 R ]
     /Count   2
>>
endobj
```

Each of these blocks of text defines a PDF object.

PDF is not sequential

PostScript files are *sequential;* that is, all of page one's stuff comes first in the stream, followed by all of page two's stuff, and so on through the document. This allows a PostScript file to be consumed and interpreted a bit at a time; there is no limit to the size of a stream a PostScript viewer can handle, since it never needs to see more than a single bufferful of code at a time.

PDF files are completely random-access. The objects that make up page one may be located literally *anywhere* in the PDF file. As a result, a PDF viewer must have access to the entire file in order to display any part of the file.

This characteristic of PDF makes life extremely difficult for printer engineers who want to directly consume PDF files as a print stream. A printer that directly renders PDF files must have either a hard disk or a large amount of RAM so it can temporarily store the entire file.

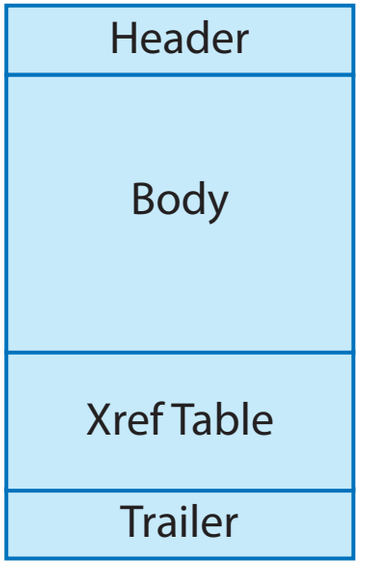
In an effort to ease this problem, the PDF spec defines something called *linearized PDF,* in which each page's objects are contiguous within the PDF file and, furthermore, the first page's contents come first within the file. This doesn't help the printer folks, since the location of the pages *other* that page one within the PDF file is arbitrary; page one could be followed by all of page eight's stuff, followed by all of page 76's stuff, etc.

File layout

PostScript is a very free-form file format. Drawing code must occur in page order procedures and variables must be defined before they are referenced, but there is otherwise no particular structure required by the PostScript language specification. Granted, PostScript files generally conform to a near-universal file organization—definitions first (the "prolog"), followed by drawing commands (the "script")—but this is not part of the PostScript language, but rather a broadly-followed convention (the "Document Structuring Convention," to be precise).

The PDF file format, by contrast, is much more strictly prescribed. A PDF file has four well-defined sections, none of them optional:

- The *header,* which identifies the file as a PDF file and specifies the version of the PDF specification to which the file adheres.
- The *body,* which contains the drawing commands, font definitions, and other PDF "code" that creates the actual document.
- The *cross-reference ("xref") table,* which identifies the location of every object within the PDF file.
- The *trailer,* which contains the initial information required by a viewer to find the cross-reference table and other key items within the PDF file.



These four sections are formally required by the PDF file specification; they are not merely a conventional organization.

Drawing on the Page

PDF Content Streams

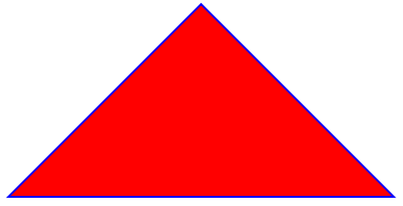
A PostScript file is a stream of executable code that either draws on the page or defines procedures, variables, and other items that are used to draw on the page.

A PDF file, on the other hand, is composed of a series of numbered PDF objects, each providing information needed for the construction of the document; for example, consider this PDF code from a previous page:

```
2 0 obj
<<   /Type    /Pages
     /Kids    [ 3 0 R   4 0 R ]
     /Count   2
>>
endobj
```

This block of text supplies information for an object #2, which in this case is a dictionary. (Note that dictionaries are constructed with double-angle brackets, as in PostScript.)

