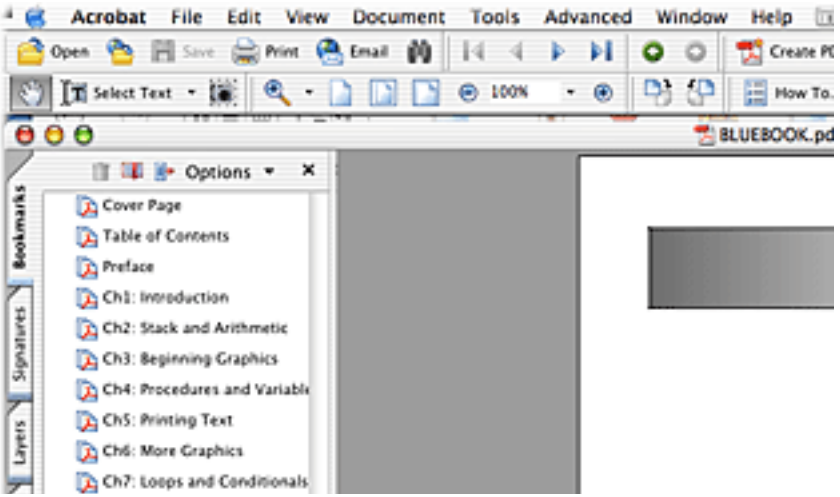
Table of Contents

Acumen Training

Acrobat 6 Form Tools



Acrobat 6 has been released and, although in many ways it is the same Acrobat software we have come to know and/or love, it sure *looks* different. (Among other things, there is clearly a feeling that you cannot have too many toolbars!)

Acrobat 6 also has added a lot of new capabilities, some of which are going to be quite useful. This month, however, I'd like to discuss something that has changed almost completely in its user interface, though only a little in its capabilities: the mechanism by which you add form fields to an Acrobat file.

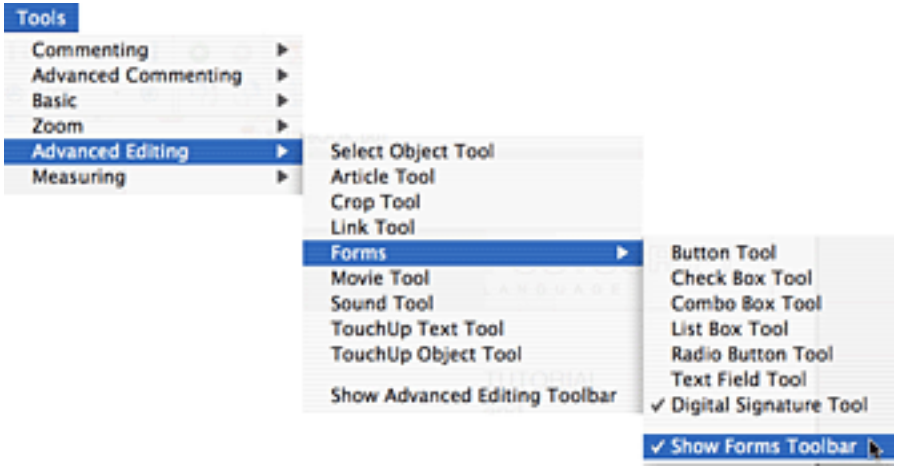
The capabilities of Acrobat's form mechanism are substantially the same in Acrobat 6 as in Acrobat 5, but the means by which you make form fields, specify their properties, and otherwise turn your Acrobat file into an interactive form has changed its look completely. Specifically, it's better.

Let's see what it looks like now.

Next Page ->

The Forms Toolbar

Acrobat 6 replaces the earlier Form Tool with a new *Forms Toolbar* containing a button for each of the seven types of form fields supported by Acrobat. This tool-bar may not be initially visible; to get to it, you need to select: *Tools>Advanced Editing> Forms>Show Forms Toolbar*

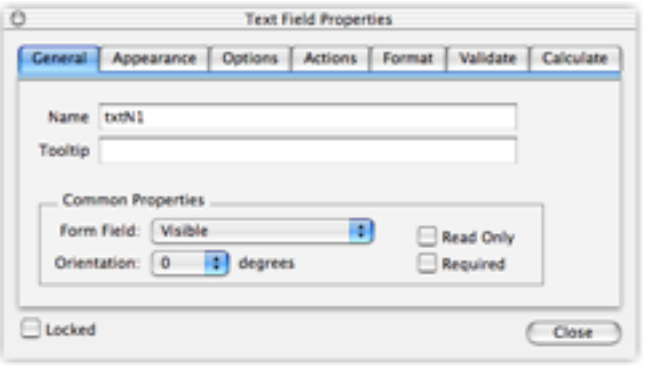


Making a Form Field

To place a new form field on an Acrobat page in Acrobat 6, you do the following:

1. Make the Forms toolbar visible, if necessary.
2. Click on the appropriate button in the tool bar.
3. Drag out a rectangle on the Acrobat page.

You will now be faced with the new incarnation of the *Field Properties* dialog box. This is much improved over the Acrobat 5 version. (Especially in Mac OS X, in which the *Field Properties* dialog box was particularly ugly.)



Next Page ->

Text Field Properties

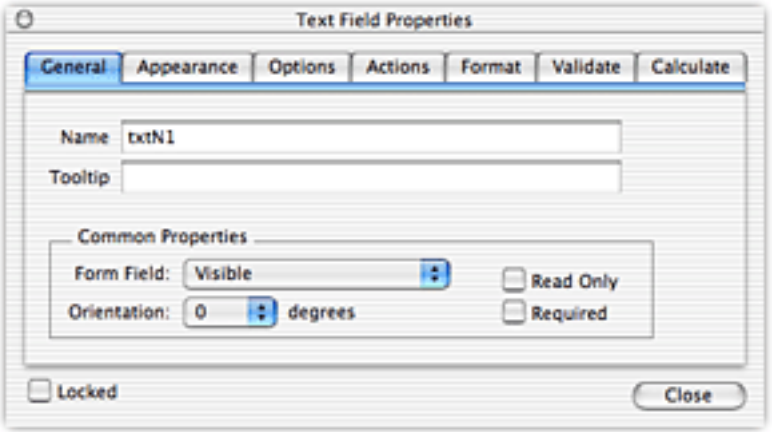
Let's look at the contents of the new *Field Properties* dialog box. We shall start by examining the each of the properties panels for Text fields (since they have the richest set of properties). This is a brief overview; we shall discuss in detail only those panels that have a feature new to Acrobat 6.

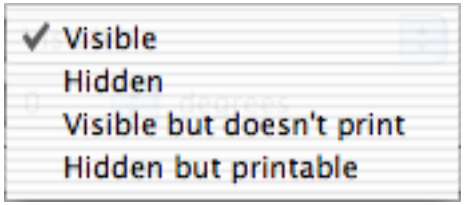
General Panel

All of the *General* properties panel's controls were available in Acrobat 5. You specify the name of your form field and a variety of other properties. The only things new here are UI features:

- The old "Short Description" field is now more-usefully labeled "Tooltip."
- The visibility pop-up menu has been reworded again. I like the new wording better than Acrobat 5's.

