Table of Contents

The Acrobat User

Image Resampling in Acrobat Distiller
Distiller and many other programs that create PDF files allow you to resample the images placed in that file. Most people are vague about what resampling is and how it differs from compression. This month we look at this powerful feature.

PostScript Tech

Handling PostScript Errors, Part 1
The default PostScript error handler typically sends the name of the error and the offending command to the output stream. This month and next, we'll see how to write our own error handler to report on additional information and print the message to the current page.

Class Schedule

May-Aug

What's New?

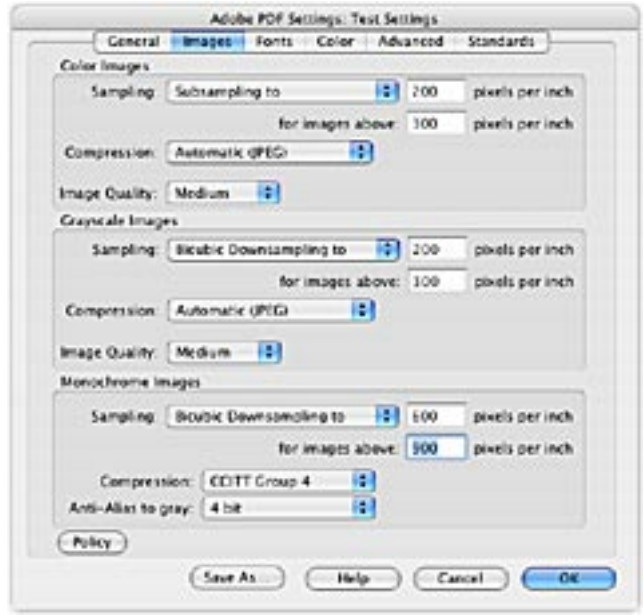
Nothing new really. Been busy.
The second *PDF File Content and Structure* class will be ready mid 2005. Really!

Contacting Acumen

Telephone number, email address, postal address

Journal feedback: suggestions for articles, questions, etc.

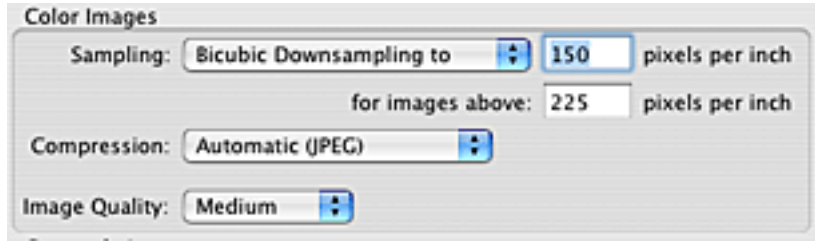
Image Resampling in Acrobat Distiller



Among Distiller's PDF settings, you will find a set of controls that dictate whether and how images in the PDF file should be compressed.

For compression, you get your choice among ZIP and various flavors of JPEG. There are also three controls in each group—a pop-up menu and two text fields—that ask you if, when, and how you want images to be "sampled."

Many people find these controls genuinely alarming, so this month we shall look at what, exactly, sampling is and how you should set these controls. This is worth knowing because resampling your image data is a powerful way of greatly reducing PDF file size and, simultaneously, *improving* the appearance of some of the images in your file.



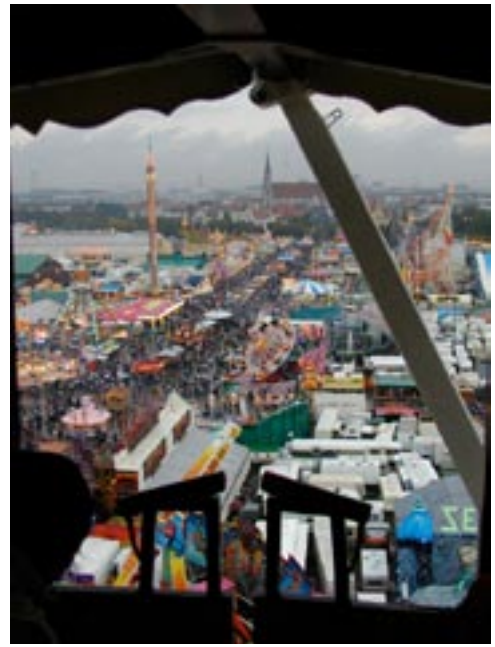
Truly, image resampling is a Good Thing in the PDF world and well worth learning about.

So let's learn.

Next Page ->

Background: What's it for?

Image resampling addresses itself to the case where an image has more data in it than the printer or display device can reproduce.



Consider the image at right. The original of this image had a size of 507 x 676 pixels. At right this image has been reproduced on the screen nominally 2½ inches wide (ignoring details like the pitch of your screen).

Five hundred seven image pixels spread across a width of 2½ inches comes to a final, scaled resolution of about 200 pixels per inch. This is far more data than the your computer screen can reproduce. In fact, you could discard a remarkable amount of this data and the difference would never be noticeable.

This is what image resampling is: removing data from an image to match the amount of data a display or printing device can actually use.

