Table of Contents

Acumen Training

Compression in Acrobat

In my most recent Acrobat class at Seybold, one student told me, as a preamble to a question, that her graphic artist insists that they can't use any compression in their PDF files because their work is of too high quality to permit it.

This is a common statement among people who work with PDF and displays a misconception of how compression works. Some forms of compression do, indeed, reduce the quality of your images, but others have literally no effect whatsoever on image quality.

In an effort to clear up some of the common misconceptions, let's talk this month about how compression actually works. Next month, we shall see how this applies to Acrobat.

What is Compression?

Compression is one of those magical things. (Maybe that's overstated; I used to be very interested in compression algorithms. As a hobby. This kept my social life simple.) Stated most broadly, compression is a general term method for some method of reducing the size of a set of data while still keeping the data usable.

StuffIt on the Macintosh and PKZIP on Windows are software products that compress the contents of a file. The compressed file is still usable; if you uncompress the file, you can then open the document or run the application, just as before.

What compression does, under the hood, is replace the original data with a description of that data. The goal is to make that description smaller than the original data.

Next Page ->

Lossless vs Lossy Compression

Compression methods fall into two broad classes:

- **Lossless** compression is a general term for any means of compression that does not discard data.
- **Lossy** compression is the term applied to a compression method that does discard data.

Let's discuss these at more length.

Next Page ->

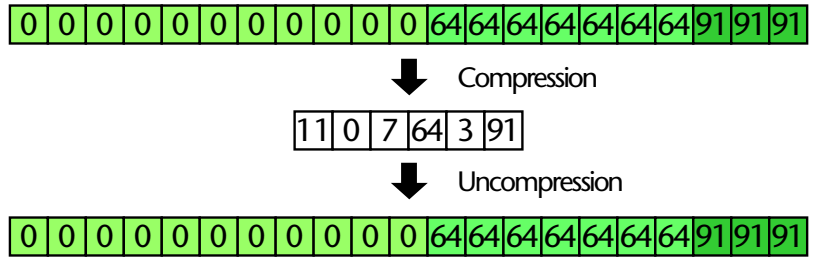
Lossless Compression

Lossless compression has no net effect on your data. The compressed file, when uncompressed again, will be the same, byte for byte, as the original.

To see how this works, consider the data that makes up a scanned image. In most images, there will be many scan lines that contain contiguous bytes that all have the same value; for example, 150 bytes in a row that all have the value 0.

If we wished, we could replace the 150-byte run of identical bytes with two bytes: one that specifies how many bytes are in the run and a second that supplies the byte value. We will have gone from 150 bytes of original data to two bytes of compressed data.

As another example, examine the illustration at right. We start with a scan line consisting of three runs of identical values: zeros, 64's, and 91's.



Upon compression, this 21 bytes of data becomes 6 bytes, as illustrated. Each pair of bytes in the compressed data is the length and value of a run of repeating bytes in the original image.

Uncompression, of course, entails running through the compressed data and creating a new file that reconstructs the set of bytes in the original data, generating 11 zeros, seven 64's, and three 91's.

This kind of compression is called **Run Length** compression; Distiller can apply this to monochrome bitmap images.

Next Page ->

There are a couple of useful facts to note from our Run Length compression example.

Lossless Compression

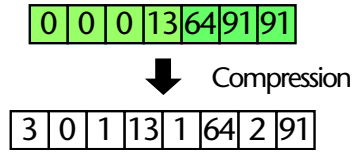
The first point is that, because this is a lossless compression algorithm, the uncompressed data exactly matches the original data. There is absolutely no loss of image quality due to the compression process.

This will always be true of lossless compression methods. ZIP and Run Length are lossless compression methods that are supported in PDF. You may apply ZIP compression to any type of image; you may apply Run Length to monochrome images. They are both absolutely harmless with regard to image content.

Variable Compression

The second point is that the amount of compression you get from a particular algorithm is dependent upon the nature of the data. In our example from the previous page, our original 21 bytes compressed to 6 bytes, a compression ratio of $^{6}/_{21}$ or about 30%.

But if we apply Run Length compression to the set of bytes at right, our compression ratio is $^{8}/_{7}$ or 114%. Our "compressed" data is bigger than the original. This is not good. (As a rule of thumb, you want to avoid compressions over 100%.)



Every compression method has sets of data on which it works well and others on which it does poorly.