Next Page ->

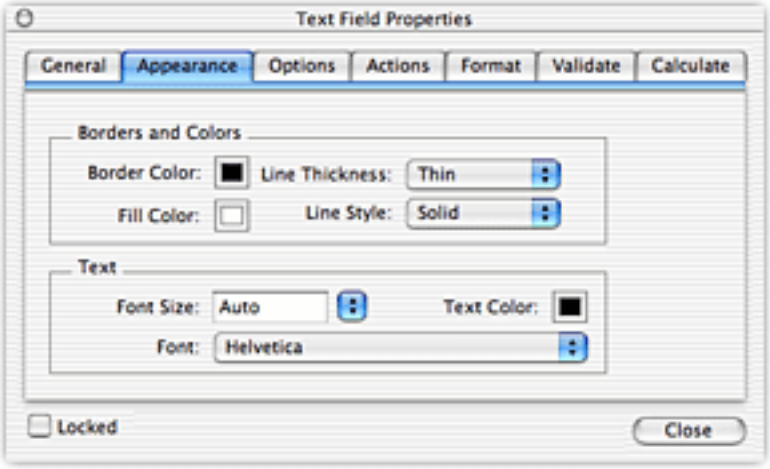




Appearance Panel

The *Appearance* properties haven't changed, aside from appearance.

It does look nice, though.

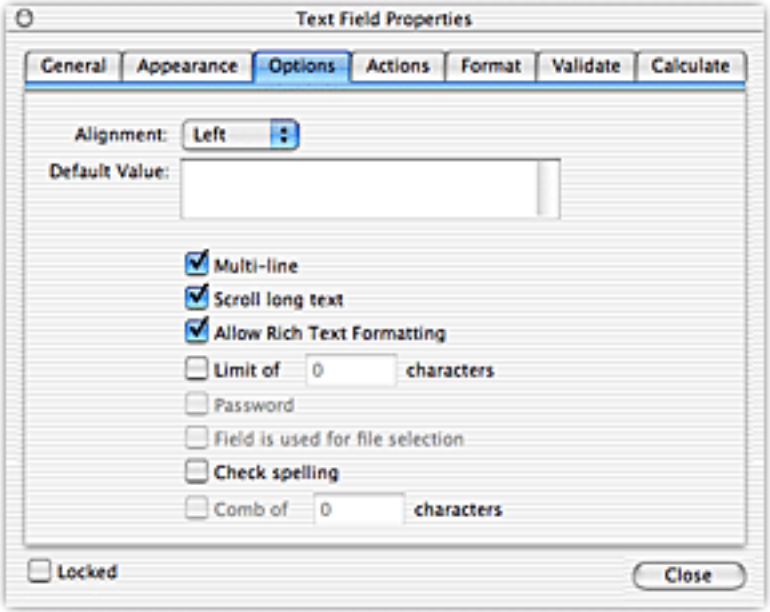


***Options* Panel**

The new *Options* panel for Text fields does have a couple of new options:

- *Allow Rich Text Formatting* allows the Text field to contain styled text.
- *Comb of xx characters* spreads the field's contents (up to a specified number of characters) across the width of the text field. Each character is separated by vertical border, as in the illustration at left. (This is for forms that need to have each character in a separate box.)

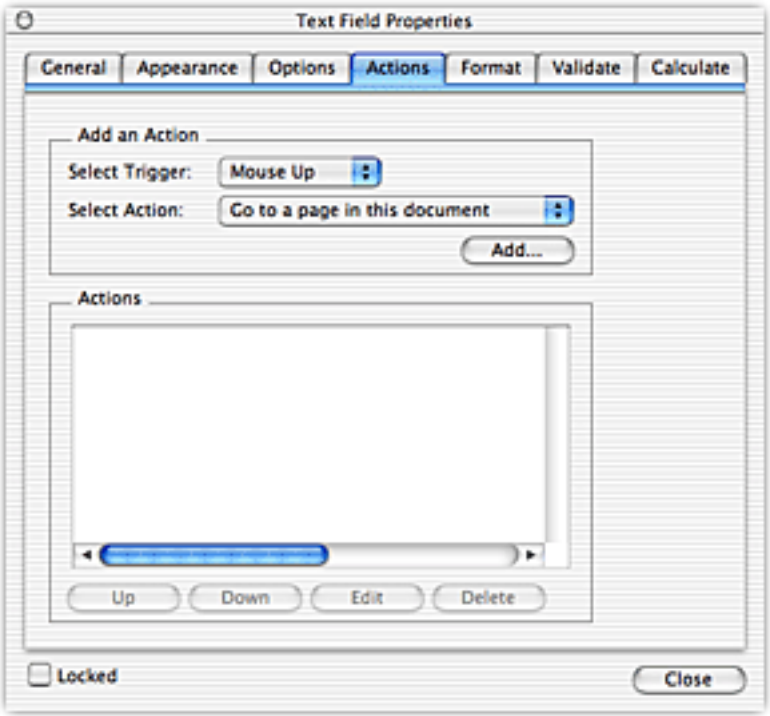
A	B	C	D	E

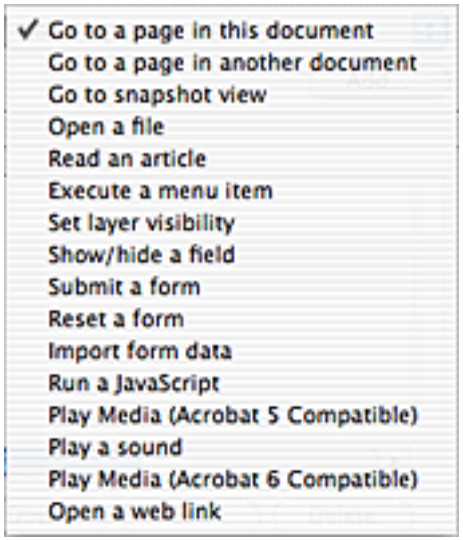


Next Page ->

***Actions* Panel**

The *Actions* panel, itself, hasn't changed except in appearance; however, the *Select Action* pop-up menu gives us some significantly useful new actions that a form field can carry out.





What's new here are:

- *Go to a page in this/another document*
  These two actions send the user to another location within this or another PDF file. In Acrobat 5, links and bookmarks had a "Go to" action, but form fields did not; form fields needed to use a JavaScript for this activity.

Next Page ->

- *Go to a snapshot view*
  If you use the Acrobat *Snapshot* tool to grab a picture of some part of a page and then later make a form field, this action will convert the most recent snapshot to a destination for a *Go To* action. The net effect is to present the user with a view that is zoomed in on the earlier snapshot's target area.

- *Set layer visibility*
  Applications that support the notion of "layers" within a document can now have those layers survive into the PDF file created from that document. This action lets you change the visibility of those layers.

