Table of Contents

The Acrobat User — **Acrobat 8's Typewriter Tool**
The Typewriter tool in Acrobat 8 seems initially to have no reason for existing. Once you realize what it's for, however, you find yourself using it more and more until it becomes one of your favorite tools. This issue's short article discusses how much better your life can become with this tool.

PostScript Tech — **Fun & Games With *pathforall***
One of PostScript's less-used operators is *pathforall*. This operator lets you step through the components of the current path, performing an operation on each step. This allows you to do some fun and occasionally very useful things.

Class Schedule — July, August, September

What's New? — **A new 2–day course: *Support Engineers' PDF***
Acumen Training's curriculum expansion continues with this two-day course on PDF for support engineers.

Contacting Acumen — Telephone number, email address, postal address

Journal feedback: suggestions for articles, questions, etc.

Fun & Games With *pathforall*

The *pathforall* operator allows you to step through the set of *moveto, lineto,* and other components that make up the current path and perform some operation on each of them. Though it is rarely used, there are some interesting things you can do with *pathforall*, though they are mostly more fun and interesting than useful.

This month, we shall look at how the operator works and see what it's good for.

The *pathforall* Operator

As you know, the *moveto, lineto,* and other path operators each adds a small wad of data to a PostScript internal structure called the *current path*. I usually think of this as a linked list of path elements: a *moveto* point, followed by a *lineto* point, followed by another *lineto* point, etc. The current path is actually composed of only four different types of path element: *moveto* points, *lineto* points, *curveto's,* and *closepath's*.

The *pathforall* operator traverses the list of path elements that makes up the current path, executing a particular procedure for element.

```
{ movetoProc } { linetoProc } { curvetoProc } { closepathProc } pathforall
```

The operator takes as its arguments a set of procedure bodies, each associated with a type of path element. For each element in the current path, *pathforall* will push one or more *x,y* pairs onto the operand stack and execute the procedure body associated with that type of path element. The arguments each procedure takes are:

movetoProc	The *x* and *y* coordinates of the moveto point.
linetoProc	The *x* and *y* coordinates of the lineto point.
curvetoProc	The *x* and *y* coordinates of the two control points and end point handed to the *curveto* operator.
closepathProc	No arguments.

Next Page ->

All of the above coordinates are expressed in *current* User Space. These may not be the same numbers that were originally handed to the path operators when the current path was created; if you changed User Space (with *scale, translate,* etc.) after creating the current path, the numbers handed to the *pathforall* procedures will reflect User Space as it is at *pathforall* time.

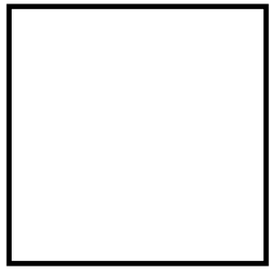
For Example

Let's use *pathforall* to mess around with the current path. The path we'll modify will be a simple rectangle; our original program—without *pathforall*—is as follows:

The Original Program

```
100 600 moveto
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath

2 setlinewidth
stroke
```



Pretty simple, but I like to start that way. The output from this program is reproduced above right, just in case you haven't seen a rectangle for a while. (Could happen.)

Add pathforall

Now let's add our call to *pathforall:*

```
100 600 moveto
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath
```

Next Page ->

Acumen Journal: PostScript Tech 3

```
{ 5 sub moveto }                          % moveto proc
{ 10 add exch 10 sub exch lineto }        % lineto proc
{ 20 sub curveto }                        % curveto proc
{ closepath }                             % closepath proc
pathforall

2 setlinewidth
stroke
```

Looking at the path element procedures, we see that they each modify the coordinates on the stack and then call their corresponding PostScript operator.

{ 5 sub moveto } The *moveto* procedure subtracts 5 from its *y* coordinate and then calls *moveto*.

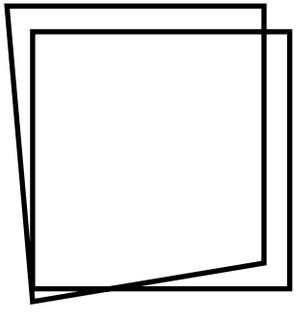
{ 10 add exch 10 sub exch lineto }
The *lineto* procedure adds 10 to the *y* value, subtracts 10 from *x*, and then calls *lineto*.

{ 20 sub curveto } The *curveto* procedure subtracts 20 from the *y* value and then calls *curveto*.

{ closepath } The *closepath* procedure takes no arguments, so it simply calls *closepath*.

