Table of Contents

Acumen Training

Acrobat 5 Hairlines, Revisited

PostScript Tech

Concatenati
Combining se
people. Turns

Acrobat 5 displays hairlines much thicker than they should be, due to a peculiarity in how it rounds off line widths. Last September, we looked at a technique for fixing this problem with a combination of automatic stroke adjustment and the PostScript *BeginPage* mechanism.

This technique doesn't work with the PostScript — and therefore the PDF files — produced by some applications, notably Microsoft Word.

This month, we'll add to our earlier code to extend our fix to some of the missing applications.

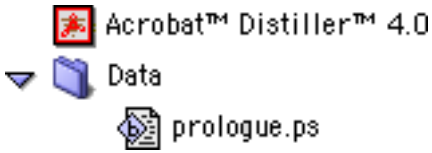
Next page ->

Acumen Journal: The Acrobat User

Where We Left Off

When last we left our heroes, we had fixed Acrobat 5's hairlines by turning on automatic stroke adjustment. We did this with a call to *setstrokeadjust* installed as a *BeginPage* procedure that would execute at the beginning of each page.

We did this by placing the following PostScript code into a file named *prologue.ps* in Distiller's *Data* folder.




```
<< /BeginPage
     { pop true setstrokeadjust} bind
>> setpagedevice
```

Important Note: You probably should reread the September 2001 issue of the Acumen Journal before reading the rest of this month's article.

The Problem Continues...

In most cases, this technique works very well. Unfortunately, some applications are resistant.

- Some applications explicitly turn stroke adjustment off again, undoing our fix.
- Some applications draw their underline strokes as very thin, filled rectangles, which are unaffected by automatic stroke adjustment. (Microsoft Word and FrameMaker do this.)

This month, we'll add a little more PostScript code to our fix to bypass some of these resistant strains.

Next page ->

Acumen Journal: Acrobat User

Disabling *setstrokeadjust*

One of PostScript's very useful characteristics is the ability to redefine the PostScript *operators,* the intrinsic PostScript commands. When the PostScript output says to do one thing, a PostScript programmer can make it do something else, entirely.

This is somewhat dangerous programming, but very powerful. Many PostScript imposition and other post-processing software make extensive use of this feature of the language.

We can use this to keep a program's PostScript output from changing automatic stroke adjustment back to *false,* undoing our *prologue.ps* activities. We are going to add the following line to our *prologue.ps* after the earlier call to *setstrokeadjust.*

```
/setstrokeadjust /pop load def
```

This redefines the *setstrokeadjust* operator to do nothing at all. This redefinition will affect all calls to *setstrokeadjust* that occur after our prologue.ps. In particular, when an application's PostScript output tries to change stroke adjustment to *false,* nothing will happen.

Our *prologue.ps* file now looks like this:

```
<< /BeginPage
     { pop true setstrokeadjust} bind
>> setpagedevice

/setstrokeadjust /pop load def
```

Next page ->

Fixing Rules-as-Rectangles

Fixing the PostScript output from applications that use filled rectangles, rather than stroked lines, for their underlines must be done on an application-by-application basis. I'm going to present a fix here that should work for many applications, including Microsoft Word.

Redefining *rectfill*

The PostScript *rectfill* operator draws a filled rectangle on the page. This operator is used by many applications to draw filled rectangles quickly. Microsoft Word, in particular, uses this operator to draw the underline strokes (as filled rectangles) for underlined text.

To fix this, we can redefine *rectfill* so that it draws a stroked line that exactly matches the filled rectangle. That is, our *rectfill* will draw a stroked horizontal line that extends through the middle of the proposed rectangle and that has a line width equal to the rectangle's height. This stroked line will look exactly like the original filled rectangle. (See the illustration at right.)

Filled Rectangle

Stroked Line

This stroked line will be subject to automatic stroke adjustment and, therefore, improve their on-screen appearance.

Next page ->

Acumen Journal: Acrobat User

Our New prologue.ps

So, our Prologue.ps now looks like this:

```
<< /BeginPage
     { pop true setstrokeadjust} bind
>> setpagedevice

/setstrokeadjust /pop load def

/rectfill
{
     dup 8 gt
     { rectfill }
     {    gsave
          4 2 roll moveto
          dup setlinewidth
          2 div 0 exch rmoveto
          0 rlineto stroke
          grestore
     }
     ifelse
} bind def
```

The new *rectfill* definition checks the height of the rectangle and substitutes a stroked line if the height is less than a threshold value (8, in our example above). Otherwise, it does a regular *rectfill.* (Note to PostScript programmers: we're depending on *bind* to keep this from being recursive.)