  Note that it is the responsibility of the application that creates the PDF file to preserve the layers. I would be astonished if the next versions of Adobe's products (and perhaps the upcoming QuarkXpress 6?) didn't support this.

- *Play Media (Acrobat 6 Compatible)*
  This is nearly the same as the Acrobat 5 *Movie* action. The important differences that Acrobat 6 compatible movies support a much broader range of media types (avi, flash animations, etc.) and the movie can be embedded in the PDF file, rather than residing as an external file.

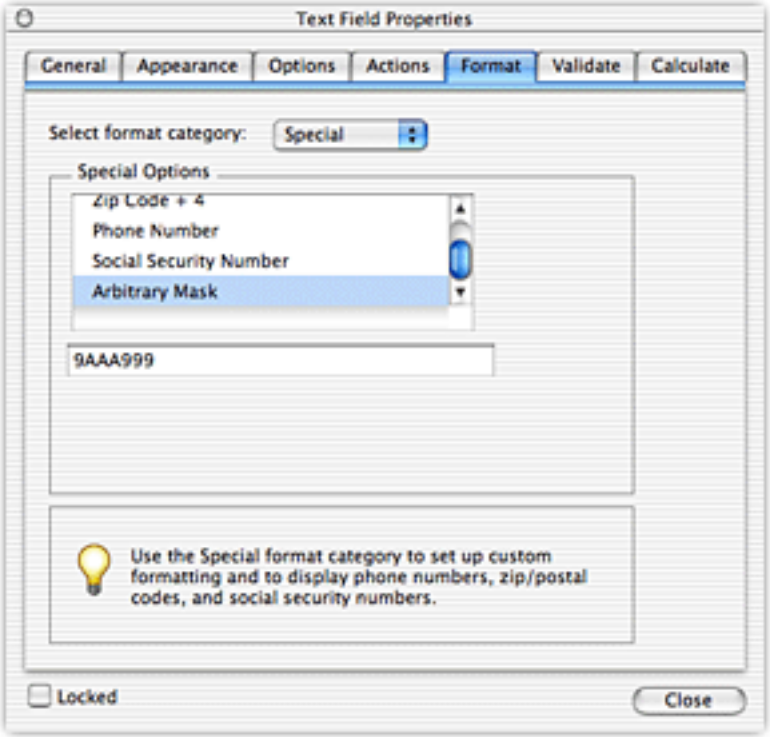
Next Page ->

***Format* Panel**

The *Format* panel adds one new feature to the Text field's predefined formats: *Arbitrary Mask.* This allows you to specify a character sequence pattern using a very simple formatting "language." ("A" matches any alphabetic character, "9" matches any numeric character, etc.)

Thus, the mask "9AAA999" matches California's automobile license plate numbers, such as "4XYZ123."

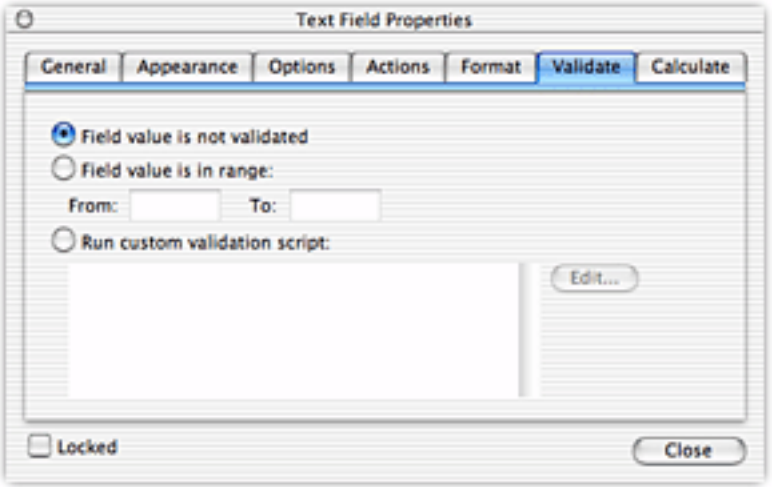
We'll describe this in an upcoming issue of the *Journal,* since this is a common need in many forms that could be done in Acrobat 5 only using much-more-difficult regular expressions.



Next Page ->

Validate Panel

The *Validate* panel is pretty much unchanged from Acrobat 5, except for looks.

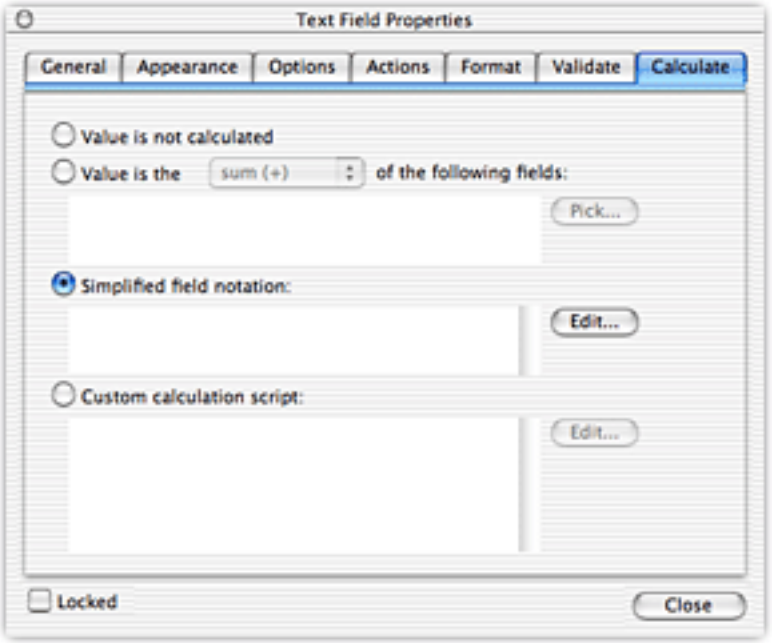


Calculate Panel

A significant addition to the *Calculate* panel is *Simplified field notation.* This allows you to calculate a Text field's contents using a simplified syntax that is much easier than *JavaScript.* The new calculation format has some of the flavor of a spreadsheet macro.

Again, we'll see this in much more detail in a future *Journal* article.

Next Page ->



Other Field Types

The other form field types have some additional properties added to them, as well. Let's review what's new.

