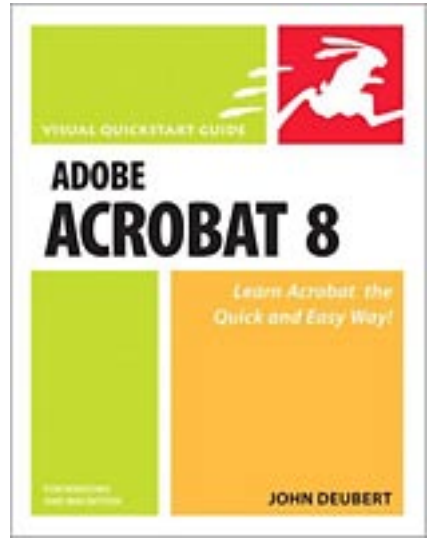
Table of Contents



Customizing Acrobat 8 Toolbars

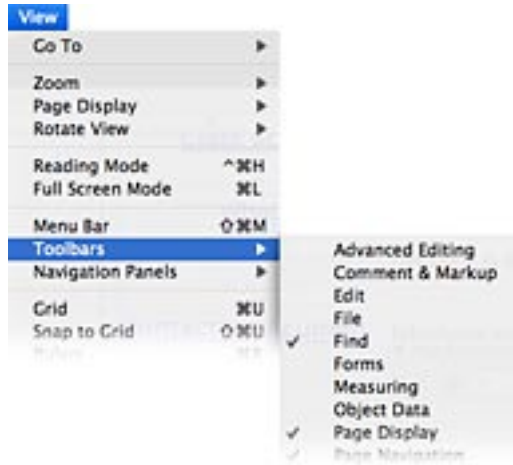
I like Acrobat 8 better than I did its previous two predecessors. Many of the more sophisticated features of Acrobats 6 and 7 (digital signatures, group reviews, etc.) were hard for unsophisticated users to work with. In some cases, there was no reasonable way to work with these features without having an IT department set up server processes and do other background chores.

Acrobat 8 has addressed these problems and is one of the best versions of Acrobat I have seen in a while. In writing the Acrobat 8 Visual Quickstart Guide (of which you need to buy many copies), I had a good opportunity to learn the workings of Acrobat 8 in detail.

The next few Journal articles will look at some of the new features and the improvements in some of the old features. We are going to start this month with a short article on a simple, yet powerful, feature: the ability to modify the contents of Acrobat's toolbars.

Next Page ->

Acrobat 8 Toolbars



Like the previous two versions of Acrobat, the new version has a very large number of toolbars for a variety of special purposes. I didn't like this at first, but now I've come to see this as a benefit; Acrobat has such a very large set of abilities that it is useful to be able to make visible only those toolbars you need at a particular moment. When I am annotating a PDF file, I make the Comment & Markup toolbar visible; otherwise, I leave it hidden and out of the way.

As in the past, you can choose which toolbars should be visible using the *View* > *Toolbars* submenu, selecting those toolbars that you want to be able to see.

A major UI difference between Acrobat 8 and previous versions is that the toolbars are now attached to the top of each document window, as at right. I'm not clear as to the intended benefit of this UI feature; it seems to use up a lot of window area that could otherwise be used to display the document.

On the positive side, Acrobat 8 allows you to tailor the contents of the individual toolbars so that you need only see the controls you are likely to use within each toolbar. This is our topic for the month.