Next page ->

Acumen Journal: Acrobat User

Caveats to the *rectfill* redefinition

There are some important points to keep in mind when using this version of prologue.ps.

Application-specific

The PostScript code that converts a rectangle to a stroked line is *very* dependent upon the details of the PostScript output. Our prologue.ps will fix underline strokes for documents created in Microsoft Word, but not necessarily any other application's document. You'll need to try it and see. (For that matter, I'm doing all my work with the Macintosh version of Microsoft Word; it may fail on documents created with the Windows version.)

Mildly dangerous

Conceivably, this redefinition of could cause some exotic PostScript output to fail. A redefinition of *rectfill* should be reasonably safe, but if some application's output suddenly fails to Distill, you may want to return to the simpler prologue.ps.

Next page ->

A Call for PostScript Samples

Again, the prologue.ps that fixes an application's underline strokes is very specific to that particular application; it may also work on others, but there's no guarantee.

If you have a document that this prologue.ps doesn't fix, please send me the PostScript output that exhibits the problem. I'll try to put together a prologue.ps that fixes that application's output. (For example, could someone please send me some PostScript output from FrameMaker?)

Please send me PostScript from the simplest document you can create that demonstrates the problem. (Maybe a document with a single, underlined word.)

Prologues.zip

All of the versions of prologue.ps are on the Acumen Training resources page in a file named *Prologues.zip*. (www.acumentraining.com/resources.html) I'll add new prologue.ps files for particular PostScript output as they become available.

To start, there are only two prologue.ps files in this zip file:

prologue.ps This is last September's *prologue.ps* plus the *setstrokeadjust* redefinition. Use this unless you are having trouble with a particular piece of output.

prologueWord.ps This contains the *rectfill* redefinition required by Microsoft Word and other applications.

To use one of these files, rename it to *prologue.ps,* if necessary, and place it in the *data* folder in your system's *Distiller* folder.

Return to Main Menu

Saving PostScript Code to Disk

Here's a frequent question: how to you store often-used PostScript code on a printer so that you can repeatedly execute it within a PostScript stream?

Let's say you have a logo created in Adobe Illustrator (or whatever) and saved as an Encapsulated PostScript file.

If you use this EPS file as an illustration in a PostScript document, you will need to embed the EPS' PostScript code in the print stream once for each instance of the EPS illustration in the document. If you place this logo on each page of a 50-page document, then the EPS code will need to be embedded in the PostScript print stream 50 times.

(The "Sparrow Boy" EPS file above contains just under 100k of PostScript; the 50 embedded instances would amount to 5 megabytes of PostScript.)

What we would like to do is send the EPS file's PostScript to the RIP only once and then just refer to that stored PostScript whenever we want to print the EPS file.

This is perfectly do-able if you have either a hard disk available to your RIP or a PostScript Level 3 printer.

This month we'll look at the solution that uses a hard disk.

Next page ->

Acumen Journal: PostScript Tech

Overview of the Process

In short, what we shall do is this:

1. Store our repeatable PostScript on a hard disk available to our PostScript RIP.
2. Whenever we want to execute this PostScript, we shall do so with the PostScript *run* operator.

```
(%disk1%EPSFiles/SparrowBoy.eps) run
```

We shall discuss this technique in terms of a repeatedly-printed EPS file, but it works as well with any PostScript code or data. If you have a commonly-used set of PostScript definitions, you can save it in a file and execute the definitions with a single *run.*

A True Story

This is a very effective method for reducing the size of a PostScript stream. As an example:

The overheads file for my Acrobat Essentials class used to have the Acquired Knowledge logo (as at right) on each page as an EPS file. Printing the 400-page document to a PostScript file (in preparation for Distilling to PDF) yielded a 12 MB PostScript file.

It turned out that 10 MB of the 12 MB file was the 400 repeated instances of the AKI logo EPS code. Storing the code on disk and using *run* reduced the PostScript file to a bit *under* two megabytes.

Next page ->

Saving PostScript to Disk

Saving a stream of PostScript to disk is relatively simple. It's actually just a variation on one of the examples we present in the *PostScript Foundations* and *PostScript for Support Engineers* classes. (For those who've taken one of those classes, it's the *readline* example in which we printed the Mark Twain blue jay quote to the current page.)