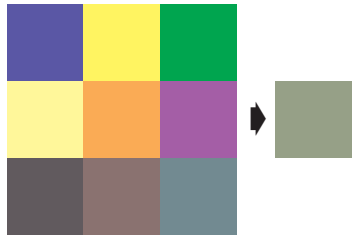
We could the image at right from 200 dpi to 72 dpi, discarding about 87% of the original data. Done right (and we'll discuss what that means), there would be no perceptible difference in quality, since the screen only used 13% of the original data in the first place.

Next Page ->>

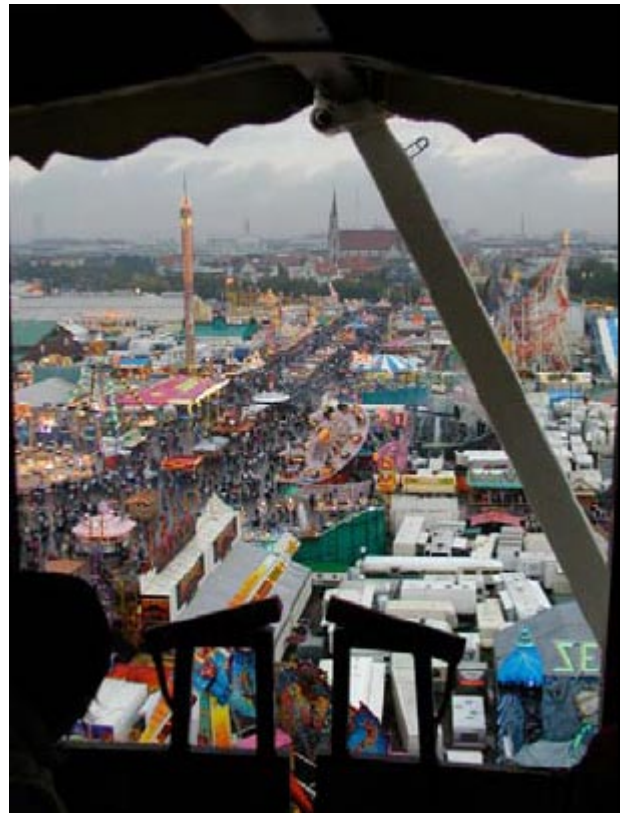
Resampling Methods

Files on the Web Site

As usual, these images are available on the Acumen Training Resources page. Look for the file *Resampling.zip*.



Resampling entails replacing clusters of pixels in the original data with single pixels. The software that does the resampling (Acrobat, in our case) must calculate a color for the replacement pixel that will look like the overall color of the originals.

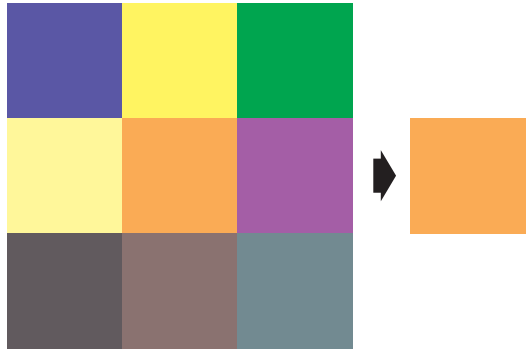


Acrobat gives us a choice of three methods by which it can calculate the replacement color: *subsampling, linear downsampling,* and *bicubic downsampling*.

We are going to see how each of these works and see what effect they each have on our photograph. Specifically, we shall zoom in on the lower-right corner of the image and see how the image changes with each resampling type.

Next Page ->>

Subsampling This is the simplest method of resampling images; Acrobat simply discards all of the pixels but one in each cluster. The replacement pixel has exactly the same color as one of the pixels in the original set.



Advantage The only advantage of this method of resampling is that it is very fast, so it takes somewhat less time to create the PDF file. Note that this has no effect on the later viewing of the PDF file; it only speeds the creation of the file.

Disadvantage Subsampling has a seriously bad effect on the quality of the resulting image:

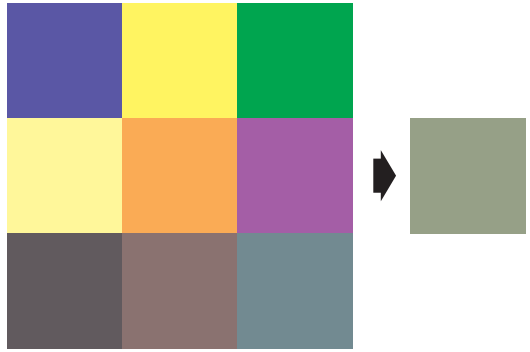
- Colors can change significantly, since the process completely discards the data in the missing pixels.
- Edges become excessively "stairstepped," reducing the smoothness of the image.

(We'll see a comparison of the results of subsampling in just a moment.)

Next Page ->>

Downsampling

Downsampling calculates the color of the replacement pixel from the colors of the original pixels. This yields *much* better results, because all of the removed pixels contribute to the color of the single replacement pixel. This minimizes color shifting and retains the appearance of sharp edges.



Acrobat can apply one of two different kinds of downsampling to an image:

Average Downsampling

Average Downsampling

Average downsampling calculates the color of the replacement pixel to be the arithmetic average of the colors of the pixels in the original cluster. (Thus, the red component of the replacement pixel is the sum of the red components of the original pixels divided by the number of pixels in the cluster.)