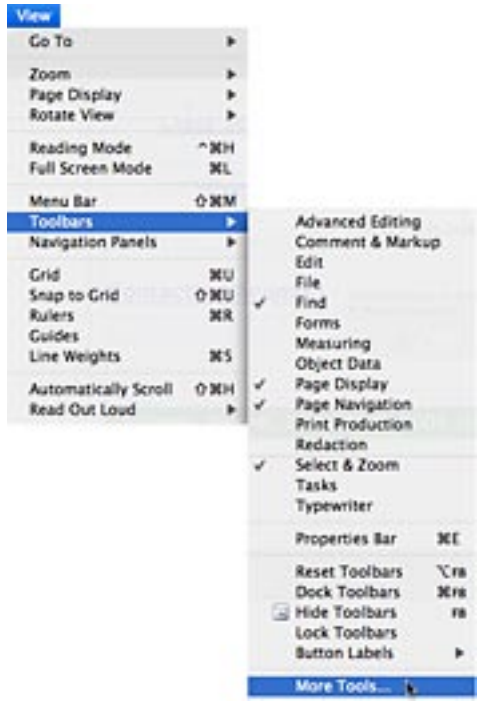
Next Page ->

Customizing the Page Navigation Toolbar

Let's see how to customize a toolbar by adding tools to one of the most-used toolbars: Page Navigation. This toolbar has the controls you routinely use to move around within your document. The default Page Navigation toolbar is rather skimpy, as at right. It contains only a pair of arrows for moving to the next and previous pages and a text field into which you may type a page number, moving you to that page in the document.

Where are the *First Page* and *Last Page* buttons? The *Next* and *Previous View* buttons (which I find indispensible)? They exist in Acrobat 8, but don't by default reside in the Page Navigation (or any other) toolbar; we need to add them.

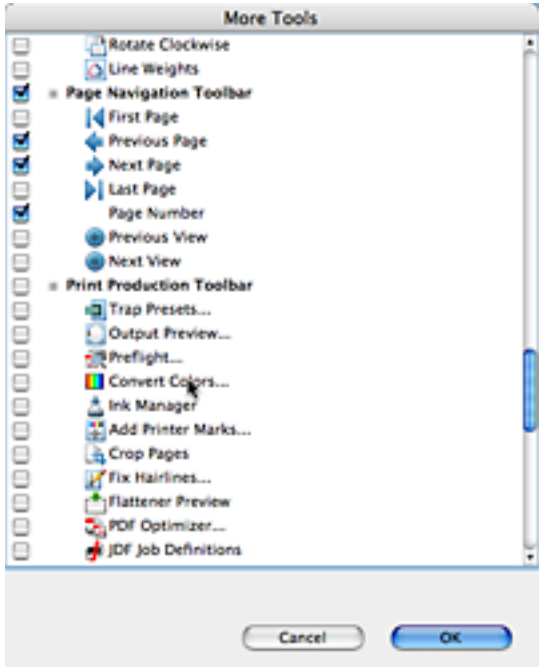
We do this by selecting *View* > *Toolbars* > *More Tools*. Acrobat 8 will display the *More Tools* dialog box (next page).



Next Page ->>

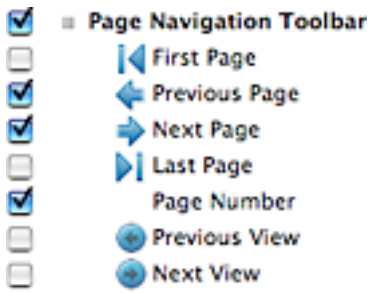
More Tools Dialog Box

The *More Tools* dialog box displays a hierarchical list of all the toolbars contained in Acrobat and the items that can appear in each toolbar.



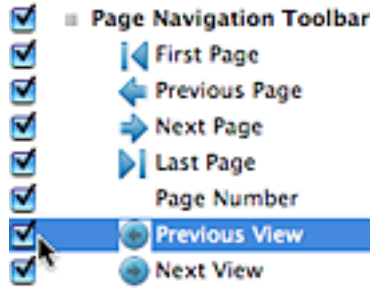
Each item in the list has a checkbox next to it; if the checkbox is selected, that item is visible, otherwise, it's hidden. It's exactly as simply as that! Just "turn on" the checkboxes corresponding to items you want to appear in a toolbar (and turn off those you don't want).

For example, the entries for the *Page navigation* toolbar initially look like the illustration at right; only three of the seven entries are selected, indicating the default contents of the toolbar. (Note there is also a checkbox for the entire Page Navigation Toolbar, indicating whether the toolbar itself should be visible.)