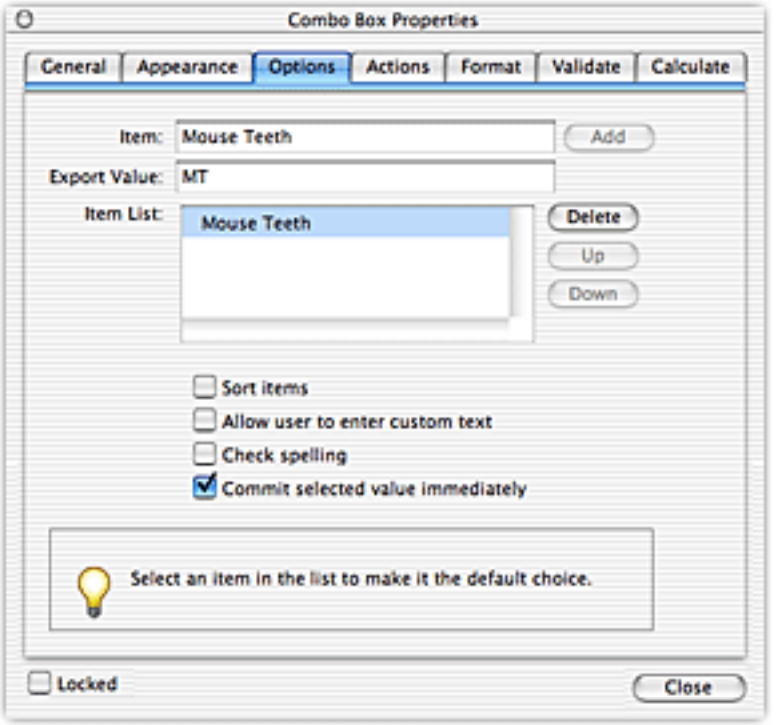
Button Fields

Button fields have no new features that I have noticed so far. I especially appreciate the improved user interface here, however.

Combo & List Boxes

Combo Box and List fields have a new option: *Commit selected value immediately* When the user clicks on an item in the List or selects an item in the Combo Box, that item immediately becomes the field's value. (Normally—and always in Acrobat 5—the user's selection doesn't become the field's value until the user tabs out of the field or selects another field.)

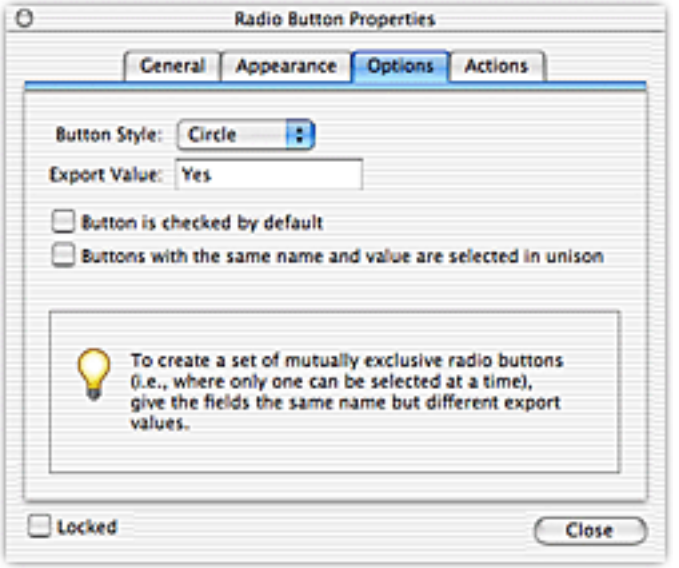
This option will make it easier to implement some interactive form features, such as "live" updating of other fields based on the user's selection in the list.



Next Page ->

Radio Buttons

Radio buttons have a new property, *Buttons with same name and value are selected in unison.* The name describes it pretty well: click on a radio button and *all* radio buttons with that name and export value become selected.



Signature Fields

There is nothing new among the properties for Signature fields that I can see. Looks nicer, of course.

Other New Features

In addition to new form field properties, there are a number of new form-related features in the Acrobat 6 application. The ones I've gotten to know and like the best so far are:

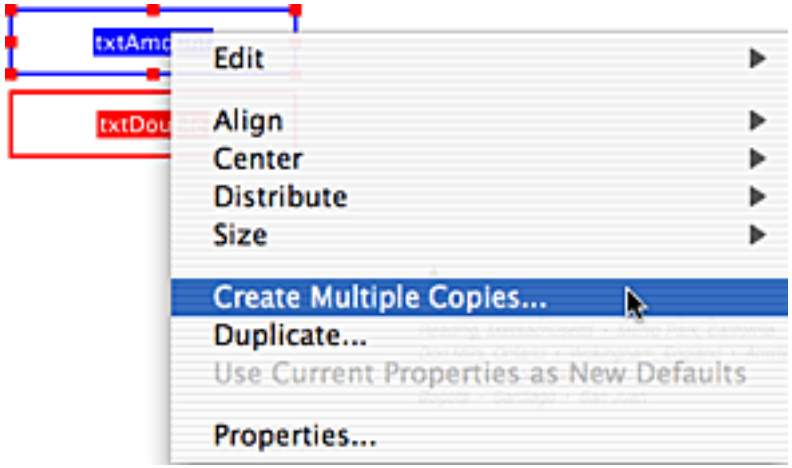
Expanded JavaScript Support

The Acrobat 6 JavaScript interface defines a number of new objects and adds properties and methods to the previously-existing objects. Some of these are quite interesting, but I'll defer this whole topic until next month, since it's quite a lot to chew on.

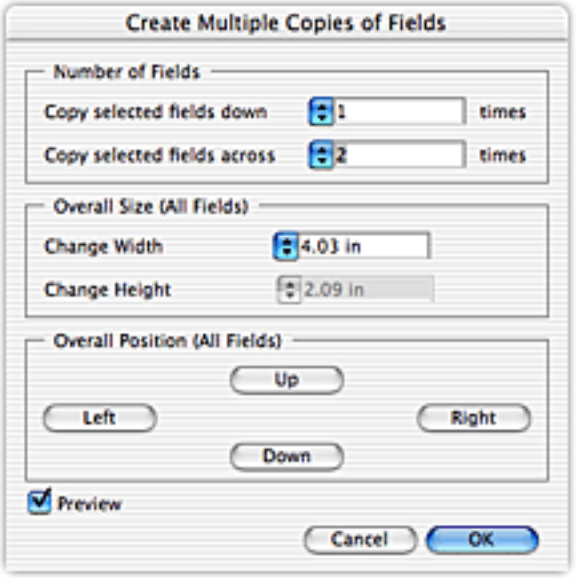
Next Page ->

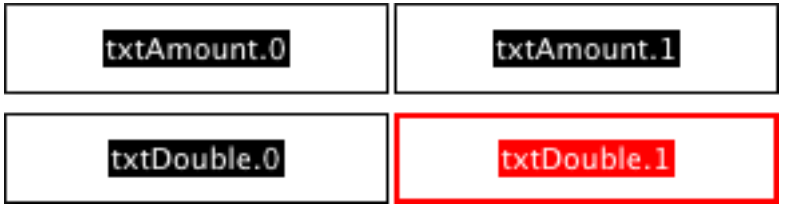
Creating Tables With *Create Multiple Copies*

If you right-click (or control-click, if you have a one-button mouse) on a form field, you can select *Create Multiple Copies* from the resulting contextual menu. This feature replaces the old, hidden create-a-table feature in Acrobat 5.



When you select this feature, Acrobat presents you with the *Create Multiple Copies* dialog box, that tells Acrobat to create an array of form fields based on the fields you selected. Acrobat automatically names all the fields (your originals, as well as the new ones) in accordance with the Adobe Hierarchical Naming Convention. (The fields *txtAmount* and *txtDouble* were renamed as below.)