Drawing commands within the PDF file are encapsulated in an type of PDF object called a *content stream,* as in the following example:



```
10 0 obj
<<
   /Length 240
>>
stream
100 100 m
200 200 l
300 100 l
h
1 0 0 rg
0 0 1 RG
B
endstream
endobj
```

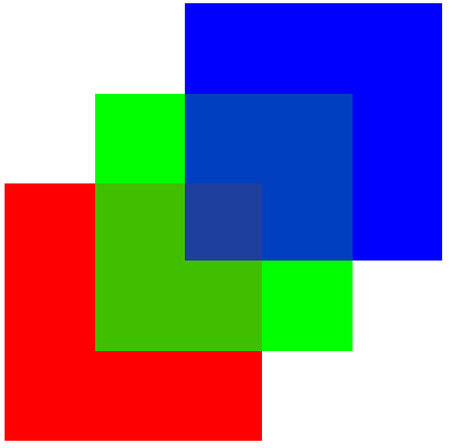
Here, object 10 is a content stream whose component drawing commands paint a triangle on the page.

Graphics

With one important exception, the graphics model implemented by PostScript and PDF are very nearly identical. They use the same User Space to draw on the current page; their conceptual approach to drawing images and line art are identical; they both use a mixture of Type 1, TrueType, and Type 3 fonts to print text.

There are some differences, however.

Transparency The most technically troublesome difference is the fact that PDF supports transparency.



When you paint an object onto the page in PDF and PostScript, they apply the current color, line width, coordinate transformation, and all the other graphic state parameters appropriate to the task. However, PDF applies the current *opacity,* as well. (Actually, it applies something called the current *alpha,* but for the purpose of this discussion, this is the same as the opacity.)

This causes no end of printing trouble in the PostScript world, since PostScript simply has no notion of opacity *at all*. When printing a PDF containing transparency to a PostScript printer, a PDF viewer has three choices:

- Render the page ahead of time (including all the transparent items) and print the entire page as an image.
- Cut up each transparent object into a series of single-color polygons and draw those. (Note that the three overlapping rectangles above could be rendered as seven monochrome polygons. This is impractical for any but the simplest graphics.
- Discard the transparency information completely and simply print the objects as opaque.

Some printers render PDF directly and so can be sent the PDF file as-is. The Jaws PostScript RIP has long done this; others came along later. I believe the current version of most (and perhaps all) PostScript RIPs these days render PDF natively, so printing transparent objects should become less of a problem as new printers replace old ones.

Color commands

Most of the drawing commands in PDF are identical to PostScript equivalents, as shown by the examples at right. There are, however, some significant differences.

PostScript and PDF Drawing Commands

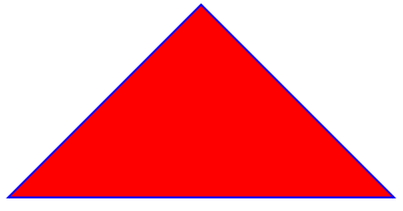
PostScript	*PDF*	*PostScript*	*PDF*
moveto	m	setcolorspace	cs
lineto	l	setcolor	sc
closepath	h	curveto	c
setlinewidth	w	setdash	d
setrgbcolor	rg	show	Tj
setgray	g	selectfont	Tf

For example, PDF differs from PostScript in maintaining two current colors, one for stroking and one for everything else (referred to as the "stroke" and "non-stroke" colors, reasonably enough).

Each PDF color-related operator has two forms, indicated in the PDF code by upper case and lower case names. The upper case version of each operator applies to stroking operations; lower case sets the non-stroke color.

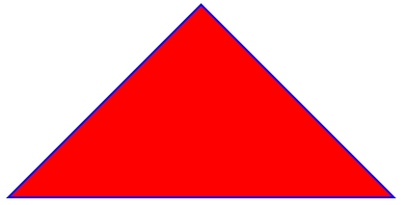
As an example, PostScript code that draws a triangle filled with red and outlined in blue looks like this:

```
100 100 moveto
100 100 rlineto
100 -100 rlineto
closepath
gsave
1 0 0 setrgbcolor fill
grestore
0 0 1 setrgbcolor
stroke
```



Here is the same triangle drawn with PDF code:

```
100 100 m     % Create the path (note that percent signs indicate comments in PDF, too)
200 200 l
300 100 l
h             % This is closepath
1 0 0 rg      % Set the fill color to red
0 0 1 RG      % Set the stroke color to blue
B             % "B" is the fill-then-stroke operator
```



A couple of comments based on the above code:

- PDF doesn't do relative lineto's; all moves, lines, and curves are specified in absolute User Space coordinates.