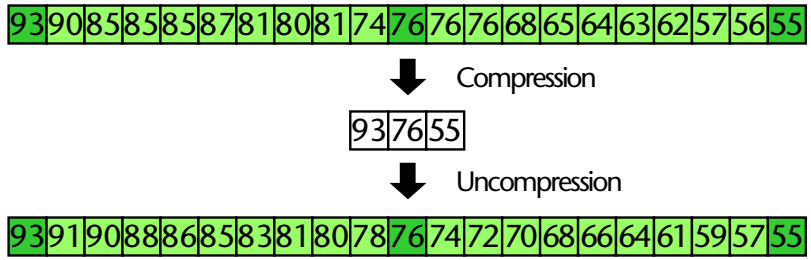
Next Page ->

Lossy Compression

Lossy compression, such as JPEG, works at least in part by discarding data. This tends to work best with complex data, such as color images.

The compression software analyzes the data and discards data that it thinks it can easily reproduce from calculation later. At uncompression time, the values of the missing bytes must be calculated from the data that was retained.

Consider the set of data at right. Our lossy compression software may decide that it needs to retain three of the twenty-one bytes diagrammed here. These are the only bytes that survive into the compressed file; the rest are discarded.



When the data is uncompressed, the uncompression software must interpolate values for the missing bytes from the data that was kept. If you compare the uncompressed bytes with the originals, you would find that, although they are close in value and retain the same general trend, the exact values are different from the originals.

The net result is that the compression method changes the image data. By how much depends on the image being compressed and, in the case of JPEG, how much "quality" you have asked be retained.

Next Page ->

Compression in Distiller

The reason all of this is relevant to Acrobat, of course, is that everything in a PDF file can (and should) be compressed. Lossless compression can be applied to text and line art; images can be compressed using either lossless or lossy methods.

The controls that specify the details will vary among the different sources of PDF (Distiller, Illustrator, PDF Creator, etc.), but they are mostly dictating the same set of options. We'll look how those options appear in Distiller.

Distiller Job Options

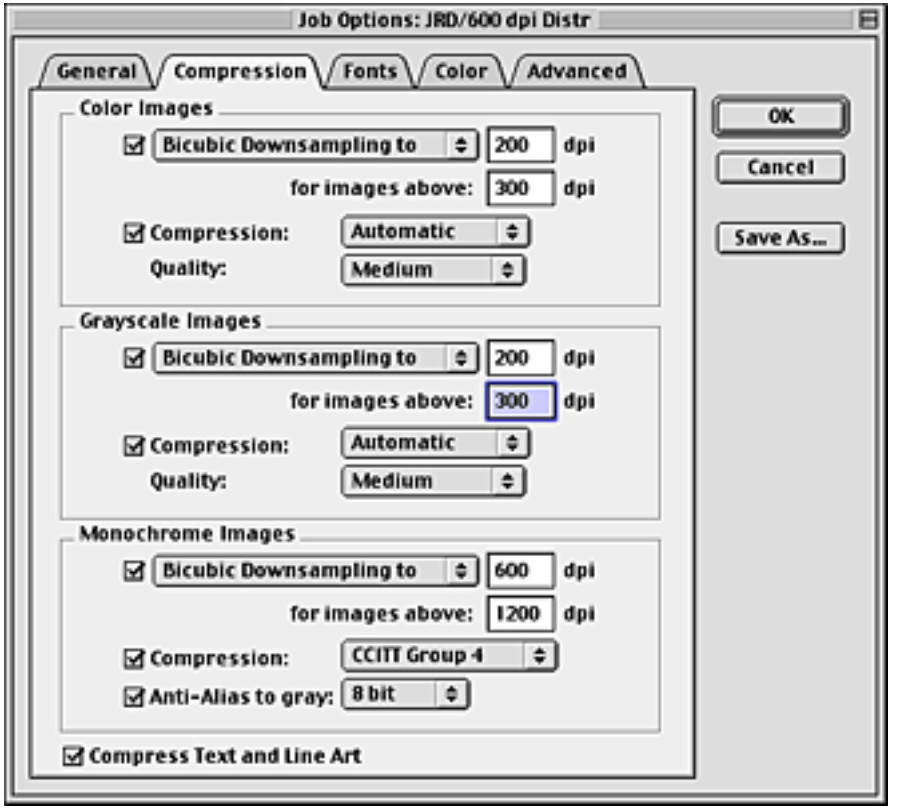
You get to the compression controls in Distiller by selecting *Job Options* from the Settings menu. This yields the five-tab Job Options dialog box (next page); the second tab is the one we want: Compression.

Next Page ->

Compression Job Options in Distiller

Distiller's Job Options dialog box provides a *Compression* panel containing four sets of compression controls:

- A single checkbox that turns on compression for text and line art.
- Three sets of controls for color, grayscale, and monochrome images.