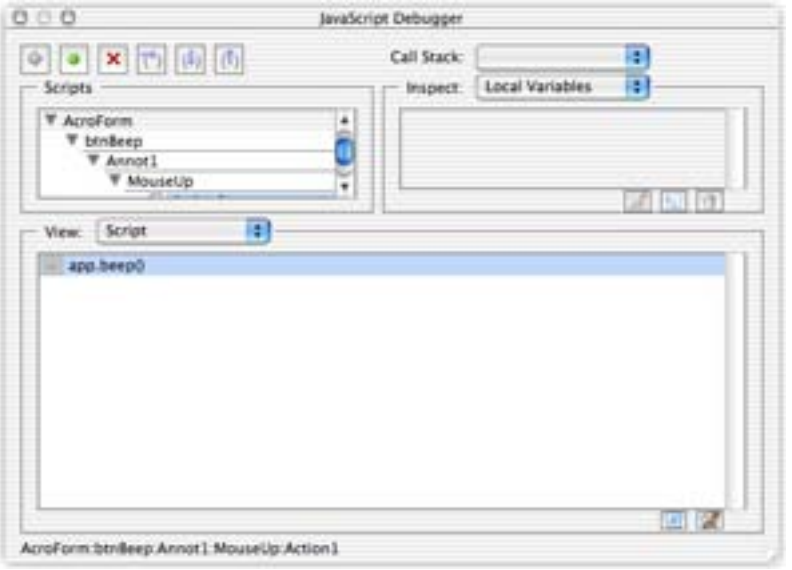


This is an extremely useful feature, *much* improved over the Acrobat 5 equivalent.

Next Page ->

JavaScript Debugger

Acrobat 6 now supplies a JavaScript debugger that is automatically invoked whenever a JavaScript error takes place. Again, I'll talk about this in more detail in a not-too-distant *Journal* article.



But, Wait! There's More

Acrobat 6 is quite a major overhaul of Adobe's Acrobat flagship. I'm very impressed.

It's too early for form designers with a general audience to be designing their documents for Acrobat 6. But people designing forms in a controlled environment should seriously consider switching the users to Acrobat 6. It offers much that is very worthwhile.

We shall be discussing a variety of the new features—the JavaScript debugger, the built-in preflighting, the prepress tools—over the next several issues of the *Journal.*

Return to Menu

Recursive Programming in PostScript

I have always had a fondness for recursive programs. They are both aesthetically and intellectually satisfying, somehow. PostScript actually lends itself quite well to this technique and this month's article will discuss a few of the things you need to know to do this successfully.

The most appealing use for recursive programming is the production of fractal images; I rather like producing natural-seeming shapes (mountains, trees, etc.) using fractals, so this month's sample programs will look at how to produce fractal trees, such as the one at right.

Next Page ->

An Introduction to Recursion

We should start with a simple description of what recursion is, for those who haven't stumbled upon it before.

"Recursion" is the term given to a procedure calling itself. The classic simple case is a procedure that calculates the factorial of an integer. You may remember from high school algebra that *n* factorial—written "n!"—is the multiplicative product of *n* and all the positive integers less than *n.* That is,

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

A recursive algorithm for calculating n! is:

If (n > 1), then $n! = n \times (n-1)!$

If (n = 1), then n! = 1

In PostScript... We can define a PostScript procedure named *Factorial* as follows:

```
/Factorial      % n => n!
{     dup 1 ne                          % Is n ≠ 1?
      { dup 1 sub Factorial mul }       % Yes: return n * ((n-1) Factorial)
      if                                % (Otherwise just leave 1 on the stack)
} bind def

6 Factorial ==
```

The above bit of code will write the value of 6! (*720,* by the way) to the output stream.

Next Page ->

Acumen Journal: PostScript Tech 15

The important significant characteristic of our *Factorial* definition is that it calls itself in the inner procedure body we hand to *if.* This is what makes it recursive.

Limits to Recursion

A PostScript procedure cannot nest calls to itself indefinitely deep; eventually, you will hit one of several limitations.

The Operand Stack

Our *Factorial* procedure leaves its original integer argument on the stack when it makes the recursive call to itself. As we proceed to deeper levels of recursion, *Factorial* will pile more and more numbers on the operand stack. This places a limit on the depth of the recursion, that is, the number of times the recursive procedure may call itself.

In this case, the maximum integer we may hand to *Factorial* is the maximum number of items that may be piled up on the operand stack without causing a *stackoverflow* error. In a Level 1 interpreter, this was typically 500; in Levels 2 and 3, the number is indefinite, since those versions of PostScript can grow the operand stack as needed.

Execution Stack

When we call a PostScript procedure, that procedure is pushed onto the Execution stack; the interpreter then steps through it in course of its normal execution loop. It follows that every time a procedure recursively calls itself, it pushes yet another instance of itself on the Execution stack.

Eventually, you will fill the Execution stack with all these repeated instances of the recursive procedure and you will get an *execstackoverflow* error. The depth at which this happens is hard to predict since, again, in modern PostScript, the Execution stack can grow to accommodate overflow conditions.

Next Page ->

Acumen Journal: PostScript Tech 16

By the way, many PostScript implementations remove a procedure body from the Execution stack *before* executing the last item in that procedure. This means that if a procedure recursively calls itself as the last thing it does, you will get no *execstack-overflow.* Thus, the following recursive procedure will never provoke such an error:

```
/A_Proc { ... Some PS code or other ... A_Proc } bind def
```

Local Variables and the Dictionary Stack

It often happens in recursive programming that you need variables that are local to the current recursion "level." PostScript has no notion of passing arguments by value nor of local variables in the usual sense. To isolate the variables used by each recursive call, you need to have your procedure start by placing its own dictionary on the dictionary stack as a repository for its variables:

```
/myRecursiveProc          % x y => ---
{
     20 dict begin        % Place temp dictionary on dict stack
     /x exch def          % Define variables
     /y exch def
     /x2+y2 x dup mul y dup mul add def
     ...                  % Do the proc stuff
     ...
     x 2 div y 2 div myRecursiveProc    % Here's the recursive call
     end                  % Remove the temp dictionary from the dict stack
} bind def
```

Next Page ->>

The variables *x* and *y* in each call to *myRecursiveProc* are separate from those in all the other calls to the procedure, because they occupy a separate dictionary. The procedure removes this dictionary from the dict stack with an *end* before returning, so *x* and *y* return to their former values when execution resumes in the next level up.

This places another limitation on recursion depth: you can only place so many dictionaries on the dictionary stack before getting a *dictstackoverflow* error. As with the other stacks, the exact depth at which this happens is ill-defined, since the dict stack can also expand, within limits, when needed. (In Level 1, a typical limit would be 18.)

gsave & grestore save & restore

You may also want to do a *gsave* at the beginning of each recursive call and a *grestore* upon returning. Alternatively, if your procedure uses VM in the course of its execution, you may need to do a *save* at the start and *restore* at the end.