- Note that we don't need to use the PDF equivalents of *gsave* and *grestore* (which are *q* and *Q*, by the way; go figure) to do this task. Instead, we set the stroke and non-stroke colors and then call the fill-then-stroke operator, *B* (for "Both," I assume).
- The fact is, the PDF *gsave/grestore* equivalents *can't* help us to do the above task because in PDF the current path is not part of the graphics state.
- Note that percent signs indicate comments, exactly as in PostScript.

Text PDF uses the same Type 1 and TrueType fonts that are used by PostScript; it also has an equivalent of the PostScript Type 3 fonts. Otherwise, the PDF approach to printing text is very different from PostScript's. We shall touch on only a few of these differences here.

Modal printing

Most basically, PDF is modal with regard to the printing of text and line art. Whereas in PostScript you may freely intermix drawing and text commands (that is, *moveto* and *lineto* on the one hand and *show* or *widthshow* on the other), in PDF you may at any moment draw either line art or text, not both.

Text mode is bound within a PDF file by the *BT* and *ET* keywords ("Begin Text" and "End Text"). Text drawing commands may appear only between *BT* and *ET* and, furthermore, only text drawing commands may occur within a *BT/ET* pair.

Thus, printing text in PDF entails a piece of PDF code looking like this:

```
BT                                          % Begin Text
/Hv 18 Tf                                   % Select an 18-point "Hv"
100 600 Td                                  % Specify the print location
(If you tell the truth, you don't) Tj       % Print a string
0 -20 Td                                    % Offset the print location
(have to remember anything.) Tj             % Print another string
ET                                          % End Text
```

Again, some notes based on the above code:

- The *Tf* operator is the PDF equivalent of *selectfont;* it sets the current font to a particular font and size. The "font name" is actually the name of a Font resource, which is only mildly similar to PostScript Font resources.
- The *Td* operator specifies the position at which the text should appear on the page. Surprisingly, it is always a relative value, which is why the second call to *Td* in our sample has the *x y* values "0 -20," meaning move over *0* and down *20* points from the start of the previous piece of text. This is because *Tj* actually is doing a coordinate translation, changing the origin of something called *Text Space*.

Text Space & Text Position

PDF's approach to printing text is based on two concepts that don't exist in PostScript:

- *Text Space* is the coordinate system within which text printing takes place. The *BT* operator creates Text Space, which is initially identical to User Space.
- The *Text Position* is the location within Text Space where the next string of text is to be printed. It is conceptually the same as the current point, except that it applies only to text. (The current point is not involved in printing text in PDF.)

The *Td* operator in our example looks very much as though it is simply a *moveto* equivalent, specifying the location of the next text printing operation:

```
100 600 Td
```

However, *Td* actually does two things:

- It translates the Text Space origin by the *x,y* distance specified.
- It moves the text position to the Text Space origin.

Note that this makes *Td* a relative operator; its *x y* arguments specify an offset from the previous *Td*. (This is behavior common to all coordinate transformations in PostScript and PDF.)

A consequence of this is that if you want to draw a line of text immediate beneath the previous line of text (as when printing lines in a paragraph), the *Td* value will specify a simple vertical offset:

```
(If you tell the truth, you don't) Tj
0 -20 Td
(have to remember anything) Tj
0 -20 Td
(- Mark Twain)
```

The above snippet prints three lines of text, each baseline 20 units beneath the previous.

Character encoding

Reencoding fonts is one of those inescapable tasks in PostScript and PDF; it is a vastly simpler task in PDF, however.

To re-encode a font in PostScript, you need to create an entirely new font that is identical to the old one, except that its character encoding matches what you need it to be. This is a mildly irksome process:

```
/MacEncoding            % This is our new Encoding array
[    /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef
     ... Lots of character names here ...
     /tilde /macron /breve /dotaccent /ring /cedilla /hungarumlaut
] def

/Helvetica findfont dup         % The font we're going to re-encode
length dict begin               % Create our new (soon-to-be) font dictionary
currentdict copy pop            % Copy our original font contents into new dict.