Text and Line Art

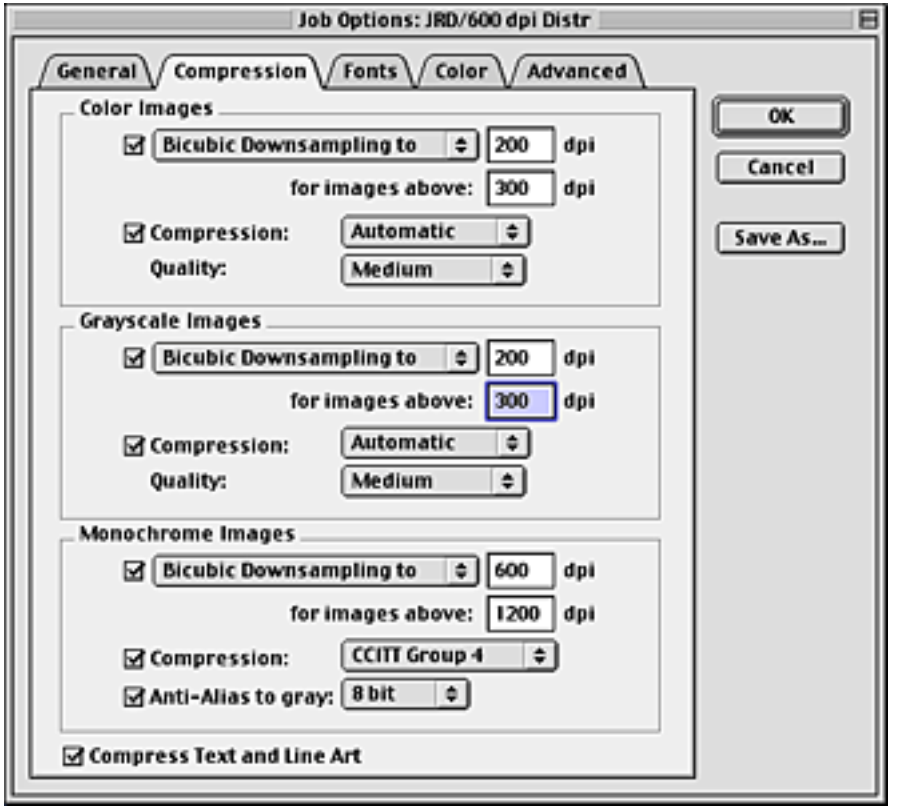
The *Compress Text and Line Art* checkbox applies lossless compression to the text and line art in your PDF file. Since this compression is lossless, it cannot hurt you.

Turn this checkbox on. There is no good reason not to compress text and line art.

Next Page ->

Color and Grayscale Images

The controls for color and grayscale image compression are identical. There are three controls associated with this compression:



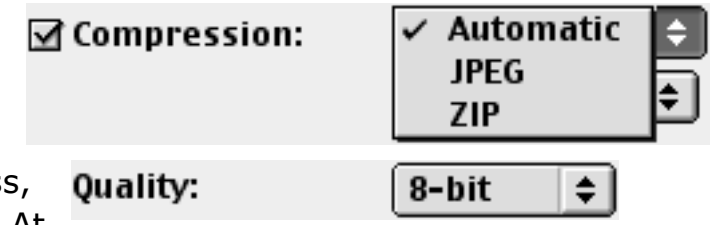
Compression This checkbox turns on the compression of color/grayscale images. Turn it on. You always want to compress your images; the question is: what kind of compression?

Automatic/JPEG/ZIP This pop-up menu lets you specify the kind of compression you want applied to your images. (We'll talk about the choices in a moment.)

Quality This pop-up menu lets you specify how much data should be retained if you are using lossy compression.

Next Page ->

Compression Types



The types of compression that you may apply to color and grayscale images are:

- **ZIP** - This is the most conservative selection. Since zip compression is lossless, this selection will not change your image. At the very least, you should select ZIP compression. (In this case, you should also select 8-bit in the *Quality* pop-up menu; the alternative is 4-bit, which is lossy.

- **JPEG** - This applies lossy JPEG compression to your images. On the plus side, this is a much more effective compression method than zip and will make your images smaller. The problem, of course, is that your images will visibly change due to the compression.

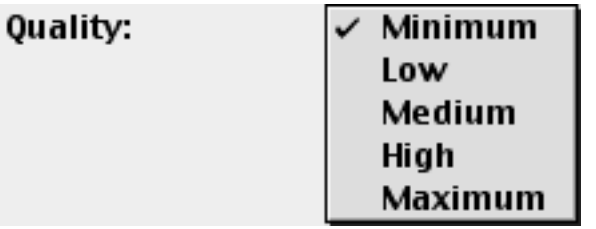
  Most people are far more paranoid about lossy compression than they need to be. Those of you who have taken my Acrobat Essentials class have already been treated to my "JPEG is not Evil" lecture. Alas, there's not room for that here (perhaps in a future Journal article?), but I do suggest you experiment with JPEG compression. Particularly if you are eventually printing on a four-color press, the color change introduced by JPEG is often *much* smaller than that due to the printing device.