Bicubic Downsampling

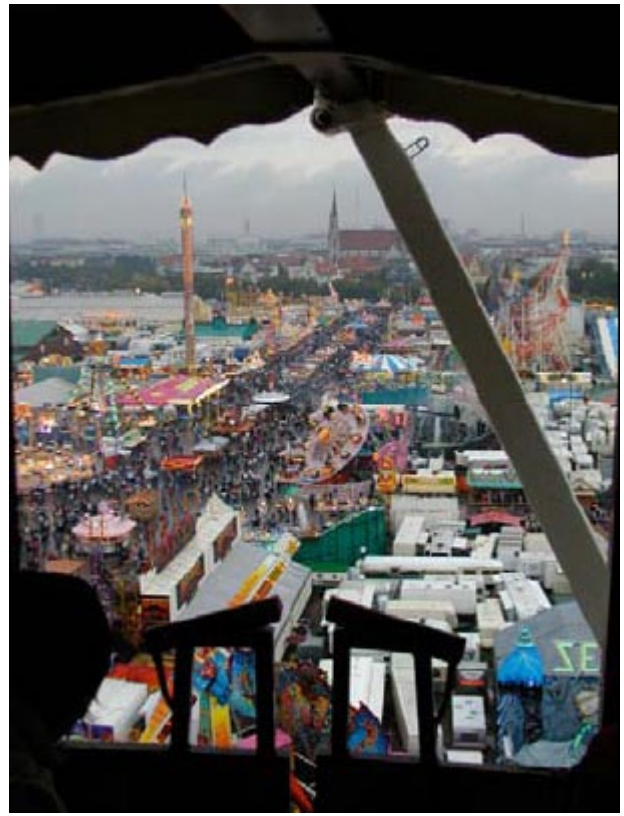
Bicubic downsampling applies a more sophisticated algorithm to the pixels in the original cluster; the resulting color of replacement pixel will closely match how the eye would blend the original colors if they were seen from a distance. This yields a new image that looks remarkably like the original. (This is the type of downsampling that *Adobe Photoshop* applies when it resizes an image.)

Next Page ->

Comparisons

Let's return to our sample image. Below are four close-ups taken from the lower-right corner of the image. They are, from left to right:

- The original image.
- The image resampled to half its original resolution using subsampling.
- The image resampled with average downsampling.
- The image resampled with bicubic downsampling.



To make it easier to see the differences among these, on the next page we have scaled the subsampled images to 200%.

Next Page ->

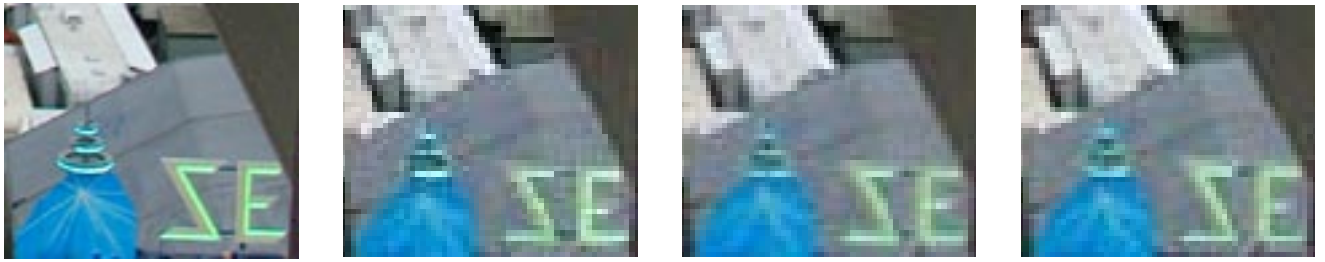
A Detailed Look

Here we have the same set of pictures, but we have zoomed to 200% on the resampled images.

From Left to Right

Again, these are:

- Original
- Subsampled
- Average downsampled
- Bicubic downsampled



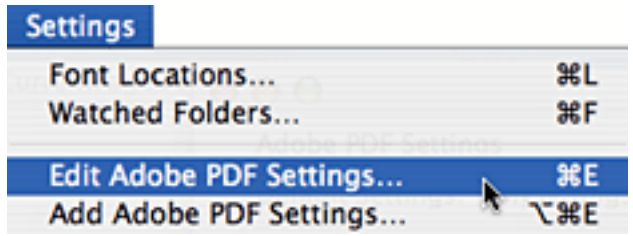
Looking at the three resampled images, we can note the following:

- The subsampled image (second from the left) is clearly the least acceptable of the three resampled images. Compared to the other two, this image has a lot of very ragged edges with accentuated stairstepping. The light blue, radial lines in the blue dome have become a smeared, lightish blotch.
- The average downsampled image looks significantly better. The calculated replacement colors provides something akin to anti-aliasing, smoothing out the edges and giving us a better representation of the original detail. The gray roof looks more uniform in color.
- The bicubic image looks remarkably like the average downsampled image. It is an improvement, but only by a little. The light blue radial lines in the blue dome are more distinct; the little blue-edged disks above the dome actually have some curvature.

Next Page ->

Distiller's Controls

In Acrobat Distiller, the controls that dictate how—and if and when—images are resampled reside among the PDF Settings, in the *Images* panel. (This may be called the *Compression* panel, depending upon your version of Acrobat.)



To get there, select *Adobe PDF Settings* from Distiller's *Settings* menu and then click on the *Images* (or *Compression)* tab in the resulting dialog box.