/Encoding MacEncoding def       % Insert the new Encoding array

/Times-Mac currentdict definefont pop    % Turn the new dict. into a font dictionary

end                             % Remove the dict. from the dictionary stack
```

PDF, on the other hand, inherently knows about the commonly-used character encodings and you can therefore invoke them by name. A font dictionary in a PDF file may look something like this:

```
20 0 obj
<<  /Type /Font
    /Subtype  /Type1
    /BaseFont /Times-Roman
    /Encoding /MacRomanEncoding
>>
endobj
```

The *Encoding* entry in the above dictionary specifies by name the character encoding to be used with the font; the value of this entry can usefully be one of three names: *MacRomanEncoding, MacExperEncoding,* or *WinAnsiEncoding.*

Note, by the way, that font dictionaries have a different purpose in PDF, compared to PostScript. In PostScript, a font dictionary actually defines the font; it contains all of the character definitions and anything else needed to make the font work. In PDF, a font dictionary *describes* the font; it specifies the character widths and other metrics, the encoding, and some other parameters. The font dictionary in the above snippet is missing most of the usual PDF font information, because Times-Roman is one of PDF's fourteen standard fonts (identical to PostScript's standard fonts, with the addition of Zapf Dingbats).

That's Enough for the Moment

I could go on and on; the differences between PDF and PostScript are too numerous and run too deep for a simple enumeration to be meaningful. What I most want you to take away from this article is the fact that PostScript and PDF, though they share a common ancestry, are *very* different beasts. In particular, you can't simply embed a PDF file in a PostScript stream and use it as an imported graphic.

Trapping in Acrobat 8

One of the important features added to PostScript when Adobe came out with Level 3 was support for "in-RIP" trapping. This feature is vital for people who create printing plates on PostScript devices.

PDF gained trapping support shortly after PostScript Level 3 came out; it is now a very important feature of that file format, as well (to some people, at least). In this issue, we shall discuss what trapping is, see how to turn this feature on, and examine what parameters are available to affect its behavior.

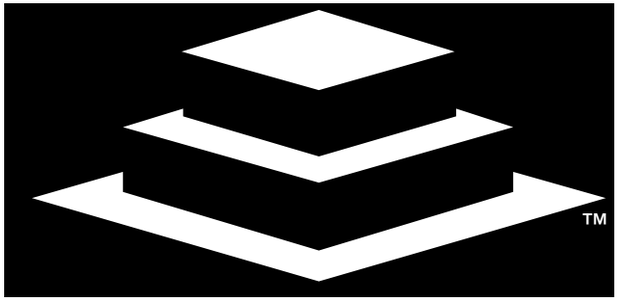
What's "Trapping?"

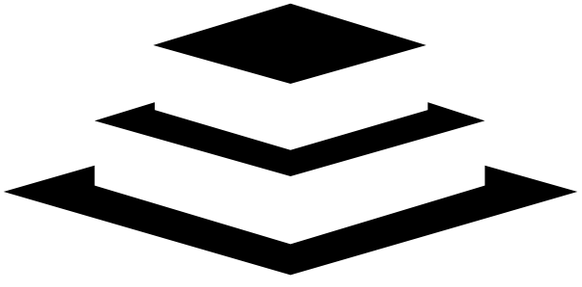
Trapping is a purposeful overlap built into color plates to ensure that color patches abut properly.

Printing Separations

Consider the color graphic at right. On a printing press, this would very likely be printed with two inks: a black ink and a purple ink. Each of these inks would require its own printing plate, one with all the black parts and another with all the purple bits.

The plates would be created on a PostScript (or other) printer as a pair of "black-and-white" pages, as at right; these would be used to create the appropriate printing plates.





Note that the purple plate's graphic exactly fits into the holes in the black plate. When printed on the same sheet of paper, the two separations combine to create our final graphic.

Misaligned Separations

The only problem is this: paper shifts and stretches as it moves through a printing press; as a result, our purple graphic will probably not fit correctly into the holes reserved for it on the black plate.

As a result, we get little flashes of white paper appear where the two colors should abut, as at right.