Next Page ->

To modify the toolbar's content, simply select the items you want to reside in the toolbar and uncheck those you don't want. For the *Page Navigation* toolbar, I select everything; I use all those buttons.



When you exit from the *More Tools* dialog box, the changes you made will be applied to the toolbars; the *Page Navigation* toolbar as I use it looks like the illustration, below right.

Toolbar Thoughts

I admit to being of two minds about the multiplication of toolbars that Adobe introduced with Acrobat 6. I've always been a minimalist with regard to toolbars; I don't like buttons cluttering up my screen unless I frequently use the corresponding commands *and* they are inconvenient to invoke using the keyboard or menus (perhaps because the commands are buried three deep in the menu hierarchy).

On the other hand, when we were doing final edits on the Acrobat 8 Visual Quickstart Guide (did I mention the new book?), it was very convenient to be able to make all of the comment and markup tools available for use in a toolbar (specifically, the *Comment & Markup* toolbar) and then have them all go away again (by unselecting the toolbar in the *View* > *Toolbars* menu) when I was done.

In this imperfect world, it is probably not bad to have a large number of toolbars, most of which I can leave invisible. And to be able to control the contents of the toolbars makes me much happier; I can be as minimalist or lavish with toolbar commands as I wish.

Return to Main Menu

Extracting Text From a PostScript File

One question that recurs on the PostScript newsgroup is how to extract text from a PostScript file. That is, given a PostScript program that prints pages of text, how could one write that text (and only the text) to a separate file?

There are presumably several ways to do this; I wouldn't doubt that among GhostScript's myriad command line flags, there is one that cause the text operators to dump their arguments to *stdout*. Doing it entirely within PostScript, however, the technique that seems most obvious is to redefine the PostScript *show* operator and its variants so that they send their string arguments to some convenient destination on the RIP's hard disk.

That is what we shall do this month.

Next Page ->>

Redefining Operators

Part of PostScript's charm is that you can redefine its operators, allowing you to change the behavior of existing PostScript code. A perfectly respectable PostScript program that prints a block of text, such as the following:

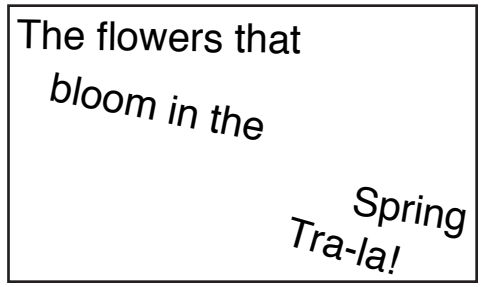
```
/xys     % (str) x y   =>   ---
{ moveto show } bind def

/Helvetica 15 selectfont
(The flowers that) 72 700 xys
(bloom in the) 72 682 xys
(Spring) 72 664 xys
(Tra-la!) 72 646 xys
```

can be made much more creative (in the sense of "unreadable") by adding the following definition to its beginning:

```
/moveto { moveto rand 20 mod 10 sub rotate } bind def
```

Now, when the *xys* procedure calls *moveto,* it will call *our moveto,* which does a random rotation each time it's called.



"But, wait!" I hear you say. "Doesn't our *moveto* procedure make a recursive call to itself?"

Nope. Remember from your PostScript class that the *bind* operator replaces the executable name *moveto* in the

Next Page ->

Acumen Journal: PostScript Tech 8

procedure body with the operator *definition* of *moveto*. There is no name to be looked up at execution time. As a corollary, the *bind* is vital to the redefinition; without it, the *moveto* procedure would repeatedly call itself and our program would die with an *execstackoverflow* error.