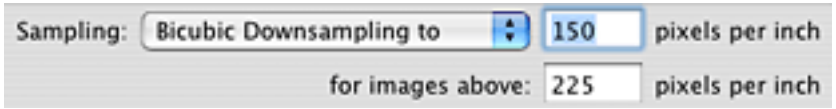
There are three sets of controls in this panel, dictating the compression and resampling of color, grayscale, and monochrome images.

We discussed compression in the May 2002 issue of the *Acumen Journal;* I'll refer you to that issue if you want to review compression.

Here we shall look at the resampling controls.

Next Page ->

Overview

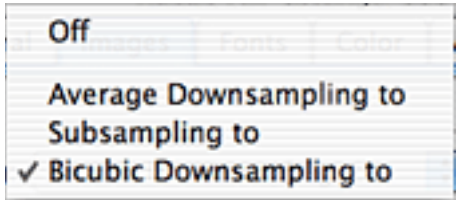


The Distiller resampling controls allow you to specify that images whose effective resolution, after scaling, are greater than some threshold value should be resampled to a specified, lower resolution.

For example, the usual rule of thumb for images intended for print is that they should be scanned at a resolution that is twice the halftone screen frequency of the printing device. Thus, an image intended to be printed on an imagesetter with a screen frequency of, say, 133 lines per inch would be scanned at about 260 dpi. Similarly, if you are making a PDF file for printing on that same imagesetter, you could safely resample any excessively-high-resolution images to the same 260 pixels per inch.

Distiller allows you to specify the values of three parameters that affect resampling:

- *Threshold value* - This is the maximum final resolution an image may have in this document. Images with a resolution greater than this will be resampled. Images with resolutions below this value will be untouched.
- *Target resolution* - This is the resolution you would like the final image to have. The image will be resampled to something close to this target value.
- T*ype of resampling* - The pop-up menu lets you select among the three types of resampling we have discussed (or no resampling at all).



Next Page ->

Color & Grayscale Recommendations

The values you should specify for resampling color and grayscale images depend upon the purpose you have in mind for the PDF document.

Printed Documents

A document whose primary purpose is to be printed needs to retain enough image data for the target printer. In particular, you should try to match the traditional printer's rule of thumb: the image's resolution should be twice the halftone screen frequency that will be used to render the document. This yields the following recommended target resolutions:

- *For typical 600 dpi laser printers:* 200 dpi.
- *For imagesetters:* 300 dpi

Acrobat calculates a threshold value of about 1½ times the target resolution; I see no reason to change this.

On-Screen Documents

If the PDF file will be primarily read on a computer screen, then you should downsample images to match the resolution of a typical computer monitor:

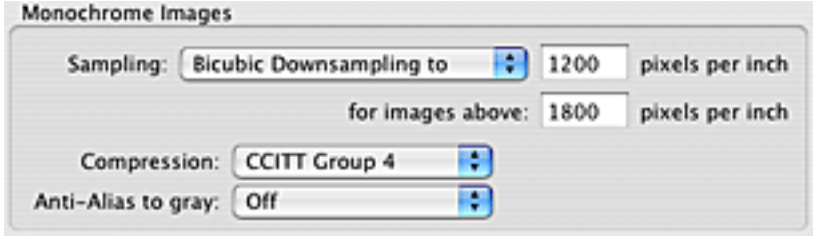
- *For on-screen:* 72 dpi

You may want to retain more data than this if you want to allow people to zoom in on your images and see more detail.

Next Page ->

Acumen Journal: Acrobat User 11

Monochrome Images



For 1-bit images (scanned line art and such), you want to match the resolution of the expected output device or monitor:

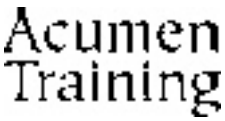
- *Laser printers:* 600 dpi
- *Imagesetters:* 2400 dpi
- *On-screen Display:* 72 dpi

Again, I wouldn't bother changing the threshold value; the default will server perfectly well.

Anti-Alias to gray

The Monochrome resampling controls include an additional pop-up menu that ask if you want Distiller to convert the image to grayscale before resampling. I strongly recommend this.

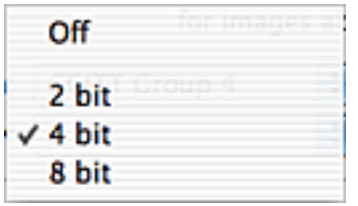
Monochrome images do not resample well. They become broken up and hopelessly jagged. At right are the words "Acumen Training," scanned at 300 dpi and then downsampled by Distiller to 72 dpi. You'll note that it doesn't look very good; I would not care to use this image as-is in any professional document.



Next Page ->

The *Anti-alias to gray* menu allows you to have Distiller convert your monochrome image to a grayscale image (that still initially has only blank and white pixels in it) before downsampling. Grayscale images downsample well, as you can see at right. This is the same 300 dpi monochrome image downsampled by Distiller with the grayscale conversion turned on.

The pop-up menu gives you three choices for the grayscale conversion: 2-, 4-, and 8-bit grayscale. The more bits per pixel, the better the resulting downsampled image will look. Practically speaking, I find that 4 bits per pixel is perfectly good; the resulting downsampled image is indistinguishable from the 8-bit equivalent.