Hence, Trapping

To solve this problem, we "trap" the plates by building a bit of overlap into the two plates. We make the purple graphic a bit larger or the hole in the black plate a bit smaller than needed.

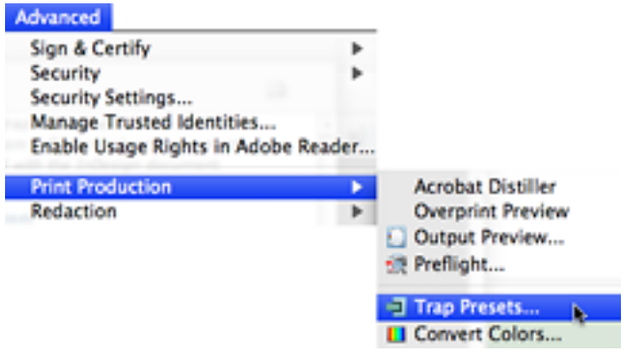
Now, as long as the paper doesn't stretch or shift by more than the amount of overlap, we will no longer see any white where colors should touch. Of course, there *will* be a zone of overlapping ink where the two colors meet, but this is almost always far less distracting to the eye than the white flash.

Trapping in Acrobat

Acrobat's trapping feature is built around the notion of *Trapping Presets.* A trapping preset is a set of parameters that define how trapping should behave on a page. Acrobat ships with a default set of trapping parameters and you may also define your own.

Using a Trapping Preset

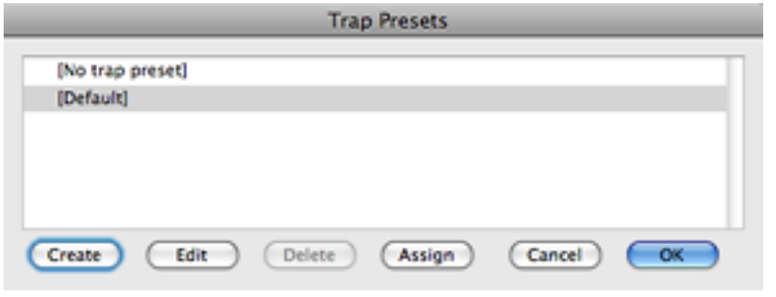
You apply a trapping preset by selecting *Advanced>Print Production>Trapping Presets.* Acrobat presents you with the Trapping Presets dialog box (right middle) from which you can select among the available presets.



To apply trapping to a range of pages, do the following:

1. Select *Trap Presets,* as described above.

   Acrobat displays the Trap Presets dialog box.



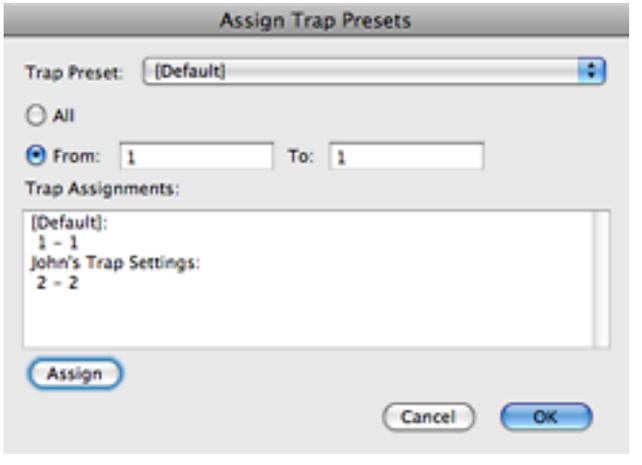
2. Click the *Assign* button.

   Acrobat will present you with the *Assign Trap Presets* dialog box (bottom right).

3. In the pop-up menu, select the preset you want to apply (generally "Default").

4. Use the two radio buttons and two text fields to specify the page range to which the preset should be applied.

   Surprisingly, the large text field that takes up so much of the dialog box's area is not editable; it only reports on the settings you have chosen.