Both of these impose a limit on recursion depth. The *gsave* operator pushes a copy of the graphics state onto the Graphics State stack; eventually this will fill up. (In Adobe's implementation, a typical default depth for the Graphics State stack is 32; it can expand, of course.)

Similarly, there can be only so many outstanding *saves.* In Adobe's implementation of PostScript, this has a limit of *16.* In most cases, this will be the most severe limit upon recursion depth, so you should avoid *save* and *restore,* if you can.

(Also, *save* and *restore* are relatively slow operators. Usually this is imperceptible, but in recursive programming, this could be significant.)

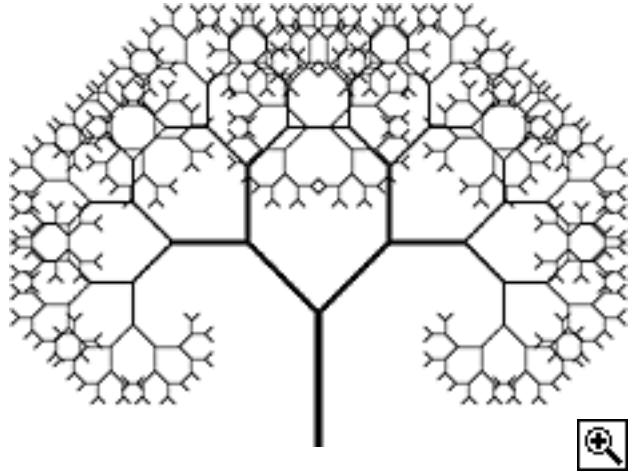
Next Page ->

Acumen Journal: PostScript Tech 18

A Fractal Tree: First Pass

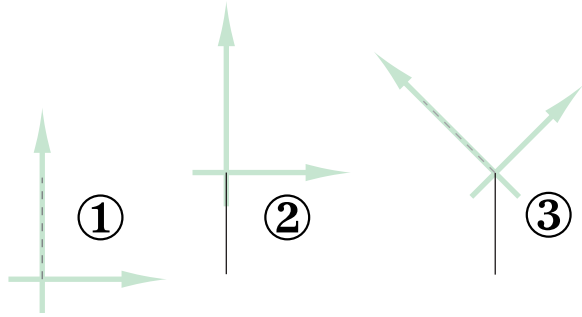
All the files in this article are on the Acumen Training Resources page. Look for the file *FractalTrees.zip.*

Now let's make a fractal tree. We're going to do this in three stages. Our first pass will be a basic tree, producing the output at right.



This tree is built up out of a relatively simple, recursive algorithm, implemented by a procedure named *Branch,* as follows:

1. Draw a branch as a vertical line up the *y* axis, starting at the origin.
2. Move the origin to the far end of the line.
3. Rotate 45° counterclockwise, scale the coordinate system by .75, and execute *Branch* (that's our first recursive call).
4. Undo the earlier coordinate transforms, rotate –45°, scale by .75, and execute *Branch* again (our second recursive call).



The above steps ignore such details as checking to see if we're at our maximum depth. We'll see all that when we look at the PostScript code.

Next Page ->

The PostScript Code

```
/branchLength 100 def     % Dimensions of a branch
/branchWidth 5 def
/maxDepth 10 def          % Exit at this recursive depth
/depth 0 def              % Our current depth
/descendSizeFactor .75 def    % Scale factor upon descent
/angle 45 def             % Rotate for recursion 1
/-angle angle neg def     % Rotate for recursion 2
```

Define Branch Procedure

```
/Branch
{    /depth depth 1 add store      % Increase the current depth
     0 0 moveto                    % Draw a branch
     0 branchLength lineto
     currentpoint translate        % Move origin to far end of branch
     stroke
     depth maxDepth lt             % Are we not at our maximum depth?
     {    gsave                    % Save the graphics state
          descendSizeFactor dup scale   % Scale
          gsave                    % Save the graphics state again
               angle rotate Branch % Rotate counterclockwise; call Branch
          grestore                 % Undo our rotate
          -angle rotate Branch     % Rotate clockwise; call Branch
          grestore                 % Undo rotation & scale
     } if
     /depth depth 1 sub store      % Decrement the current depth
} bind def
```

Next Page ->

Acumen Journal: PostScript Tech 20

Execute Branch

```
300 200 translate                    % Move the origin to the base of the tree
branchWidth setlinewidth             % Initialize the line width
Branch                               % Execute Branch
showpage
```

Step-by-Step

```
/branchLength 100 def
/branchWidth 5 def
/maxDepth 10 def
/depth 0 def
/descendSizeFactor .75 def
/angle 45 def
/-angle angle neg def
```

We start by defining a series of variables that we shall use in our *Branch* procedure. The meaning of each of these should be fairly clear from their names. The only slightly puzzling thing here might be the name of the final constant in this block. "–Angle" looks like a "negative" operator applied to the *Angle* constant; it's actually a name. Remember that a minus sign is a perfectly acceptable character in a PostScript name.

```
/Branch
{    /depth depth 1 add store
```

Our *Branch* procedure starts by incrementing the *depth* variable.

Note that I'm using *store,* here, instead of the usual *def.* The *store* operator takes a key-value pair from the Operand stack and places it into the first dictionary it finds on the Dictionary stack that already contains that key. (The *def* operator, remember,

Next Page ->

always places its key-value pair into the topmost dictionary on the Dict stack.) Since our original *depth* was defined into *userdict,* our call to *store* will always place the incremented *depth* into *userdict,* regardless of the current state of the Dictionary stack.

In our particular case, we never change the Dictionary stack, so *def* would have put our new *depth* into *userdict* anyway. However, if you were putting a new dictionary onto the Dictionary stack every time you descended a level of recursion, you might have wanted to ensure that each recursive call was incrementing (and, later, decrementing) the same instance of *depth.*

```
0 0 moveto                          % Draw a branch
0 branchLength lineto
currentpoint translate              % Move origin to far end of branch
branchWidth setlinewidth
stroke
```

Now we draw the current branch as a vertical line at the origin. Note that we move the origin to the far end of the line segment before we stroke the segment. This will become the starting point for the next level of recursion.

```
depth maxDepth lt
```

We check to see if we are at our maximum recursion depth; if not, we shall continue on to the next level.

Next Page ->

```
{    gsave
     descendSizeFactor dup scale
```

Assuming that our call to *lt* returned a *true,* we do a *gsave* and then scale our coordinate system by *descendSizeFactor,* which has a value of *.75*. (Note the *dup;* remember that *scale* wants two numbers as arguments: an *x* and a *y* scale.)

```
gsave
     angle rotate Branch
grestore
```

Now, we make our first recursive call. We save the graphics state (with another *gsave)* and then rotate the coordinate system *angle* degrees about the origin (which is located at the end of the current branch). We then call *Branch.*

This call to *Branch* will run around and do everything we are currently describing, drawing the next series of thinner branches (each of which will, in turn, call *Branch* to draw the next still thinner series, and so forth).