Next Page ->

Final Thoughts

Don't overdo it

Keep in mind that when you resample an image, the discarded data is permanently gone. Don't be too aggressive in your downsampling, since there is no way to replace the excised data.

Resampling Can Improve Image Quality

Given the previous paragraph, it might seem reasonable to turn off resampling entirely; just retain all of the image data. The reasoning is that by retaining all of the data, the image will look as good as possible, even though the resulting PDF file will be bigger than necessary.

Somewhat counterintuitively, this is not the case for a printed document; retaining all of the original image data can result in a printed image looking *worse* than the resampled image.

This will be true particularly if you are printing to a PostScript printer; if an image has more data than a PostScript printer can use, the printer will subsample the data, resulting in the loss of image quality we discussed above. If there's too much data in an image, then there's too much data; the printer needs to handle the situation somehow. PostScript chooses the method that is fastest (subsampling), rather than prettiest.

Having Distiller downsample your image data ahead of time yields better looking images. (That said, the effect will be hard to see on most printers.)

Return to Main Menu

Handling Errors in PostScript, Part 1

What's a "Back Channel"

The term "back channel" is often used in PostScript to refer to the *%stdout* output file. This is a writeable virtual file associated with the currently active communication port.

Anything written to %stdout (such as error messages) will either be written back to the computer that is the source of the PostScript stream or appear in the interpreter's log file, if it has one.

What exactly happens when a PostScript error takes place? If you send the interpreter a piece of code like:

```
1 2 3 O'Leary
```

most PostScript devices will send an error message to the *back channel* (see sidebar). Typically, this message consists of the name of the error and the offending command (that is, the object that was being executed when the error occurred). Acrobat Distiller also prints the contents of the operand stack:

```
%%[ Error: undefined; OffendingCommand: O'Leary ]%%
Stack:
3
2
1
```

Doing it yourself

Sometimes it would be useful to handle errors yourself, perhaps to report more information about the error or to print the error message on paper.

This issue and next, we shall see how to override PostScript's default error handling.

In this issue, we shall see how to gather and report more information about the error. The next issue will discuss how to print the error message to the current page.

Next Page ->

Acumen Journal: PostScript Tech 15

Review: PostScript Error Handling

Server Loop

PostScript error handling is built upon the fact that the Server Loop executes your PostScript job using the *stopped* operator.

The stopped Operator

You should remember *stopped* from your PostScript class; it allows you to implement something vaguely similar to other languages' *try* and *catch*.

```
-object-   stopped   =>   bool
```

This operator takes an executable file, string, or procedure body from the operand stack and executes it. When that executable object returns, execution drops back to the *stopped* operator, which returns a boolean on the stack.

This boolean will be *false* if the executable object ran all the way to the end without a PostScript error. It will be *true* if the execution ended early because of a PostScript error. (There is another operator, *stop,* that actually causes the break, but I'll let you look that up in your student notes.)

Next Page ->>

Executing Your Job

Here is a simple version of what the Server Loop does when it sees a PostScript stream appearing at one of a device's communication ports:

```
(%stdin) (r) file                  % Open a line to stdin
cvx stopped                        % Make it executable & execute it
{ errordict /handleerror get % If err: get handleerror...
   exec                            % ...and execute it
} if
```

Let's look at this in detail (because we need to remember this thoroughly to see how to modify it):

`(%stdin) (r) file`

Here the server loop opens the predefined file *%stdin*. This is a virtual, read-only file associated with the currently active communication port. When your program reads from *%stdin,* it actually reads data from your incoming PostScript stream.

Remember that the *file* operator returns on the operand stack a file object representing the newly-opened file, in this case representing our active communication port.

`cvx stopped`

The *cvx* operator ("convert to executable") makes the file object executable.

The *stopped* operator then executes the file object, that is, it moves the object to the execution stack. (This file object becomes the PostScript job's "input stream.")

Next Page ->

Acumen Journal: PostScript Tech 17

The *stopped* will return a boolean, *true* if there was a PostScript error. In this case, execution will have dropped out of the PostScript stream, but the error will not yet have been reported to the user.

```
{ ... } if
```

The *if* clause will be executed if *stopped* returns *true,* that is, if there was a PostScript error.

```
errordict /handleerror get exec
```

Inside the *if* clause, the Server Loop fetches, from a predefined dictionary named *errordict,* a procedure named *handleerror;* it then executes that procedure.

The *errordict* dictionary contains all of the procedures that are used by the PostScript error handling mechanism. The *handleerror* procedure within that dictionary is the error-reporting routine; this is the procedure that actually generates the error message and sends it to the back channel.

$error

The *handleerror* procedure must somehow determine the name of the error, the offending command, and what was on the operand stack when the error took place. It gets this information from another predefined dictionary, named *$error.*

The *$error* dictionary contains information about the most recent PostScript error. Its seven key-value pairs supply an error reporting procedure everything that is to be known about the error. Specifically, the contents of *$error* are as follows:

Next Page ->

Acumen Journal: PostScript Tech 18

errorname (*name*) The name of the PostScript error. This will be one of the classic PostScript error names, such as *typecheck* or *stackoverflow*.