Extracting Text

What we are going to do is redefine *show* and all of its variants so that they send the contents of their string arguments to stdout. The PostScript program on which we shall test our redefinitions is this:

```
/LM 72 def
/LnHt 17 def

/nl
{ //LM currentpoint //LnHt sub exch pop moveto } bind def

/snl     % (str) => ---
{ show nl } bind def

/dh      % dx => ---
{ 0 rmoveto } bind def


/Helvetica 15 selectfont
72 700 moveto
```

Next Page ->

```
(It is better to keep your mouth closed) snl
(and let people think you ar)show -.5 dh
(e a fool) snl
(than to open it) snl
(and remo)show -.37 dh
(v) show -.37 dh
(e all doubt) snl


showpage
```

It is better to keep your mouth closed
and let people think you are a fool
than to open it
and remove all doubt

There is nothing too unusual here. We define a few utility procedures, which we then use to print four lines of text.

Note that the a number of these lines of text are printed with multiple calls to *show;* these represent lines with kerning pairs in them, pairs of characters whose spacing needs to be adjusted because they will look too far apart if printed with their bounding boxes abutting, as normal. This will present a problem for us later, as we shall see.

Next Page ->

First *show* redefinition

The simplest redefinition of *show* we can use to extract text simply sends its string argument to stdout by calling the = procedure:

```
/show /= load def
```

If we place this definition at the start of our earlier program, what gets sent to stdout is a series of newline-delimited lines of text:

```
It is better to keep your mouth closed
and let people think you ar
e a fool
than to open it
and remo
v
e all doubt
```

It is immediately evident that we are going to need to address the kerning pairs sooner rather than later. Each of the calls to *show* in the kerning pairs generates a fresh line of text to stdout. We can go through and stitch them together by hand, but that becomes boring after a very short while.

Next Page ->>

Second *show* Redefinition

To fix this, we need a more sophisticated redefinition of *show*. Our new *show* will compare the current *y* value to the previous text snippet's *y;* if they are different, we shall emit a newline before sending the new text to stdout; if they are the same then we shall take the new text to be a continuation of the previous and send it to stdout without the delimiting newline.

As a first approximation, here's our new *show* redefinition (and some ancillary definitions):

```
/lastY 0 def                    % Previous text's y position
/yTolerance .1 def              % Tolerance for determining same line

/show
{    currentpoint exch pop      % Get the current y position
     dup lastY sub abs          % Subtract from previous y value
     yTolerance gt              % Compare to our tolerance value
     { (\n) print               % If greater, emit newline...
       /lastY exch def }        % ...and update lastY
     { pop }                    % Otherwise, discard the y value
     ifelse
     print                      % Send text to stdout
} bind def
```

Next Page ->>

Now when we prepend this to our earlier PostScript example, we get the following text to stdout:

```
It is better to keep your mouth closed
and let people think you are a fool
than to open it
and remove all doubt
```

Our new *show* redefinition successfully knits together the kerned text pieces into single lines of text.

Let's look at this in detail.

Step by step

```
/lastY 0 def
/yTolerance .1 def
```

We start by defining a couple of variables. *lastY* will hold the previous text's *y* value; we initialize it to zero because we have to choose something. *yTolerance* is a threshold value indicating that the current piece of text is a different line than the previous; if the difference between the *y* positions of the current text and the previous text is greater than this value, we shall emit a newline, separating the two pieces of text. A difference of *0.1* seems like a reasonable value for this variable.

```
/show
{   currentpoint exch pop      % stack: (str)  y
```

Our *show* redefinition starts by getting the current *y* value.

Next Page ->

Acumen Journal: PostScript Tech 13

```
dup lastY sub abs                    % (str) y Δy
```

We duplicate the *y* value and then subtract it from *lastY,* converting the difference to a positive value.

```
yTolerance gt                % (str) y bool
```

We compare the difference to our *yTolerance* value, leaving a Boolean value on the stack that will be *true* if the difference is greater than the threshold value.

```
{ (\n) print /lastY exch def }     % (str)
```

If the Boolean is *true,* we print a newline to stdout and then redefine *lastY* to be our current *y* value.

```
{ pop }                      % (str)
```

Otherwise, we discard the *y* value.

```
ifelse                       % (str)
```

This is the *ifelse* that selects between the two alternative procedure bodies. Both of the conditionally-executed procedures leave the original string—the *show* operator's original argument—on the stack.

```
    print
} bind def
```

Our *show* ends by sending the string's contents to stdout using the *print* operator. Remember that *print,* unlike the similar = operator, doesn't follow its output with a newline.