This seems very easy and straightforward. The *pathforall* operator simply pushes the appropriate coordinate values on the operand stack and then executes the appropriate procedure. Our procedures modify the *x,y* pair on the stack (only one out of three pairs for the *curveto* procedure) and then execute the original PostScript operator.

When we stroke the resulting modified path, we imagine the result should be, well, the modified path.



However, looking at the output at right, we see that the stroked path is the original square *plus* the modified path. The *pathforall* operator doesn't erase the current path; the *movetos* and *linetos* we called were adding path elements to an existing path.

Next Page ->>

Replacing the Path

If we want to completely replace the original path, our *pathforall* call must do something very unusual-looking: our *pathforall* call must construct a procedure body that will construct our modified path. The new PostScript program must do the following:

1. Create the square current path.
2. Use *pathforall* to construct a procedure body that creates a new, modified current path. (We won't execute it quite yet.)
3. Destroy the current path.
4. Execute our create-a-path procedure body.
5. Stroke or otherwise paint it.

A Digression: Constructing a Procedure

We normally create procedure bodies with { braces }, which are the obvious, convenient way of doing so. However, procedure bodies are really just executable arrays; we can make a procedure by creating an array and making it executable with *cvx*.

Thus, the following two lines of PostScript create exactly the same procedure body:

```
{ 1 2 3 }
[ 1 2 3 ] cvx
```

It's a bit trickier if our procedure body must hold a call to a PostScript operator. We can't just write

[72 mul] cvx

Remember that items within between square brackets are operating in a normal PostScript environment. The executable name *mul* in the above line would be immediately executed; it would finding a mark object and the number *72* on the stack and return a *typecheck* error. This isn't what we had in mind.

Next Page ->

Acumen Journal: PostScript Tech 5

Instead we'll have to do the following:

```
[ 72 /mul cvx ] cvx
```

This curious-looking piece of code does the following:

- Push a mark object on the stack (that's what the open bracket does, remember).
- Push the number *72* on the stack.
- Push the literal name *mul* on the stack. (Since it's literal, the name is not immediately looked up and executed.)
- Convert the literal name *mul* to an executable name with *cvx*.
- Finish the array (with the close-square-bracket).
  - We now have on the stack an array that contains a number and an executable name.
- Convert the newly-made literal array to an executable array (that is, a procedure body) with *cvx*.

A bit more roundabout than { 72 mul }, but it gets the job done.

Back to pathforall

In order for our call to *pathforall* to actually replace the current path, rather than just add to it, we shall arrange for *pathforall* to actually create a procedure that constructs a new path. When the operator returns, we can delete the current path (with *newpath)* and then execute our new procedure.

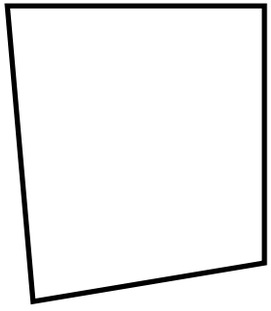
Here's the code:

```
100 600 moveto    % Create the square path
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath
```

Next Page ->

```
[                   % Push a mark onto the stack
    { 5 sub /moveto cvx }                          % moveto proc
    { 10 add exch 10 sub exch /lineto cvx }        % lineto proc
    { 20 sub /curveto cvx }                        % curveto proc
    { /closepath cvx }                             % closepath proc
    pathforall      % Call pathforall
] cvx               % Finish the array and make it executable
bind                % Bind it (out of habit)

newpath             % Erase the currentpath
exec                % Execute the newly-created procedure, creating a new path
2 setlinewidth
stroke              % Stroke the path
```



The new output looks like the path at right. Note that the original square is no longer present. We have completely replaced it with our new, modified path.

Step-by-step

Let's look at the call to *pathforall* in a little more detail:

```
[
```

We start with an open bracket that puts a mark object on the stack.

```
{ 5 sub /moveto cvx }
```

This is the *moveto* procedure. When *pathforall* executes this procedure, the *x* and *y* coordinates of the *moveto* point will be already on the operand stack. The procedure subtracts *5* from the *y* value and then pushes the executable name *moveto* onto the stack (by converting the literal name to executable).

Next Page ->>

```
{ 10 add exch 10 sub exch /lineto cvx }
{ 20 sub /curveto cvx }
{ /closepath cvx }
pathforall
```

The other three procedures are similar to the *moveto* procedure; they all modify the values of their *x,y* arguments and then push the executable name of their corresponding operator.