It's important to note that this technique requires a hard disk that is directly available to the PostScript RIP.

- If you are using Acrobat Distiller, PDF Creator, or GhostScript, files can be stored on your computer system's hard disk.
- If you are using a printing device with a RIP running on a standalone computer, the destination file will be on the RIP's system.
- If you have a self-contained desktop printer, then the hard disk must be either built into the printer itself or connected to the printer's SCSI, USB, or other port.

Next page ->

The Algorithm What we need to do is this:

1. Open the destination file into which we shall save our PostScript code.
2. Start up a loop, which does the following:
   - Read a buffer of PostScript code from the input stream.
   - Write that buffer to the destination file.

   The loop repeats until the end of our PostScript, at which time we'll close the destination file and quit.

The PostScript to be stored will follow the invocation of the loop in the PostScript file.

Next page ->

Acumen Journal: PostScript Tech

The Code This PostScript code defines a *WriteToFile* procedure, which implements our algorithm. This is available in *WriteToFile.ps* on the Acumen Training resources page: www.acumentraining.com/resources.html.

```
/InBuf 8192 string def                          % Our input buffer

/WriteToFile   % (filename) => ---
{
     /destination exch (w) file def             % Open destination file
     {
          currentfile InBuf readstring          % Read PS from currentfile
          exch destination exch writestring     % Write PS to destination
          not { exit } if                       % Exit if EOF
     } loop                                     % Else, do it again
     destination closefile                      % Close our destination file
} bind def

(psfiles/Sparrow.eps) WriteToFile               % Invoke procedure
%!PS-Adobe-3.0 EPSF-3.0                         % This is written to file
%%Title: (Sparrow Logo.eps)
...
```

Next page ->

Acumen Journal: PostScript Tech

The Code, Step-by-Step

```
/InBuf 8192 string def
```

We'll be reading our PostScript code into this buffer. I'm allocating an 8k buffer here. Make this as large as you have memory for, up to a maximum of 64k.

```
/WriteToFile    % (filename) => ---
```

This is the procedure that does the actual work. It takes a string argument containing the name of the file into which the PostScript should be stored. Note that this may be a complete pathname or simply a filename, in which case the file will be created on the RIP's default volume.

```
/destination exch (w) file def
```

Here we open the destination file with write permission and give it the key-value name "destination."

```
{
      currentfile InBuf readstring        % Read PS from currentfile
```

We begin our loop by reading a buffer of PostScript code (or other data) from currentfile. The *readstring* operator, as always, returns a boolean on top of the stack (indicating end-of-file if false) and, beneath that, *InBuf* again, now holding data read from currentfile. (Check your student notes if you've forgotten how *readstring* works.)

Next page ->

Acumen Journal: PostScript Tech

```
exch destination exch writestring
```

We bring the string full of data to the top of the stack and then write it to *destination.* This leaves the *readstring* boolean on top of the stack. We want to exit from the loop if this boolean is false.

```
    not { exit } if
} loop
```

We reverse the *readstring* boolean and exit from the loop if the reversed boolean is true (that is, if the original boolean was false).

Otherwise, we loop back and get another buffer of data.

```
    destination closefile                    % Close our destination file
} bind def
```

We finish our procedure definition by closing the destination file.

Nothing actually happens, of course, until we invoke our *WriteToFile* procedure:

```
(psfiles/Sparrow.eps) WriteToFile
%!PS-Adobe-3.0 EPSF-3.0
%%Title: (Sparrow Logo.eps)
```

Everything from the execution of *WriteToFile* to the end of the PostScript stream will be written to the destination file.

Next page ->

Acumen Journal: PostScript Tech

Executing the PostScript

Having stored the PostScript code on disk, we can execute it in any future PostScript stream sent to that particular RIP with a single call to the *run* operator:

```
(psfiles/Sparrow.eps) run
```

The *run* operator takes the name of a file on the RIP's disk and executes its contents. Instead of transmitting the original 100k EPS file, we transmit the 25-byte *run* execution.

Using Code in a Form

There are other things we can do with the stored PostScript code. For example, we could execute the file as part of a PostScript Level 2 form:

```
<<
     /FormType 1
     /BBox [ 0 0 325 155 ]
     /Matrix [ 1 0 0 1 0 0 ]
     /PaintProc     {
          pop
          (psfiles/Sparrow.eps) run
     }
>> execform
```

This way, the EPS code would be transmitted *and* executed only once.