- **Automatic** – This attempts to be intelligent about what compression is applied to an image. If the image has a lot of abrupt changes in color (which are affected horribly by JPEG), Distiller will use ZIP compression; lossless and harmless. Otherwise, it will use JPEG.

Next Page ->

JPEG Quality Settings

You can choose how lossy your JPEG compression is going to be. The *Quality* pop-up menu lets you select among a range of "qualities," from Maximum to Minimum. The higher the quality, the more of the original data will be retained by the when the image is compressed.



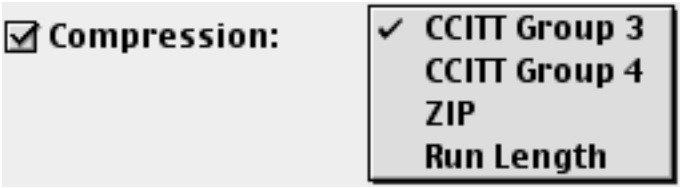
There's not space here to go into details, but the short version is:

- Maximum is acceptable for reasonably high-quality color work. (Pretty much any four-color press will change the final printed image more than does the compression.)
- Medium works well for more forgiving color work and for quick-print jobs.
- Minimum is essential for PDF files viewed on the Web.

We may discuss these in more detail in a future Journal article. (Or you could always take the Acrobat Essentials class.)

Monochrome Image Compression

All of the compression methods available for monochrome images are lossless. CCITT Group 4 is theoretically the most effective of these, but they all do a very good job of compressing monochrome images. Pick whichever you appeals to you the most.



Next Page ->

Downsampling

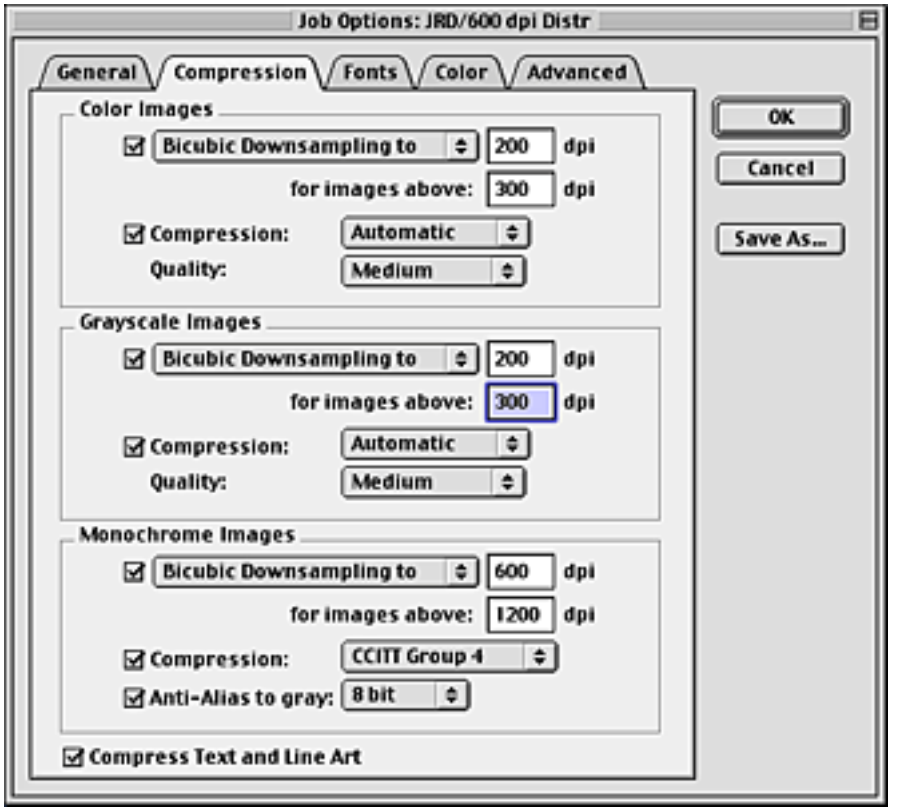
There is another set of controls on the Compression panel that control the **downsampling** of your images to a target resolution.

Properly speaking, this isn't really compression, though it is aimed at reducing the size of images.

Unfortunately, we're out of space and time this month.