5. Click *Assign* to assign your preset to the pages.

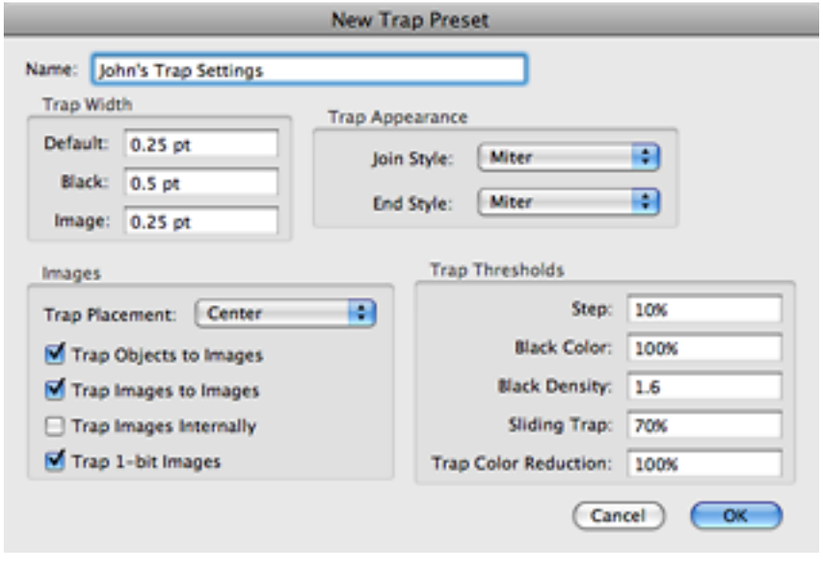
   Repeat steps 4 and 5 to assign more presets to other pages.

6. Click OK and then OK again in the *Trap Presets* dialog box to take you back to your PDF document.

Trapping will now be applied (at print time) to the pages you have specified.

Creating Trap Presets

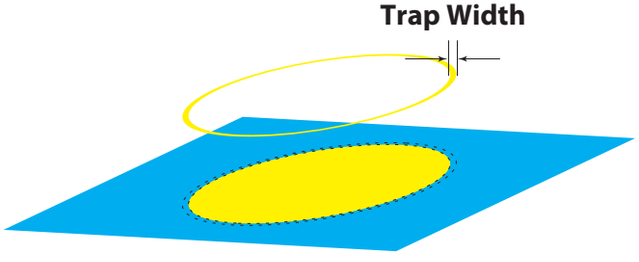
Acrobat's default trap preset is perfectly alright and I have never seen any reason to change its parameters. Of course, I just toy with this stuff, since I don't do professional printing for a living, so my standards may be low.



Those more knowledgeable than me can create custom trapping presets or edit existing presets from within the *Trap Presets* dialog box. Simply click on the *Create* button or select a preset and click the *Edit* button. Either way, Acrobat will present you with the *New Trap Preset* dialog box, as at right.

None of these parameters are hard to understand, but their names are often a bit un-obvious, so let's talk about them briefly.

Trap Width Parameters



"Trap width" refers the amount of overlap that should be built into abutting colors. This should be large enough to mask the largest amount of paper shift that you expect as the page moves through the press.

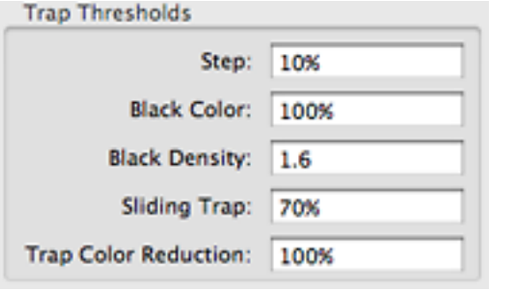
There are three trap width values you may specify:

- *Default* – The overlap to be built into most abutting colors.
- *Black* – The overlap to be used when one of the adjoining colors is black. This value should always be greater than the non-black trap value, I don't know why. (Can anyone tell me?)

- *Image* – The overlap to be applied when trapping line art against an image.

Trap Threshold Parameters

These parameters determine the difference there must be between adjoining colors for Acrobat to apply a trap. It also specifies the details of the trap's nature depending on the two colors involved. The values you may specify are:



- *Step* – The difference there must be between the adjoining colors before Acrobat will apply a trap.