When *Branch* returns, we call *grestore,* undoing the rotate and whatever other collection of changes *Branch* made to the coordinate system.

```
-angle rotate Branch
```

We rotate clockwise and then call *Branch* again, drawing the right-hand branch (and its descendents) of our tree.

Next Page ->

```
      grestore
} if
```

Finally, we use *grestore* to undo the changes we made to the coordinate system in creating the descendant branches. (This makes the *y* axis once again parallel to the trunk we drew in this instance of *Branch.)*

This ends the procedure that *if* executes while we are below our maximum depth.

```
      /depth depth 1 sub store
} bind def
```

Branch ends by decrementing the *depth* variable. When *Branch* returns, execution resumes in the instance of *Branch* one level of recursion higher. Eventually, execution will bubble back to Level 0 and execution will resume with the input stream.

Keep in mind this has all been procedure definition; we haven't yet executed *Branch.*

```
300 200 translate
branchWidth setlinewidth
Branch
```

Finally, we move the origin to a starting place on the page, set our linewidth, and execute *Branch.*

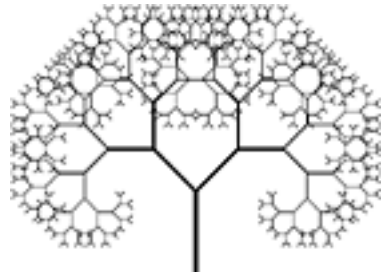
Dissatisfaction

So far, so good. We have a recursive procedure that draws a treelike structure. Still, no one would mistake this for a real tree; it has more the flavor of a diagram than of a drawing.

Next Page ->

Acumen Journal: PostScript Tech 24

One significant problem is that everything in the "tree" is too regular. The angles are exactly the same, the lengths from branch to twig vary by exactly the same proportion. In real life, nothing is so constant.



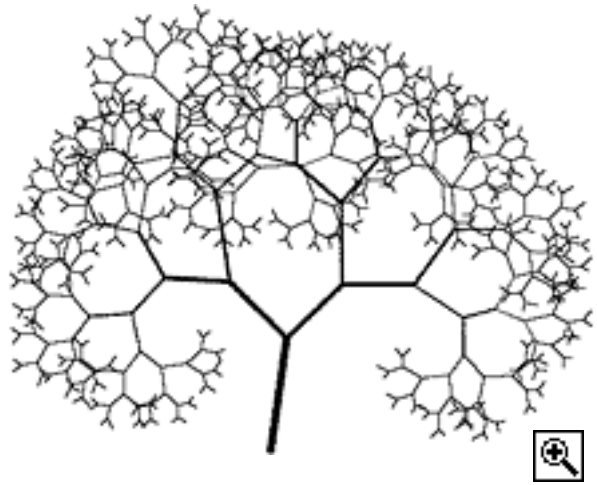
So let's add some variation and see if we get a more lifelike result.

A Fractal Tree: Second Pass

We can make our tree look a little more realistic by adding add some random variations to the angles and lengths of each generation's of branch. To do this, we shall use the PostScript *rand* operator, which returns an unsigned integer that ranges from 0 to 2^{31}-1.

Here's the new code:

Next Page ->



The Code

```
/branchLength 100 def
/branchWidth 5 def
/maxDepth 10 def
/depth 0 def
/descendSizeFactor .75 def
/angle 45 def
/-angle angle neg def
```

Some new constants

```
/dLength 25 def     % Maximum deviation for branch length
/dWidth 1.25 def    % Maximum deviation for branch width
/dAngle 12 def      % Maximum deviation for branch angle

/maxInt 2147483647 def   % Maximum integer returned by rand
```

A new procedure

```
/randomFloat    % --- => n     % Returns random number on the range 0...1
{ rand //maxInt div } bind def    % (See Feb. 2002 Journal about double-slash)

% Returns a value on the range (baseVal-dVal)...(baseVal+dVal)
/ApplyVariation           % baseVal dVal => newVal
{ dup 2 mul randomFloat mul sub add } bind def
```

Next Page ->

Acumen Journal: PostScript Tech 26

```
% This version of Branch is identical to the previous, except we apply
% a variation (using ApplyVariation) to all lengths, widths, and angles
/Branch
{    /depth depth 1 add store
     0 0 moveto
     0 branchLength dLength ApplyVariation lineto
     currentpoint translate
     branchWidth dWidth ApplyVariation setlinewidth
     stroke
     depth maxDepth lt
     {    gsave
          descendSizeFactor dup scale
          gsave
               angle dAngle ApplyVariation rotate Branch
          grestore
          -angle dAngle ApplyVariation rotate Branch
          grestore
     } if
     /depth depth 1 sub store
} bind def
300 200 translate
0 dAngle ApplyVariation rotate     % Apply a variation to the main trunk
Branch

showpage
```

Next Page ->

Step-by-Step What's different in this new PostScript program is:

New constants

```
/dLength 25 def      % Maximum deviation for branch length
/dWidth 1.25 def     % Maximum deviation for branch width
/dAngle 12 def       % Maximum deviation for branch angle
/maxInt 2147483647 def   % Maximum integer returned by rand
```

We define some constants that are used in varying the lengths, widths, and angles of the tree trunks and branches.

New Procedures

```
/randomFloat    % --- => n      % Returns random number on the range 0...1
{ rand //maxInt div } bind def
```

The *randomFloat* procedure returns a random floating point number on the range 0…1. It simply divides the integer returned by *rand* by *maxInt.*

```
/ApplyVariation            % baseVal dVal => newVal
{ dup 2 mul randomFloat mul sub add } bind def
```

Apply variation takes two numbers, a "base value" and a maximum deviation value *(dVal).* It returns a floating point value equal to the base value plus a random deviation not exceeding ±*dVal.*

The math carried out by *ApplyVariation* is:

result = baseVal + (dVal - (2 * dVal * randomFloat))

Next Page ->

New Branch procedure

```
/Branch
{    /depth depth 1 add store
     0 0 moveto
     0 branchLength dLength ApplyVariation lineto
     currentpoint translate
     branchWidth dWidth ApplyVariation setlinewidth
     stroke
     depth maxDepth lt
     {    gsave
               descendSizeFactor dup scale
               gsave
                    angle dAngle ApplyVariation rotate Branch
               grestore
               -angle dAngle ApplyVariation rotate Branch
          grestore
     } if
     /depth depth 1 sub store
} bind def
```

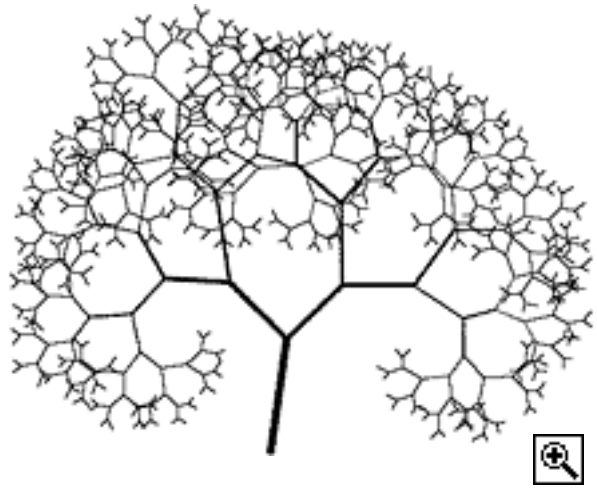
Our new definition of *Branch* differs from the previous definition only in that it applies a random variation (using the *ApplyVariation* procedure) to all of the branch lengths, widths, and angles.