Later, perhaps?

Return to Main Menu



Composite Fonts

Last month's article on outline fonts ended with a brief example that combined normal and outline characters into a single composite font, letting you switch between the two styles by placing, in effect, control codes into your *show* string. The PostScript line

```
(We often use \377\001outline\377\000 text.) show
```

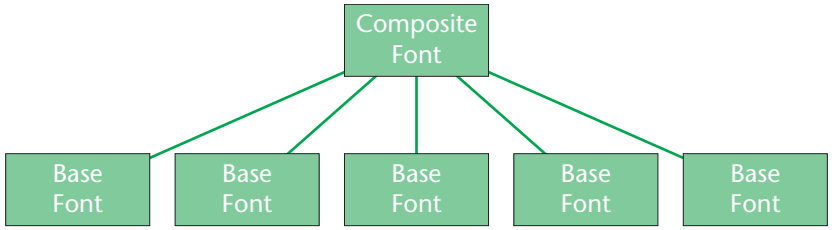
printed as:

We often use outline text.

This month, let's look at how composite fonts in some detail. We shall se how they are constructed and how we can use them to combine several different fonts into one package and thereby imporve PostScript driver output.

Next Page ->

Composite Fonts: Origin and Use



A composite font is a font that contains other fonts, rather than characters. The composite font contains "Base Fonts," which are just regular PostScript fonts containing characters. (Actually, these descendent fonts can, themselves, be composite fonts, but our discussion will ignore this case.)

Composite fonts were Adobe's first pass at supporting Kanji and other scripts with extended character sets, but they never really caught on as such. Although their character capacity was quite large, implementing any of the standard Asian character encodings (JIS, Shift-JIS, etc.) often involved some convoluted programming. At this point, support for multibyte fonts has passed completely from composite fonts to the newer CID font mechanism.

Nonetheless, the composite fonts continue to have a very good application, unintended by its designers, I believe, in the printing of Roman typography. By building a composite font that contains all of your documents fonts, you can switch among those fonts from within the *show* string. This is both faster and more convenient than executing a *setfont* and *show* every time you want to change from Times-Roman to Times-Bold.

Let's start by examining the structure of a composite font.

Next Page ->

Acumen Journal: PostScript Tech 14

Composite Font Dictionaries

This discussion assumes that you at least vaguely remember the font discussion in the PostScript Foundations or Support Engineers class.

Like any other PostScript font, a composite font is defined by a dictionary whose key-value pairs supply everything needed for the font. The key-value pairs that we must supply when making a composite font are listed at right.

Let's look at each of these in turn.

Composite Font Key-Value Pairs

Key	Value Type
FontType	0
FontMatrix	[a b c d e f]
FDepVector	[array of fdicts]
Encoding	[array of ints]
FMapType	integer

FontType This is an integer that tells the PostScript font mechanism how to interpret the contents of this dictionary. A FontType of *0* indicates a composite font.

FontMatrix This has exactly the same interpretation as in a regular font: it is a transformation matrix that maps the character definitions' coordinate system into User Space.

Composite fonts don't have characters, of course; they have descendent fonts that contain the characters. The final size of the printed characters is determined by the concatenation of the font matrices of the base font and the composite font that contains it.

This being so, the default FontMatrix in a composite font is usually an identity matrix:

```
/FontMatrix [ 1 0 0 1 0 0 ]
```

Next Page ->

Acumen Journal: PostScript Tech 15

FDepVector This is an array of the font dictionaries that make up the immediate descendent fonts within the composite font. A composite font that contained all of the fonts in the Times-Roman font family would have an *FDepVector* that looks like this:

```
/FDepVector    [
     /Times-Roman findfont
     /Times-Bold findfont
     /Times-Italic findfont
     /Times-BoldItalic findfont
]
```

Note the *findfont* calls; this is an array of font dictionaries, not font names.

The fonts within the *FDepVector* do not need to be of the original 1-point size. If one of the descendent fonts has characters significantly smaller than the others, you can scale it to better match the others:

```
/FDepVector    [
     /Helvetica findfont
     /Symbol findfont 1.2 scalefont
]
```

Next Page ->

Acumen Journal: PostScript Tech 16

Encoding This is an array of integers; each integer is an index into *FDepVector.* The *Encoding* array defines the mapping of font codes in the *show* string into font dictionaries in *FDepVector.*

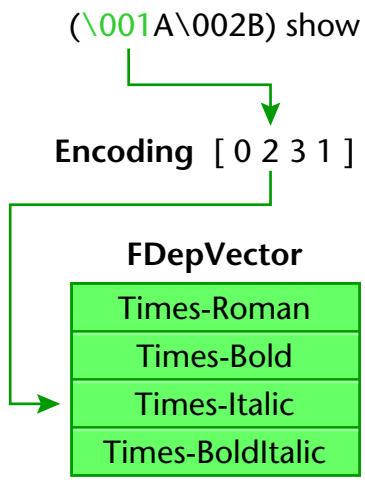
If the current font is a composite font, each byte within a *show* string will be either a character code (identifying a character to print) or a font code (identifying the descendent font within which that character resides).