Next Page ->

Acumen Journal: PostScript Tech 14

Third *show*

Our previous redefinition is fine as a first approximation, but it has a glaring weakness: it is doing all of its *y*-value-related calculations in User Space; a threshold of 0.1 is a reasonable value for default User Space, but what if the text is being set in a coordinate system that has been heavily scaled? We cannot know the physical distance to which our threshold of *0.1* really corresponds.

So what to do?

A reasonable solution would be to do our *y* calculation and comparison in Device Space. If we convert the current point coordinates to Device Space, we do not need to care about the particular User Space values that got us to that position.

This is very easily achieved; here is our new redefinition code:

```
/lastY 0 def
/yTolerance 2 def

/show
{
     currentpoint transform exch pop
     dup lastY sub abs yTolerance gt
     { (\n) print /lastY exch def }
     { pop }
     ifelse
     print
} bind def
```

Next Page ->

Acumen Journal: PostScript Tech 15

Step by step

The changes we have made are these:

```
/yTolerance 2 def
```

Since we are going to be converting our coordinates to Device Space, our threshold value must now be also expressed in Device Space. On a 600-dpi device, a threshold of *2* corresponds to $^1/_{300}$-point, which seems adequate.

```
currentpoint transform exch pop
```

We have added a call to *transform* to our *show* redefinition. This operator, you may recall, converts a User Space *x,y* pair to Device Space:

transform % stack: x_{usr} y_{usr} => x_{dvc} y_{dvc}

Thus, the *exch pop* leaves the current Device Space *y* value on the stack.

So now we don't need to worry about what aberrant changes the "host" PostScript code makes to User Space in printing its text.

Not Quite Done Yet

There are still two weaknesses we need to address.

Rotation issues

Firstly, our current redefinition can still be fooled by *rotate*. For example, if the text is being printed with a rotation of 90°, then two successive *show* calls that represent the same line of text will have the same *x* coordinate, but differing *y* values, exactly the opposite of our redefinition's assumption.

Next Page ->

We B Rotated

The easiest way to fix this is to simply kill the *rotate* operator:

```
/rotate /pop load def
```

Now *rotate* discards its argument and has no effect on the User Space and, therefore, the orientation of our text. This means the text will come out looking wrong on the page, but we don't intend to actually print this PostScript code, anyway.

If you are feeling a bit paranoid, you might also put a pin through any other operator that can be used to rotate the coordinate system:

```
/concat /pop load def
/setmatrix pop load def
```

Device Space Variations

Finally, and a bit more subtly, our redefinition also assumes that vertical movement on the page converts into a change in *y* in Device Space. This is not necessarily the case. In most devices, the direction of the Device Space *y* axis matches the direction of paper movement; a printer whose paper feeds long-edge-first will likely have vertical movement convert into motion along the Device Space *x* axis, invalidating our check of whether two successive *show* calls represent the same line of text.

The simplest fix for this is to force a known Device Space on the printer. This will certainly not match the real characteristics of the printing device, but, again, we don't plan on actually printing this file, so it doesn't matter.

Next Page ->>

We override the interpreter's default Device Space with a call to *setmatrix:*

```
[ 4 0 0 -4 0 0 ] setmatrix
```

The above line of PostScript sets the interpreter's current transformation matrix to something consistent with a 288 dpi device. (Review your PostScript Foundations notes to see how the CTM contents relate to a device's resolution.) Again, it doesn't at all matter what the resolution or other characteristics of the device actually are; we just need something that will give us consistent results across all devices.

While we're about it, we should disable any operators that change or initialize the CTM. This should certainly include *setpagedevice, initgraphics,* and *initmatrix;* depending on our level of paranoia, it cound also operators such as *letter, legal, a4, initigraphics,* and *initmatrix:*

```
/setpagedevice /pop load def
/initgraphics /pop load def
/initmatrix { } def
/setmatrix /pop load def
```

Show Variants

Finally, we should redefine not only the *show* operator, but also all of its variants:

```
/ashow { show pop pop } bind def
/widthshow { show pop pop pop } bind def
/awidthshow { show pop pop pop pop pop } bind def
/xshow { pop show } bind def
```

The above are the most important redefinitions; you may want to add *xyshow, kshow,* and the others to the list.