The above is a bit simplified. In particular, the *PaintProc* should do a *save/restore* and other bookkeeping chores.

Next page ->

Acumen Journal: PostScript Tech

A Few Final Notes

The data is only available on *this* RIP

Keep in mind that your call to *run* will work only on printers on which you have stored your PostScript code. This ties your PostScript streams to a particular set of printers. This is perfectly acceptable if you work in a print environment with a known set of printers. It makes this technique inappropriate for PostScript that will be sent to wholly unknown printers.

Other Data on Disk

In our example, we saved to disk PostScript code taken from an EPS file for later repeated execution. This technique may also be used to store and use other kinds of data.

For example, you could save image data to the disk (just follow the invocation of *WriteToFile* with image data, instead of executable PostScript) and then print the image, taking the data from disk:

```
200 300 8 [ 200 0 0 -300 0 300 ]
(datafiles/monkeytoes.data) (r) file
image
```

PostScript will automatically close the data file when you hit the end of file, by the way.

Next page ->

Acumen Journal: PostScript Tech

Compressed Data/PostScript

With a relatively minor modification to *WriteToFile,* you can write your information to disk through a compression filter, storing the data or PostScript code in LZW compressed form, for example. There is a second sample file on the Resources web page, *WriteToDisk.ps* which demonstrates this.

Executing Compressed PostScript

By the way, you need to do a little work to execute compressed PostScript code; you can't just execute *run* with the filename. Instead, you must explicitly open the file and execute it through the appropriate decode filter.

For example, the following would execute a file containing LZW-compressed PostScript:

```
(PSFiles/Bunny_Yoga.ps) (r) file
/LZWDecode filter cvx exec
```

Next page ->

Acumen Journal: PostScript Tech

What if my RIP doesn't have a hard disk?

If your RIP doesn't have direct access to a hard disk, then your only good option is to use the PostScript Level 3 *ReuseableStreamDecode* filter, which will let you save your PostScript code or other data as a "virtual" file in RAM.

That's a *very* long story, occupying a good hour's discussion in the Advanced PostScript class. (By which I mean it's not a topic that will ever appear in the Journal. Sorry.)

But my RIP isn't Level 3, either.

Bummer.

In that case, there is no good method for storing arbitrary PostScript or data in the printer for repeated use. If the PostScript is extremely simple and doesn't attempt to read the input stream, you may be able to turn your PostScript into a procedure:

```
/MyEPSFile
{
      ...
      ... Your PS Code Here
      ...
} bind def
```

This is way more likely to fail than not, usually with a *stackoverflow* error. (PostScript Foundations students: do you remember why this fails with *stackoverflow?* Well, why not?)

Time to upgrade your printer, maybe?

Return to Main Menu

Schedule of Classes, January – April, 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	March 18 - 22	
Advanced PostScript	January 21 – 25	April 29 – May 3
PostScript for Support Engineers	January 14 – 18	April 15 – 19
Jaws Development	April 2 – 5	

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.

Registration →
Acrobat Classes →

Acrobat Class Schedule

On-Site Only These classes are taught only on corporate sites. If you have an interest in any of these classes for your group, please see the Acumen Training website regarding arranging an on-site class.

Acrobat Essentials This class teaches the student how to make perfect PDF files. It includes complete coverage of the meaning and proper settings of all of the Distiller Job Options.

Interactive Acrobat Here we show you how to add bookmarks, links, buttons, sounds, movies, form fields, and other interactive features to an Acrobat file.

Creating Acrobat Forms This class shows you how to make interactive forms in Adobe Acrobat. It steps you through creating the form, posting form contents to a server, and everything else you need to create a working PDF form.

Troubleshooting with Enfocus' PitStop This class shows the student how to use all of the capabilities of this popular editing and preflight software.

Back to PostScript Classes

Return to First Page

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 us 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?

See you at Seybold

I'll be teaching two Acrobat classes at the New York Seybold Seminars, February 19 & 20. These are:

PDF for Prepress

A slightly shortened version of *Acrobat Essentials.*

Creating Acrobat Forms

A slightly shortened version of my usual *Creating Acrobat Forms* class.

If you attend the Seybold Seminars in February, come by and say "Hi."

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, or does it make you want to make faces at small children?

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