Whithin the *show* string, a font code is taken as an index into the *Encoding* array; the resulting number is then taken as an index into *FDepVector*, identifying the descendent font that should be taken as the source of character shapes.

Thus, if our font's *FDepVector* and *Encoding* are defined as follows:

```
/FDepVector    [
     /Times-Roman findfont
     /Times-Bold findfont
     /Times-Italic findfont
     /Times-BoldItalic findfont
]

/Encoding [ 0 2 3 1 ]
```



Within the *show* string, a font code of *1* would map into Times-Italic, as illustrated at right. Index *1* (from the font code) into *Encoding* gives us the number *2;* index *2* into *FDepVector* gives us Times-Italic.

Next Page ->

FMapType Finally, this numeric code indicates exactly how font codes and character codes are packed into the *show* string. There are only two values I'm going to discuss here:

2 **8/8 Mapping –** The *show* string consists of alternating 8-bit font codes and character codes. If we are using this mapping, and *FDepVector* and *Encoding* are defined as on the previous page, the following call to show:

```
(\000A\001B\002C) show
```

Would print as follows:

A*B**C*

You can use this mapping to support double-byte fonts.

3 **Escape Mapping –** Bytes within a *show* string are treated as character codes until an escape character is encountered (by default, \377); the byte following the escape character is a font code. The string starts out in font 0. Again, using our previous *FDepVector* and *Encoding*:

(This is Times \377\001Italic. \377\000And how are you?) show

Would print as:

This is Times *Italic.* And how are you?

The latter example is germane to our plot, since it would take three *show's* with two interleaved *setfonts* to print this without composite fonts. We'll return to this later.

Next Page ->

Acumen Journal: PostScript Tech 18

Creating a Composite Font

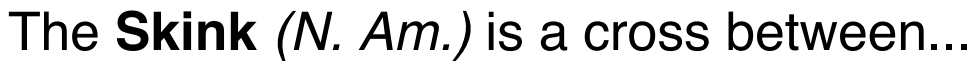
Creating a composite font is relatively easy. Here, we create a composite font named "Helv" that contains the four fonts in the Helvetica font family:

This file is on the Resources page of the Acumen Training website.

```
/Helv
<<
     /FontType      0                   % Composite font
     /FontMatrix    [ 1 0 0 1 0 0 ]     % Identity matrix
     /FDepVector    [                   % Array of descendant fonts
          /Helvetica findfont
          /Helvetica-Bold findfont
          /Helvetica-Oblique findfont
          /Helvetica-BoldOblique findfont
     ]
     /Encoding      [ 0 2 3 1 ]         % Map font codes to fonts
     /FMapType      3                   % Use escape mapping
>> definefont pop

/Helv 20 selectfont
72 600 moveto
(The \377\003Skink \377\001(N. Am.)\377\000 is a cross between...) show

showpage
```



Next Page ->>

Discussion This example uses bits and pieces we have already discussed, so I won't repeat the descriptions of *FDepVector,* etc. Overall, the program is pretty straightforward:

```
/Helv
<<
     ...
>> definefont pop
```

We create a dictionary and turn it into a font named Helv. Because *FontType* is *0* within the dictionary, the font is a composite font.

```
/Helv 20 selectfont
72 600 moveto
```

Note that you use a composite font just as you do any other PostScript font; *findfont, findresource, selectfont,* and all the other PostScript font operators make no distinction between composite fonts and any other font type.

```
(The \377\003Skink \377\001(N. Am.)\377\000 is a cross between...) show
```

And finally, here is the single call to *show* that prints text in three different fonts. Note that we are using escape mapping, so the string starts out in font *0* (Helvetica, in our case). This string prints out as:

The **Skink** *(N. Am.)* is a cross between...

Next Page ->

Acumen Journal: PostScript Tech 20

Why Is This Cool?