  There is no point creating a trap between to abutting objects if they are of the same or very similar colors. The *Step* value determines how different the two colors must be to provoke a trap. The smaller this number, the more aggressively Acrobat will apply traps. (Also, the slower the page will process.)

- *Black Color* - The shade of gray that Acrobat will consider to be black for trapping purposes; if either of the adjoining colors is a shade of gray darker than this value, Acrobat will apply a black trap (which is thicker than a normal trap, remember).

  What Acrobat is accommodating here is *dot gain,* the inevitable darkening of shades of gray when they are printed. Dot gain comes from various sources, depending on the printing technology being used. For example, halftoned shades of gray darken because the paper wicks up the ink, increasing the size of the halftone dots.

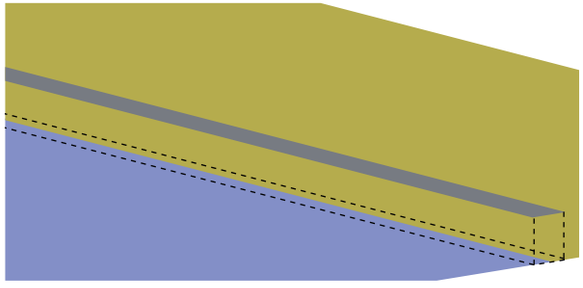
- *Black Density* – This is similar to Black Color, but applies to spot and process colors. It is the *neutral density* that will be considered black for trapping purposes. (Neutral density is a measured value that reflects how visually "dark" a color is; 100% yellow has a very low neutral density; 75% black has a relatively high neutral density.

If either of the trapped inks has a neutral density greater than the Black Density, a black trap will be applied, rather than the thinner, normal trap.

- *Sliding Trap* – This value indicates the point at which Acrobat will change to applying a centerline trap.

  Normally, the trap applied by Acrobat is the same color as the lighter of the abutting colors; in effect, Acrobat extends the lighter ink into the darker.

  However, if the two trapped colors are too similar (as determined by their neutral densities), spreading one into the other produces a trap that is itself a distraction; in this case, Acrobat applies a centerline trap—that is, a thin strip of ink centered on the boundary between the two colors—of a color intermediate between the two trapped colors.

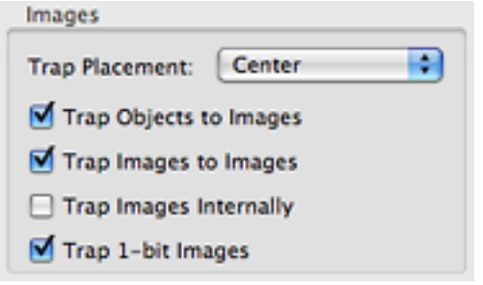


  The *Sliding Trap* value determines how close together two colors must be before Acrobat will switch to applying a centerline trap. The smaller the value, the more similar the two colors must be before Acrobat will apply a centerline trap.

- *Trap Color Reduction* – Trapping delicate colors, such as pastels, can create an ugly dark overlap that is more distracting than the flash of white the trapping is intended to eliminate. In this case, you would like the traps to be lighter than the combined original colors. The *Trap Color Reduction* parameter allows you to specify how much lighter the trap should be than the combined colors.

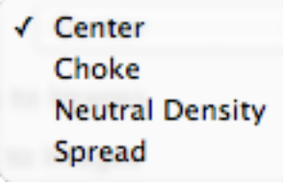
  A value of *0* tells Acrobat that the the trap should have the same neutral density as the darker of the two trapped inks. At about 10%, the trap's neutral density will be the same as the combined inks.

Image Trap Parameters



This set of controls determine whether and how Acrobat traps images against a variety of other graphic objects. The controls consist of a pop-up menu and four self-explanatory checkboxes; the checkboxes are:

- *Trap Objects to Images* – Add trapping to the intersections of images with other graphic objects.
- *Trap Images to Images* – Trap the border between overlapping images.
- *Trap Images Internally* – Trap the border between adjacent pixels within an image. I would expect this to greatly increase the time it takes to process the page.
- *Trap 1-bit Images* – Trap the a monochrome image's painted pixels against adjoining colors. Note that "monochrome" in this case means that the pixels are only black or white; there are no grayscale pixels.