Next Page ->

By the way, these redefinitions are using our new *show* definition, so they must follow the latter in your code.

Final Versions

Having killed the *rotate operator,* set our CTM to something dependable, and hunted down every last *show* variant, our complete set of PostScript redefinitions looks like this:

On the Web Site

This PostScript code is on the Acumen Training Resources page. Look for *ExtractText.ps*.

```
/lastY 0 def                          % Previous text's y-value
/yTolerance 2 def                     % Same-string y threshold
/rotate /pop load def                 % Prevent rotate
/setpagedevice /pop load def          % Kill setpagedevice
/initgraphics /pop load def           % Disable initgraphics
/initmatrix { } def                   % Throttle initmatrix

[ 4 0 0 4 0 0 ] setmatrix             % Set the CTM to a known value...
/setmatrix /pop load def              % ...& then mask setmatrix


/show                                 % The show redefinition
{    currentpoint transform exch pop % Get our current Dvc Space y
     dup lastY sub abs yTolerance gt % Compare to previous y
     { (\n) print /lastY exch def }  % Over threshold? Emit a newline
     { pop }                         % Otherwise, throw away the y value
     ifelse
     print                           % Send text to stdout
} bind def
```

Next Page ->

Acumen Journal: PostScript Tech 19

```
/ashow { show pop pop } bind def     % Redefine all the show variants.
/widthshow { show pop pop pop } bind def
/awidthshow { show pop pop pop pop pop } bind def
/xshow { pop show } bind def
```

Just paste this in front of any PostScript code and you should be able to extract the text.

Caveat I haven't tested this extensively. It seems to work well with the PostScript output that I had immediately at hand, but I don't doubt that there will be some PostScript programs that will defeat it. Feel free to modify this as you wish; if you come up with modifications that fix incompatibilities with some software's output, feel free to send them to me; I'll post the fix—properly credited—on the website alongside my original file.

Return to Main Menu

Schedule of Classes, January-March 2007

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 1	Jan 22–25		Mar 5–8
PDF File Content and Structure 2		Feb 5–8	
PostScript Foundations	Jan 15–19		Mar 19–23
Variable Data PostScript		Feb 12–16	
Advanced PostScript			Apr 9–12
PostScript for Support Engineers			Mar 12–15

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

Registration Info

Acrobat Class Schedule

I shall be presenting the *Acrobat for the Enterprise* class in March 2007.

Watch for it!

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

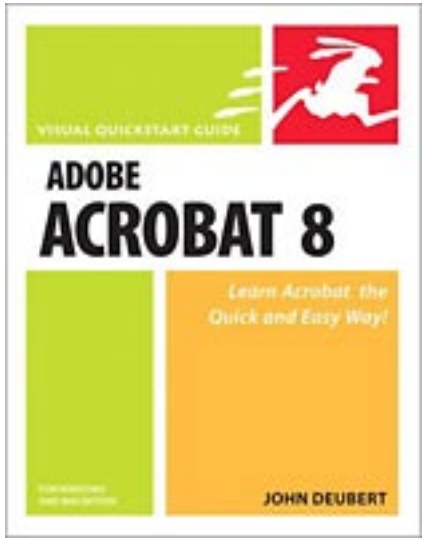
What's New at Acumen Training?

Acrobat 8 Visual Quickstart Guide

This should be available at Finer Bookstores Near You. This newly-written Visual Quickstart Guide steps you through all the important things you can do with Acrobat 8, covering everything from launching the application through conducting company-wide document reviews. The books introduces you to creating Acrobat forms, describes how to import a wide variety of images and other files into PDF, and steps you through the intricacies of digital signatures.