The *pathforall* operator executes the appropriate procedure for each element in the current path. Upon its return, the operand stack will contain a mark and, piled on top of it, all of the *x* and *y* values (some of them modified) and the executable operator names that were used to construct the current path.

```
] cvx
```

The close bracket then collapses the entire contents of the operand stack, down through the mark object, into an array; *cvx* makes that array executable. What we have on the operand stack is now a procedure body that will recreate the current path with our modifications applied.

Note that at this point the original current path is still in place.

```
newpath
exec
```

Now we erase the current path with *newpath* and then immediately reconstruct it, with changes, by executing our newly-made procedure body.

Our modified current path has replaced the original, as we set out to do.

Randomizing a Path

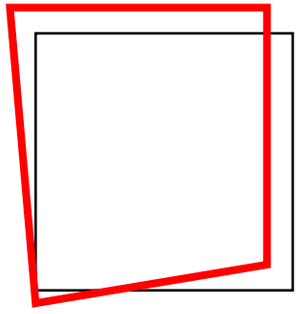
Let's build on the previous example to use *pathforall* to randomize a path, that is, to add a random offset to each of the points in the current path.

We shall use the *rand* operator to generate our offsets. This operator returns a positive integer on the range 0…2^{31}–1; we shall convert each random integer to a floating point value on the range –10…10 and add the result to each *x* and *y* in the current path.

Next Page ->

Making a Randomized Path

Here's the PostScript code:



```
/RandomScale 10 def

/CalcOffset        % --- => n'
{    rand 1001 mod 500 sub 500 div RandomScale mul } bind def

/RandomizePoint      % x y => x' y'
{     CalcOffset add exch CalcOffset add exch } bind def

/MakePathProc       % ---   =>   { proc }
{    [
       { RandomizePoint /moveto cvx }
       { RandomizePoint /lineto cvx }
       { 3 { RandomizePoint 6 2 roll } repeat /curveto cvx }
       { /closepath cvx }
      pathforall
     ]
     cvx bind
} bind def

100 600 moveto    % Create square path
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath

MakePathProc      % Create new "pathproc" (leaving the proc on the stack
```

Next Page ->

Acumen Journal: PostScript Tech 9

```
stroke              % Stroke the original path

exec                % Execute the pathproc, creating the new, randomized path
1 0 0 setrgbcolor % Stroke the new path with a red line
3 setlinewidth
stroke

showpage
```

Let's look at this in a bit of detail.

Step by step

```
/RandomScale 10 def
```

We start by defining a constant named *RandomScale,* which will be the maximum value for our random offsets.

```
/CalcOffset         % --- => n'
{    rand 1001 mod 500 sub 500 div RandomScale mul } bind def
```

Our *CalcOffset* procedure returns a random number between –RandomScale and +RandomScale. There are lots of ways to convert a positive integer into a floating point value on a given range; I'm using a way that is conceptually simple, though some of you may prefer other algorithms that produce better, more-random results; you are invited to write your own procedure.

This procedure does the following:

1. Use the *rand* operator to get a random integer.
2. Mod this by 1000, yielding a random integer on the range 0…999.
3. Subtract 500 from this, yielding an integer on the range –500…499.
4. Divide by 500, resulting in a floating point value between –1 and (almost) 1.

Next Page ->

Multiply this by our RandomScale, giving us a floating point value on the proper range.

```
/RandomizePoint      % x y => x' y'
{     CalcOffset add exch CalcOffset add exch } bind def
```

The *RandomizePoint* procedure takes an *x,y* pair from the stack and applies a random offset to each of the two numbers, leaving the modified values on the stack.

```
/MakePathProc       % ---   =>   { proc }
{    [
       { RandomizePoint /moveto cvx }
       { RandomizePoint /lineto cvx }
       { 3 { RandomizePoint 6 2 roll } repeat /curveto cvx }
       { /closepath cvx }
       pathforall
     ]
     cvx bind
} bind def
```

The *MakePathProc* procedure uses the technique from the previous example to create a make-a-path procedure, leaving this procedure on the stack as its return value. Note that each of the four path-element procedures apply the *RandomizePoint* procedure to its *x,y* arguments.

```
100 600 moveto    % Create square path
0 100 rlineto
100 0 rlineto
0 -100 rlineto
closepath
```

Now we construct our square path.

Next Page ->

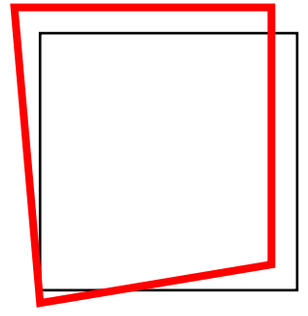
Acumen Journal: PostScript Tech 11

```
MakePathProc
stroke
```

And then call *MakePathProc,* which leaves a path-construction procedure on the stack.