The *Trap Placement* pop-up menu specifies the nature of the trap that should be applied when images are trapped against line art. There are four choices in the menu, presented in a mildly puzzling order:

- *Center* – All traps between images and line art should be centerline traps.
- *Choke* – The trap should consist of the line art extending into the image.
- *Neutral Density* – The image should be treated as line art for trapping purposes. That is, a spread, choke, or centerline trap should be applied, depending upon the neutral densities of the line art and the individual image pixels.
- *Spread* – The image should extend into the line art by whatever the trap width is.

Trap Appearance

A trap is basically a colored line placed onto or adjacent to the border between the two abutting colors. The *Trap Appearance* pop-up menus let you specify how corners and endpoints within the trap should appear. I admit that I am vague as to the benefits of the various choices here.

When Does Trapping Happen?

Trapping is entirely a printing thing and, furthermore, applies mostly to the production of color printing plates. Thus, it will be invisible when viewing a PDF file on screen. It is up to the software printing the PDF file to decide how to apply trapping when creating plates.

When printing to PostScript level 3 printer, the printing software may simply pass the trapping parameters along to the PostScript RIP and let the printer handle the trapping. If printing to a PostScript Level 2 or non-PostScript printer (or even a Level 1 printer, I suppose, but in that case you *really* need to update your printer), the PDF software will have to add the trapping as a series of colored lines (in the case of a PostScript printer) or add the trapping to the bitmap rendering of the PDF pages.

Low-Cost Trapping

Trapping is an inescapable activity in commercial printing. Before PostScript Level 3 came along, it was also a very expensive thing to do, since a graphic designer would have to add the traps to the design "by hand." This was tedious and time consuming.

With PostScript Level 3, it became possible to simply "turn on" trapping and suddenly trapping became (at least in principle) a relatively cheap activity, added as needed by the PostScript printing device. The PDF file format supports that same set of trapping parameters as Postscript and, as with PostScript, has made it possible to add inexpensive trapping to color PDF print jobs.

Schedule of Classes, January – May 2008

Following are the dates of Acumen Training's upcoming PostScript and PDF classes. Clicking on a class name will take you to the description of that class on the Acumen training website. These classes are taught in Orange County, California and on-site at corporate sites world-wide. See the Acumen Training web site for more information.

PDF Courses

PDF 1: File Content and Structure	Jan 21–24		May 5–8
PDF 2: Advanced File Content			
Support Engineers' PDF			May 15–16

PostScript Courses

PostScript Foundations	Jan 14–18		Mar 24–27
Advanced PostScript		Feb 4–7	
Variable Data PostScript			
Troubleshooting PostScript			May 12–14

Course Fee Classes cost $2,000 per student, except for *Troubleshooting PostScript* and *Support Engineers' PDF,* which are $1,500 per student. There is a 10% discount for signing up three or more students. If you have four or more students that need to take a class, it will almost certainly be cheaper to arrange an on-site class.

Contacting John Deubert at Acumen Training

For more information

For class descriptions, on-site arrangements or any other information about Acumen's classes:

Web site: www.acumentraining.com **email:** john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Registering for Classes

To register for an Acumen Training class, contact John any of the following ways:

Register On-line: www.acumentraining.com/register.html

email: john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

On-Site Classes

Information regarding classes on corporate sites is available at www.acumentraining.com/Onsite.html. These courses are taught throughout the world; for additional information on classes outside the United States, go to www.acumentraining.com/OnsitesWorldWide.html.

Back issues

All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

What's New at Acumen Training?

XPS File Content and Structure in July

I've had to push back the release of the XPS class. It's going to be a busy first half of the year for me with a lot of classes and a book due out in May. (There'll be more information on the book in the next *Journal,* once the software is announced.)

Journal Feedback

If you have any comments regarding the *Acumen Journal,* please let me know. In particular, I am looking for three types of information:

Comments on usefulness. Does the Journal provide you with worthwhile information? Was it well written and understandable? Do you like it, hate it? Did it make your endoscopic exam seem a more pleasant memory, somehow?

Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

Please send any comments, questions, or problems to:

john@acumentraining.com

New Trap Preset Dialog Box