$error Contents

errorname	*name*	Name of the PostScript error
command	*any obj*	The offending command
ostack	*array*	Contents of operand stack
dstack	*array*	Contents of dictionary stack
estack	*array*	Contents of execution stack
ErrorInfo	*array*	Misc. error information
newerror	*bool*	Indicates new information

command *(any object)* The offending command. Note that this is not the *name* of the offending command, but the command, itself. That is, if a call to *moveto* caused the error, the value of *command* will be the operator object for *moveto,* not the name "moveto."

ostack, *dstack*, *estack* *(array)* These three arrays contain the complete contents of the operand, dictionary, and execution stacks as they were when the error took place. Element *0* of each array was the bottom of the stack.

ErrorInfo *(array)* This array contains supplemental information regarding the error. What exactly will appear in this array is hard to predict; it is frequently empty. Anything that *does* appear in this array is worth examining, however.

newerror *(Boolean)* This Boolean value indicates the information in *$error* has not been reported to the user. *Handleerror* checks this to make sure that an error has actually occured and that it hasn't been called by accident.

Next Page ->

Acumen Journal: PostScript Tech 19

Writing Your Own *handleerror*

The most common reason for overriding PostScript's default error handling mechanism is to change how errors are reported, typically to either display more information about the error or to print the error message on paper. The easiest way to do this is to write your own *handleerror* procedure, defined in *errordict:*

```
errordict begin      % Place errordict on the dictionary stack
/handleerror         % Define a procedure named handleerror
{
    ... error handling stuff
} bind def
end                  % Remove errordict from the dict stack again
```

Now, when a PostScript error occurs and the Server Loop goes to *errordict* and executes *handleerror,* it will get your *handleerror,* rather than the default.

By the time your *handleerror* is executed, *$error* will have been loaded up with its information describing the error. You can do anything you wish to report this information to the user.

Next Page ->>

A Skeletal *handleerror*

A typical definition of *handleerror* would look like this, in outline:

```
/handleerror
{
    $error begin      % Move $error to the dict stack
    newerror {        % Is neweerror true?
        ...           % Yes: report error...
        error reporting stuff
        ...
        /newerror false def  % ...and reset newerror
    } if
    end               % Remove $error from the dict stack
} bind def
end
```

Step by step

Here's what the above skeleton does:

```
/handleerror
{
    $error begin      % Move $error to the dict stack
```

Our *handleerror* will start by placing *$error* on the dictionary stack; PostScript will now automatically search this dictionary, in addition to *userdict, globaldict,* and *systemdict.*

Next Page ->

Acumen Journal: PostScript Tech 21

```
newerror    {
    ...
} if
```

We shall use the *if* operator to conditionally execute our error reporting code. We want to report on the contents of *$error* only if *newerror* is true, indicating we have a new error. (It is possible for the Server Loop to execute *handleerror* in circumstances other than a PostScript error; our *handleerror* should not report these instances as an error.)

```
newerror {
    ...
    ... error reporting code goes here
    ...
    /newerror true def
} if
```

Presuming *newerror* is *true,* we shall report the error to the user, using the information stored in *$error.* (We'll look at an example of some error reporting code in a moment.) One thing you must always do: set *newerror* to *false* when you are done reporting the error. Now that the error has been reported to the user, it is by definition no longer a new error.

```
    end
} bind def
```

Finally, our *handleerror* should finish by removing *$error* from the dictionary stack with a call to *end.*

Next Page ->

Acumen Journal: PostScript Tech 22

A Distiller Annoyance

Acrobat Distiller does something unexpected, given what we have just discussed: its Server Loop calls a procedure named *dhandleerror* (presumably "Distiller handleerror"), rather than *handleerror*. To allow for the case (common for me) where you are sending your error handler to Distiller, you should include the following definition of *dhandleerror* immediately after the definition of *handleerror:*

```
/dhandleerror /handleerror load def
```

This will cause *dhandleerror* to call the same procedure as our *handleerror*.

A Basic Error Handler

Let's add some code that will actually emit an error message. The next page presents a full, if simple, definition of *handleerror,* followed by some flawed PostScript code whose resulting error we shall report.

Next Page ->>

A Basic Error Handler

On the website

As usual, this file can be found among the PostScript examples on the Acumen Training website's resources page. Look for the file *BasicErrorHandler.ps*.

```
errordict begin                 % Put errordict on the dict stack
/handleerror                    % Define handleerror:
{   $error begin                % Put $error on the dict stack
    newerror {                  % Is neweerror true?
        (*** PostScript Error ***) =    % Yes: emit error msg
        (Error: ) print errorname =     % Print error name
        (Offending command: ) print /command load ==
        (Operand stack: ) =     % Print a label
        clear                   % Clear the operand stack
        ostack aload pop        % Unload ostack
        count {                 % Start of repeat loop
            (\t) print ==       % Print item to %stdout
        } repeat                % Repeat for each item
        flush                   % Flush %stdout
        /newerror false def     % Reset newerror
    } if
    end                         % Remove $error from the dict stack
} bind def                      % End of handleerror definition

/dhandleerror /handleerror load def     % accommodate Distiller
end                             % Remove errordict from dict stack
% Now let's try the new error handler
1 2 /3 (O'Leary) moveto         % moveto will yield a typecheck error
```

Next Page ->

Acumen Journal: PostScript Tech 24

The execution of this program yields the following error message sent to stdout:

```
*** PostScript Error ***
Error: undefined
Offending command: --moveto--
Operand stack:
    (O'Leary)
    /3
    2
    1
```

The error reporting code makes use of three PostScript operators you may not have encountered. (We don't talk about them in most of the PostScript classes)

=, ==, *and print*

Our *handleerror* uses three operators that emit text to stdout:

= The "equal" operator prints the value of the object on top of the operand stack to %stdout; the object is removed from the operand stack. The text sent to stdout is followed by a newline. Thus:

```
(Howdy) =
```

prints the text

```
Howdy
```

to %stdout.