Consider our text from the previous example. A typical PostScript driver would print this line of text without composite fonts, generating PostScript that looks like this:

```
% In the prolog...
/F1 /Helvetica findfont 20 scalefont def
/F2 /Helvetica-Bold findfont 20 scalefont def
/F3 /Helvetica-Oblique findfont 20 scalefont def

% Later, in the script...
72 600 moveto
F1 setfont
(The ) show
F2 setfont
(Skink ) show
F3 setfont
((N. Am.)) show
F1 setfont
( is a cross between...) show
```

Note that we need four *shows* and three intermediate *setfonts* to print this text, compared with the previous example's single *show:*

```
(The \377\003Skink \377\001(N. Am.)\377\000 is a cross between...) show
```

Next Page ->

Acumen Journal: PostScript Tech 21

Driver Implications

A PostScript driver could create a single composite font that contains all of the fonts used in that document, scaled as needed.

```
/AllFonts
<<
     /FontType      0
     /FontMatrix    [ 1 0 0 1 0 0 ]
     /FDepVector    [
          /Helvetica findfont 20 scalefont
          /Helvetica-Bold findfont 20 scalefont
          /Helvetica-Oblique findfont 20 scalefont
     ]
     /Encoding      [ 0 1 2 ]
     /FMapType      3
>> definefont pop
```

It could then generate a single *setfont* at the beginning of each page:

```
/AllFonts findfont setfont
```

For the remainder of the page, it would never need to change fonts, instead placing appropriate font codes into its *show* strings.

```
(In their hit single \377\002Don’t Touch My Bric-a-Brac\377\000 the) show
```

Next Page ->

Acumen Journal: PostScript Tech 22

Try It!

This topic has long been a bit of an obsession with me. I am convinced that composite fonts could moderately speed up and considerably tidy up the PostScript generated by PostScript drivers. In the Advanced PostScript class I tend to harp on this during the Composite Font discussion. Some of my students (particularly those who are doing variable data printing and other tasks where they write their own PostScript code) have adopted this technique and seem to like it quite a bit.

Give composite fonts a try, if you are printing text.

What can it hurt? And it would make your mother so happy.

Return to Main Menu

Schedule of Classes, May – July, 2002

Following are the dates and locations of Acumen Training's upcoming PostScript and Acrobat classes. Clicking on a class name below will take you to the description of that class on the Acumen training website.

The PostScript classes are taught in Orange County, California and on corporate sites world-wide. See the Acumen Training web site for more information.

PostScript Classes

PostScript Foundations	May 13 – 17	July 29 – Aug 2
Advanced PostScript	June 24 – 28	
PostScript for Support Engineers	June 3 – 7	
Jaws Development	July 23 – 26	

For more classes, go to www.acumentraining.com/schedule.html

PostScript Course Fees

PostScript classes cost $2,000 per student.
These classes may also be taught on your organization's site.
Go to www.acumentraining.com/onsite.html for more information.

Registration →
Acrobat Classes →

Acrobat Class Schedule

These classes are taught quarterly in Costa Mesa, California, and on corporate sites. Clicking on a course name below will take you to the class description on the Acumen Training web site.

Acrobat Essentials	Jun 20 (1/2-day, morning)
Interactive Acrobat	
Creating Acrobat Forms	Jun 20 (1/2-day, afternoon)
Troubleshooting with Enfocus' PitStop	June 21 (Full day)

Acrobat Class Fees

Acrobat Essentials and *Creating Acrobat Forms* (1/2-day each) cost $180.00 or $340.00 for both classes. *Troubleshooting With PitStop* (full day) is $340.00. In all cases, there is a 10% discount if three or more people from the same organization sign up for the same class.

Registration ->

Return to Main Menu

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: http://www.acumentraining.com **email:** john@acumentraining.com

telephone: 949-248-1241

mail: 25142 Danalaurel, Dana Point, CA 92629

Registering for Classes

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

Register On-line: http://www.acumentraining.com/registration.html

email: registration@acumentraining.com

telephone: 949-248-1241

mail: 25142 Danalaurel, Dana Point, CA 92629

Back issues

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

Return to First Page

What's New at Acumen Training?

Quarterly Acrobat Classes in SoCal

In response to continual inquiries, Acumen Training will now be offering its Acrobat classes in Costa Mesa, CA. For the moment, this makes them of interest mostly to people in the Los Angeles area, but one must start somewhere.

The first classes will be offered June 20 and 21. The $^1/_2$-day *Acrobat Essentials* and *Creating Acrobat Forms* classes will be conducted on the 20th and the PitStop course on the 21st. See the Acumen Training website for course descriptions, pricing, etc.

Credit Card Payments

Acumen Training is now accepting credit cards for on-line payment of class fees for the Costa Mesa classes. When you register for class on-line, there is a button that takes you to the on-line payment page. Of course, I still accept payment by company check, as well.

Note that a 5% surcharge applies to PostScript classes that are paid by credit card.