Since *MakePathProc* has no effect on the current path, we can immediately call *stroke,* which draws the original path onto the page.

```
exec                % Execute the pathproc, creating the new, randomized path
1 0 0 setrgbcolor % Stroke the new path with a red line
3 setlinewidth
stroke
```



Finally, we execute the path-construction procedure, which creates the randomized version of our original path, and then stroke the new path with a red line.

```
showpage
```

And then we eject the page.

Useful Operator

The *pathforall* operator can be a useful little guy, if not very often. When you need it, there's no substitute.

For example, one of the most common text-along-a-path algorithms uses *pathforall.*

A pity we don't have time to talk about it in this issue.

Later, perhaps.

Return to Main Menu

Acrobat 8 Typewriter Tool

The Typewriter tool in Acrobat 8 is a bit of a sleeper. The first dozen times I worked with Acrobat 8, I could not imagine why anyone would use this tool; it didn't seem to offer anything of any particular value and had all the characteristics of just being a pet project of someone significant on the Acrobat 8 design team.

Live and learn, I suppose. Having used Acrobat 8 for nearly a year, now (counting the time I was working on the Acrobat 8 Visual Quickstart book, of which you should buy several, by the way), I have found the Typewriter tool to be extremely useful on frequent occasion. It's not something I use every day, but when I use it, I'm very glad it's there.

Let's look at what it does.

The Scenario

The circumstance under which I use the Typewriter tool is actually pretty common: when I receive and need to fill out a paper form, either scanned or as actual paper (quaint, but it happens).

Consider the form at right, emailed to me as a PDF file. Although this file is in PDF format, it is not really a form, in the PDF sense of the word. There are no interactive elements on the page; the form fields are just lines drawn on the page, not text fields or checkboxes into which I can type information.

If this were a form I would use repeatedly—if I were converting my own paper form into a PDF form for sending to many customers, for example—I would lay PDF form fields on top of the page's lines and rectangles, converting it into a true PDF form.

Infect Your Neighbors
A PLAGUE ON BOTH THEIR HOUSES

Application for Infection

Neighbor 1

Name _____
Street _____
City, State, ZIP _____

Reason for Infection
- ☐ Trash in yard
- ☐ Funny looking
- ☐ Beats up my kid

Desired outcome
- ☐ Rash
- ☐ Funny looking

Signature _____
Date _____

Next Page ->

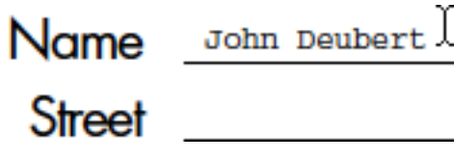
However, this was a form was created and sent to me by another company (it could have been a tax ID form, for example); I just want to fill the form out, send it back, and get on with my life.

This is what the Typewriter tool is for.

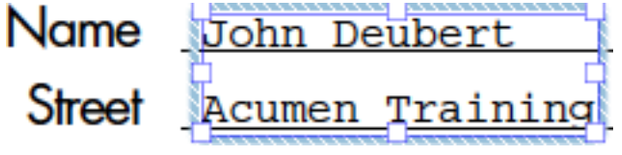
The Typewriter Tool

The Typewriter tool allows you to conveniently fill out paper form fields by placing text annotations directly on the PDF page. You simply click on the tool and the cursor turns into a standard I-Beam. Click anywhere on the page and type the text you want.

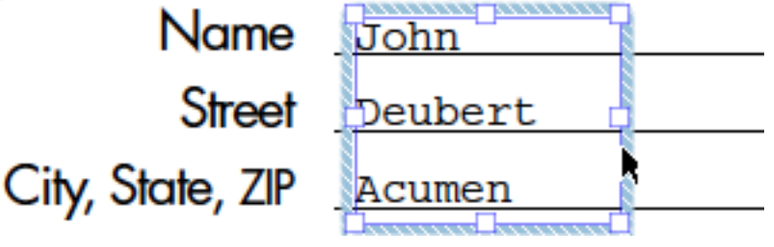
I used to do this with the Free Text annotation tool. In fact, the Typewriter tool leaves on the page what appears in every way to *be* a Free Text annotation. However, with the Free Text tool, I was forever discovering that I'd left the tool's defaults so it drew red text against a blue background, or something equally inappropriate to an official form. This entailed an easy (but nonetheless annoying) trip to the annotations' Properties dialog box.