Buy several copies! The kids will love it for Christmas.

Trust me.



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 cause indigestion?

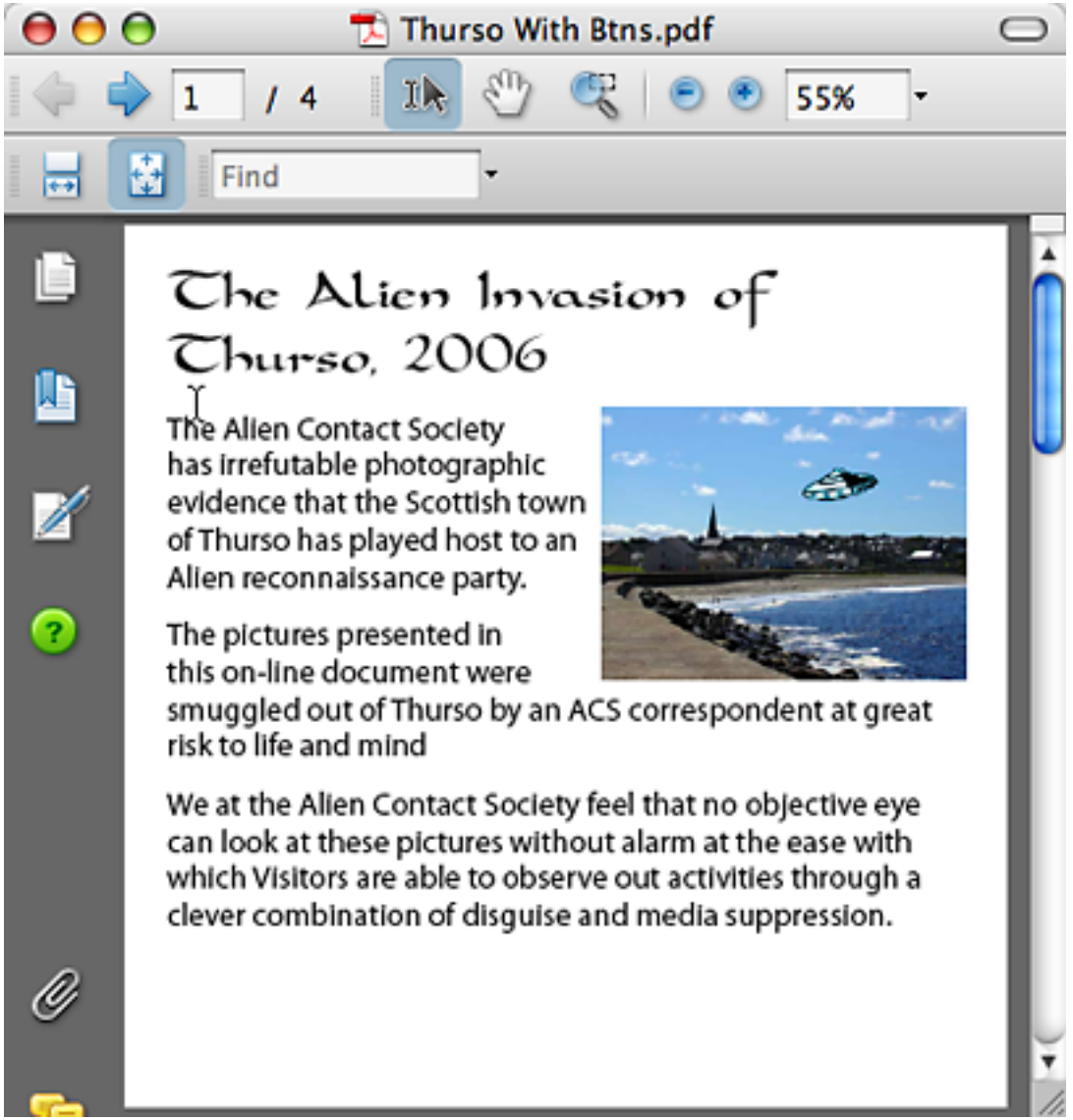
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