Return to First Page

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 you regret having ever learned to read?

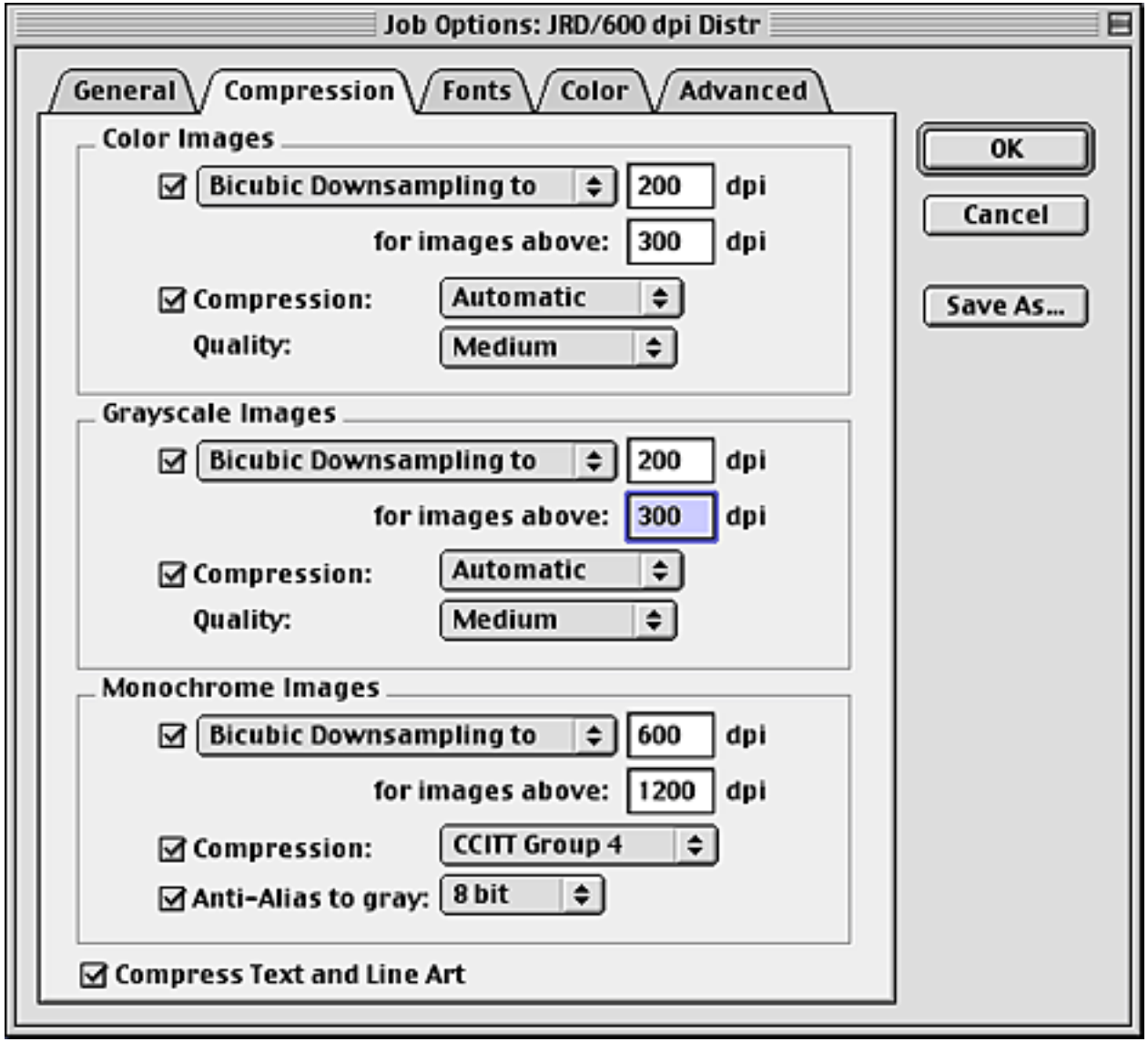
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

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:

journal@acumentraining.com

Return to Menu


Job Options: JRD/600 dpi Distr
General
Compression
Fonts
Color
Advanced
Color Images
Bicubic Downsampling to
200
dpi
for images above:
300
dpi
Compression:
Automatic
Quality:
Medium
Grayscale Images
Bicubic Downsampling to
200
dpi
for images above:
300
dpi
Compression:
Automatic
Quality:
Medium
Monochrome Images
Bicubic Downsampling to
600
dpi
for images above:
1200
dpi
Compression:
CCITT Group 4
Anti-Alias to gray:
8 bit
Compress Text and Line Art
OK
Cancel
Save As...