Next Page ->

Acumen Journal: PostScript Tech 25

==

"Double-equal" prints the value of the topmost item to %stdout. It differs from = in two ways:

- It prints an indication of the nature of the data.
- It can print the contents of arrays and procedure bodies.

Like =, double-equal emits a newline to %stdout.

For example,

```
(Howdy) =
```

prints the text

```
(Howdy)
```

to stdout. (Note the parentheses in the outuput, indicating this is a string.)

print

The *print* operator takes a string (and only a string) as its argument, printing the contents of that string to %stdout. Unlike the two equal operators, *print* does not emit a newline.

Thus,

```
(The string is )print (Howdy) =
```

prints the text

```
The string is Howdy
```

to %stdout.

Next Page ->

Acumen Journal: PostScript Tech 26

Now let's see how our *handleeror* uses these to generate an error message. Here's our definition with the parts we've already discussed grayed out:

```
/dhandleerror
{    $error begin
     newerror {
          (*** PostScript Error ***) =
          (Error: ) print errorname =
          (Offending command: ) print /command load ==
          (Operand stack: ) =
          clear
          ostack aload pop
          count {
               (\t) print ==
          } repeat
          flush
          /newerror false def
     } if
     end
} bind def
```

Step by Step

`(*** PostScript Error ***) =`

We begin by printing an alert line; this simply catches the user's eye and, we hope, makes them immediately aware that there has been an error.

Next Page ->

Acumen Journal: PostScript Tech 27

```
(Error: ) print errorname =
```

We print the label "Error:" followed by the value of the *errorname* entry in *$error*. (Remember that *$error* is on the dictionary stack at this point, so we have access to all of its key-value pairs by simply referring to their keys.)

```
(Offending command: ) print /command load ==
```

Now we print the offending command. Note that in this case we do *not* just refer to *command,* but instead do an explicite *load* on that name. We need to do this because the *command* entry in *$error* will be associated with whatever object was being executed when the error took place. In our example, the offending command will be the operator object that implements *moveto;* if we simply invoked the name *command,* the interpreter would immediately execute the *moveto* operator, giving us another error. (Come to think of it, that would recursively call our new *handleerror.)*

The *load* operator will place the offending command on the operand stack and then we can let the double-equal operator display it. (Happily, double-equal does the appropriate name lookup on operator definitions.)

```
(Operand stack: ) =
clear
ostack aload pop
```

Now we are going to print the contents of the *ostack* array. We begin by clearing the operand stack and then unloading the contents of *ostack* onto the operand stack. We do this with another operator you may not have encountered: *aload.*

Next Page ->

Acumen Journal: PostScript Tech 28

```
[ obj0 obj1 ... ] aload   =>   obj0 obj1 ... [ array ]
```

The *aload* operator takes an array as its argument and "unpacks" it on the operand stack; precisely, it places all of the contents of that array on the stack and then pushes the array itself back onto the stack.

In our code, I had no use for this copy of the array, so I just popped it off.

```
count {
```

We are now going to print each of the items we just placed on the operand stack. We shall use a *repeat* loop to do this, executing *count* to determine how many times our loop must repeat. (Remember *count* returns on the stack the number of items on the stack.)

```
    (\t) print ==
} repeat
```

For each item in the loop, I do two things:

- Print a tab to %stdout.
- Print the object on top of the stack with == (which removes the object from the stack).

```
flush
```

Finally, we execute *flush*. This operator flushes the output buffer associated with %stdout, ensuring that the text we have been writing actually makes it to the log file or host computer.

Next Page ->

Acumen Journal: PostScript Tech 29

Whew! At this point, we have an error handler that behaves like Distiller's default error handler: it reports the error's name, the offending command, and the contents of the operand stack.

Now what? In the next issue, we shall add to our error handler, printing the contents of the dictionary stack and the *ErrorInfo* array. We'll also see how to print all of this stuff to an error page, accommodating systems that don't receive error messages back from the printer.

See you next time.

Return to Main Menu

Schedule of Classes, April - July 2005

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

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

Technical Classes

PDF File Content and Structure	Apr 18–21		
PostScript Foundations			July 18–22
Variable Data PostScript			
Advanced PostScript			
PostScript for Support Engineers		May 23–27	
Jaws Development		*On-site only*	

Course Fee The PostScript and PDF classes cost $2,000 per student.

Registration Info

Acrobat Class Schedule

These classes are taught occasionally 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 — *No Acrobat classes scheduled for this quarter.* See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Acrobat Class Fees — *Acrobat Essentials* and *Creating Acrobat Forms* (½-day each) cost $180.00 or $340.00 for both classes. 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: 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: http://www.acumentraining.com/registration.html

email: registration@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Back issues

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

Return to First Page

What's New at Acumen Training?