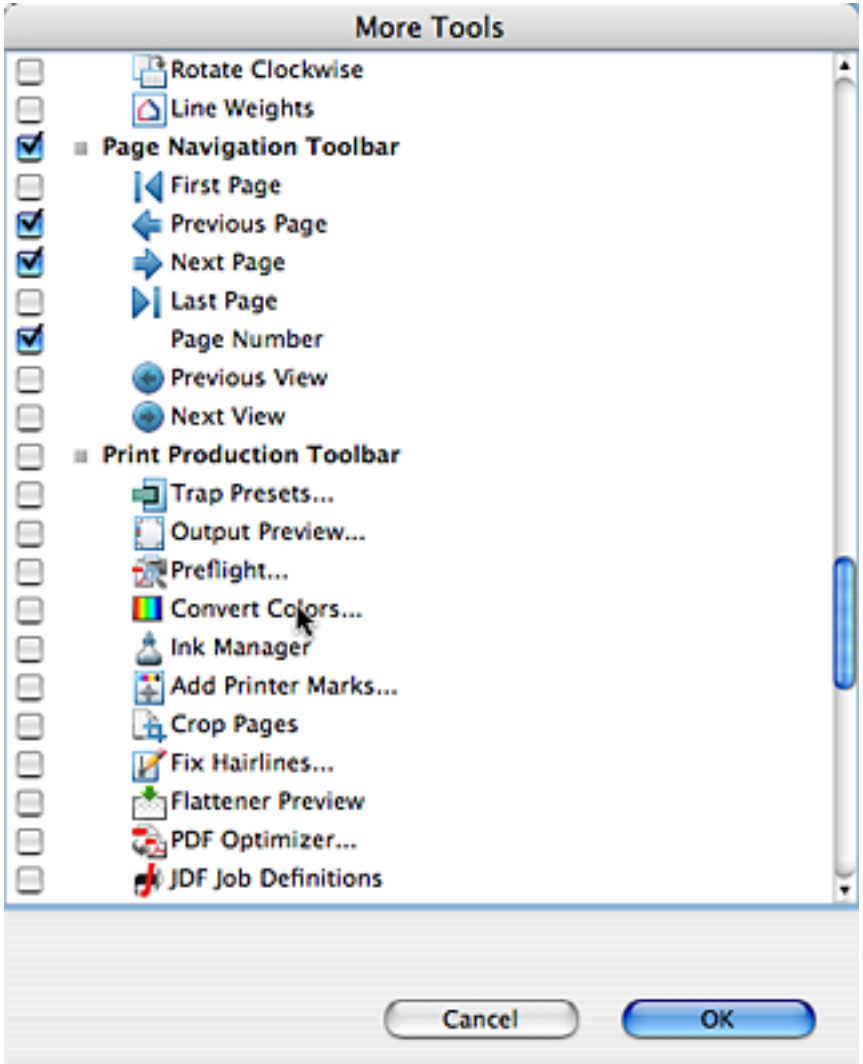
Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

Please send any comments, questions, or problems to:

journal@acumentraining.com

Return to Menu


Thurso With Btns.pdf
1
/ 4
55%
Find
The Alien Invasion of Thurso, 2006
The Alien Contact Society has irrefutable photographic evidence that the Scottish town of Thurso has played host to an Alien reconnaissance party.
The pictures presented in this on-line document were smuggled out of Thurso by an ACS correspondent at great risk to life and mind
We at the Alien Contact Society feel that no objective eye can look at these pictures without alarm at the ease with which Visitors are able to observe out activities through a clever combination of disguise and media suppression.


More Tools
Rotate Clockwise
Line Weights
Page Navigation Toolbar
First Page
Previous Page
Next Page
Last Page
Page Number
Previous View
Next View
Print Production Toolbar
Trap Presets...
Output Preview...
Preflight...
Convert Colors...
Ink Manager
Add Printer Marks...
Crop Pages
Fix Hairlines...
Flattener Preview
PDF Optimizer...
JDF Job Definitions
Cancel
OK