The Typewriter tool provides convenience, pure and simple; click and type and that's it. You can type multiple lines of text, as at right, and double-clicking on the typed text gives you a border that you can drag around in order to fine-tune the text's position on the page.



Note the border also has standard handles at the sides and corners that allows you to change the size of the text box. Resizing the text box does not change the size of the text, as you might expect; it simply changes the size of the box to which the text is wrapped. As you change the size of the box, Acrobat wraps the text as necessary to keep it within the box; text that wraps beneath the bottom of the box disappears from view.



Next Page ->

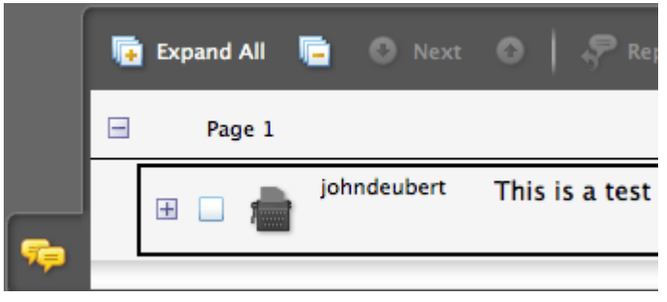
Changing Text Size and Leading

If you double-click on the typewritten text, Acrobat gives you back the blinking I-beam cursor and lets you select and edit the text.

At this point, you can also use the Text Size and Text Leading tools to change the size of the text and the distance between successive lines of text within the text block. These changes apply to the entire block of text; there is no way to change the properties of only a word or two within the block. The intent is to make your typewritten text match the spacing and size of the form fields over which you are typing.

There is also no way to change the font used by the Typewriter tool; you are pretty much restricted to Courier. This seems like a more serious limitation than it really is; the purpose of the tool is to fill out printed forms and, for this, Courier works adequately well. Still, I do often wish that I could change to a proportional font.

Compatibility



As I said earlier, the Typewriter tool adds a variation of a Free Text annotation to the PDF page. Like all annotations, the Typewriter text appears among the comments in the *Comments* navigation pane along the left edge of your document windows (at right, top).

Interestingly, earlier versions of Acrobat can open the file and display the typewritten text, as well. Acrobat versions 5 and later see the text as a standard Free Text comment (right, bottom). I haven't tried to open a typewritten file with Acrobat 4, but I'll bet that would work, too.

Next Page ->>

So, What's Bad? Not much, actually, within the context of the tool's purpose. The only two things I have found irksome when using the tool are:

- I wish you could change the font; not a big deal, but Courier is such an ugly font.
- Every time you want to start a new piece of text (when moving from "Name" to "Address," for example, you need to click again on the Typewriter tool button. Again, not a real trial, but surprisingly annoying when you are filling out a large form with a lot of small fields. (It'd be nice if clicking outside of the current block of text would start a new one.)

Otherwise, this is a surprisingly useful tool and I recommend you experiment with it.

Return to Main Menu

Schedule of Classes, July–October 2007

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

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

PDF 1: File Content and Structure		Aug 13–16	Oct 1–4
PDF 2: Advanced File Content			Oct 8–11
PostScript Foundations	Jul 30–Aug 3		Sept 17–21
Variable Data PostScript			
Advanced PostScript			
Troubleshooting PostScript		Aug 27–29	

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

Register on Acumen Training website

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: john@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 Main Menu

What's New at Acumen Training?

Support Engineers' PDF

Support Engineers' PDF is a two-day, hands-on, technical introduction to the PDF file format. It discusses the basics of the structure and contents of a PDF file, emphasizing those parts of the PDF specification most important to printed documents. The course is a good, quick introduction to PDF structure for people who need to examine and diagnose troublesome PDF files.

Note that this is a class in the PDF file structure, not the use of Adobe Acrobat. The course does examine some commercial tools that are useful in the diagnosis of PDF problems.

Course Outline

Day 1

- PDF Data Types
- PDF Objects
- PDF File format
- The Page Tree
- Content Streams
- Simple Drawing
- Introduction to Color
- Drawing Text
- Coordinate Transforms
- Compression & Transmission Filters

Day 2

- Color and Color Spaces
- Image XObjects
- Form XObjects
- Transparency
- PDF Font Structure
- Examination of Common PDF Files
- PDF/X & PDF/A
- PDF Troubleshooting Tools

Availability

Support Engineers' PostScript will be available July 2007. Watch the Acumen Training website for pricing and schedule of classes.

Return to Main Menu

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 seem to have been inexpertly translated from Japanese?

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

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

Please send any comments, questions, or problems to:

john@acumentraining.com

Return to Main Menu