New PDF Class

Nothing new this month. I am laying out and researching topics for the second PDF File Content and Structure class. I am expecting to teach the first class in late September. The topic list is below; as before, if you think something should be added to or dropped from this list, send an email to john@acumentraining.com.

Preliminary Topic List

- Overprinting
- CID Fonts
- Halftones
- Marked Content
- Rendering Intents
- Smooth shading
- Reference XObjects
- Cross reference streams
- BX & EX
- File Spec
- Masked Images
- Digital Signatures
- AcroForm
- Transfer Functions
- Shape dictionaries
- Layers
- Name Dictionaries
- Patterns
- Composite Fonts
- Linearized PDF
- Stroke Adjustment
- Halftones
- Text Knockout
- Object streams
- More on data structures

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 remember fondly your last root canal?

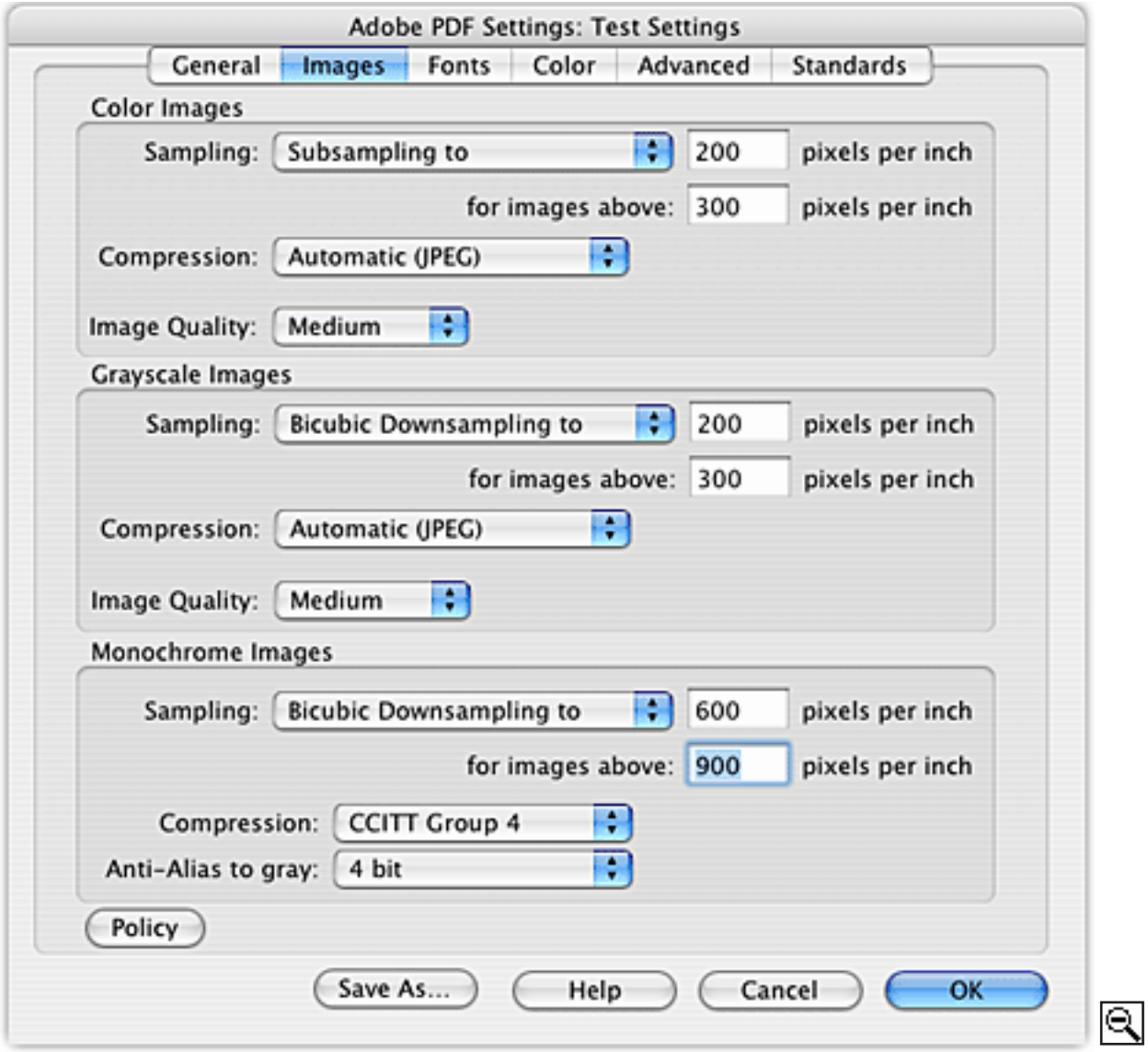
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

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


Adobe PDF Settings: Test Settings
General
Images
Fonts
Color
Advanced
Standards
Color Images
Sampling: Subsampling to 200 pixels per inch
for images above: 300 pixels per inch
Compression: Automatic (JPEG)
Image Quality: Medium
Grayscale Images
Sampling: Bicubic Downsampling to 200 pixels per inch
for images above: 300 pixels per inch
Compression: Automatic (JPEG)
Image Quality: Medium
Monochrome Images
Sampling: Bicubic Downsampling to 600 pixels per inch
for images above: 900 pixels per inch
Compression: CCITT Group 4
Anti-Alias to gray: 4 bit
Policy
Save As...
Help
Cancel
OK