Next Page ->

```
300 200 translate
0 dAngle ApplyVariation rotate
Branch
```

Having finished defining *Branch,* we need to use it to draw a tree: move the origin to some useful starting point; rotate by 0±dAngle° (to add an initial variation to the first trunk's direction); execute *Branch.*

The results are as we see at right.



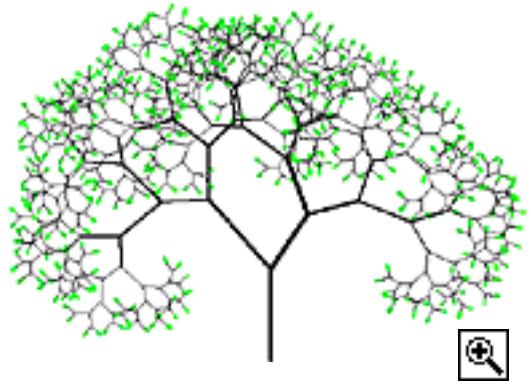
Additional Trees

It is easy to spend endless hours adding this and that to one's fractal trees. The file *Fractal Trees.zip,* on the Acumen Training Resources page, has two files in it in addition to the two trees we looked at in detail in this article. These additions try to increase the realism of our trees.

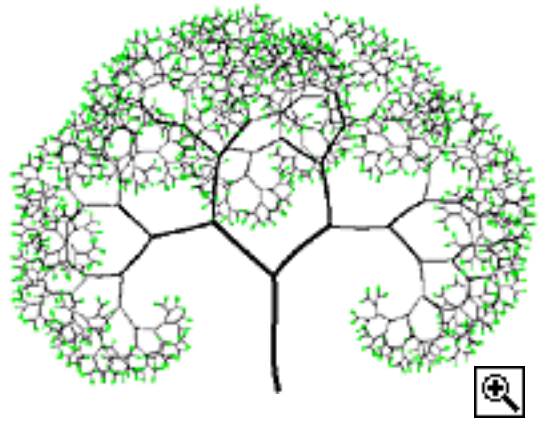
Fractal Tree 2.ps

This file's definition of *Branch* checks to see if it is a the maximum depth and, if so, draws a little leaf whose color is a varying shade of green. I increased the maximum depth by one, so that we have as many branches this time as last, but each final twig has a leaf at its end.

Next Page ->



Fractal Tree 3.ps We get some more-interesting, more natural shapes by replacing the line segments with bezier curves, applying variations to the positions of all the end points and control points.



Fractal Tree 4.ps Now we go completely crazy. I've added a third branch at each level with an angle of 0° (plus or minus the usual variation). I'm also scaling in the *y* direction by a factor of .7±.3, in an attempt to simulate branches pointing toward or away from the viewer (not too successfully, I admit).

This file yields a surprisingly large (7 MB) PDF file, caused by the jump to three branches per level of recursion; we go from $2^{11}-1$ branches to $3^{11}-1$, *i.e.,* from 2,047 to 177,146. (That's why the illustration on this page is a tif, rather than an EPS file.)

Next Page ->

A Call for Readers' Trees

If you find this fun and interesting and feel moved to extend what we're doing here, and if you come up with anything especially interesting, please email it to me. I'll assemble a gallery of readers' trees and post it on the website.

Other Recursive PostScript

As I said at the start of this article, recursive programming can be very addictive. You can find many examples on the web. (Searching Google for "PostScript recurs" returns an even 1,400 hits.)

For an example of a non-graphic recursive program, the Acumen Training *Resources* page contains *QuickSort.ps,* a QuickSort procedure implemented in PostScript. I wrote this nine years ago and I'm not absolutely sure I'll get a chance to spiff it up before this issue of the *Journal* is posted, so the code may not be entirely exemplary.

Return to Menu

Schedule of Classes, Jun – Aug 2003

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 Acrobat class schedule is on the next page.

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		Jul 14–18	
New!	Variable Data PostScript	Jun 16–20		Aug 18–22
	Advanced PostScript			Aug 25–29
	PostScript for Support Engineers		Jul 28–Aug 1	
	Jaws Development		*On-site only*	

PostScript Course Fees PostScript classes cost $2,000 per student.

On-Site Classes These classes may also be taught on your organization's site. Go to www.acumentraining.com/on-site.html for more information.

Registration Info →
Acrobat Classes →

Acrobat Class Schedule

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

Acrobat Essentials

No Acrobat classes scheduled for this quarter. See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Troubleshooting with Enfocus' PitStop

Acrobat Class Fees

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

Registration ->

Return to Main Menu

Contacting John Deubert at Acumen Training

For more information

For class descriptions, on-site arrangements or any other information about Acumen's classes:

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

telephone: 949-248-1241

mail: 25142 Danalaurel, Dana Point, CA 92629

Registering for Classes

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

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

email: registration@acumentraining.com

telephone: 949-248-1241

mail: 25142 Danalaurel, Dana Point, CA 92629

Back issues

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

Return to First Page

What's New at Acumen Training?

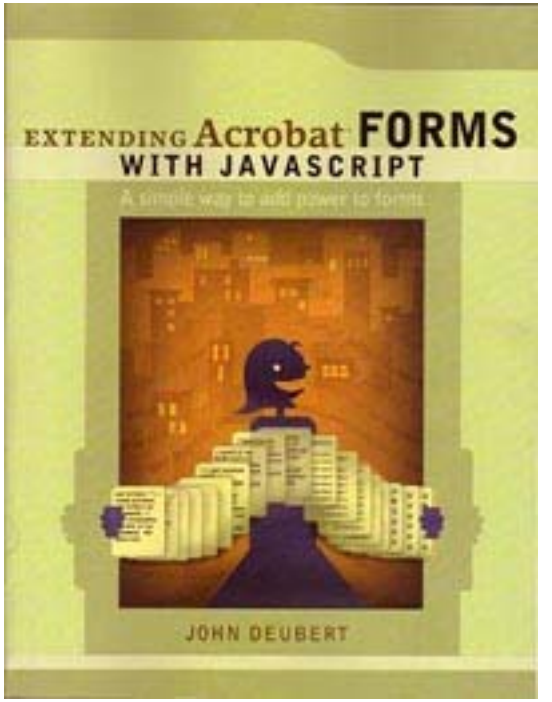
Well, Nothing Very Much

A quiet month, relatively speaking.

You *have* bought several copies of the new JavaScript book, haven't you?

www.acumentraining.com/Book-AcroJS.html

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? After reading it, do you unaccountably want to wash your hands with really hot water?

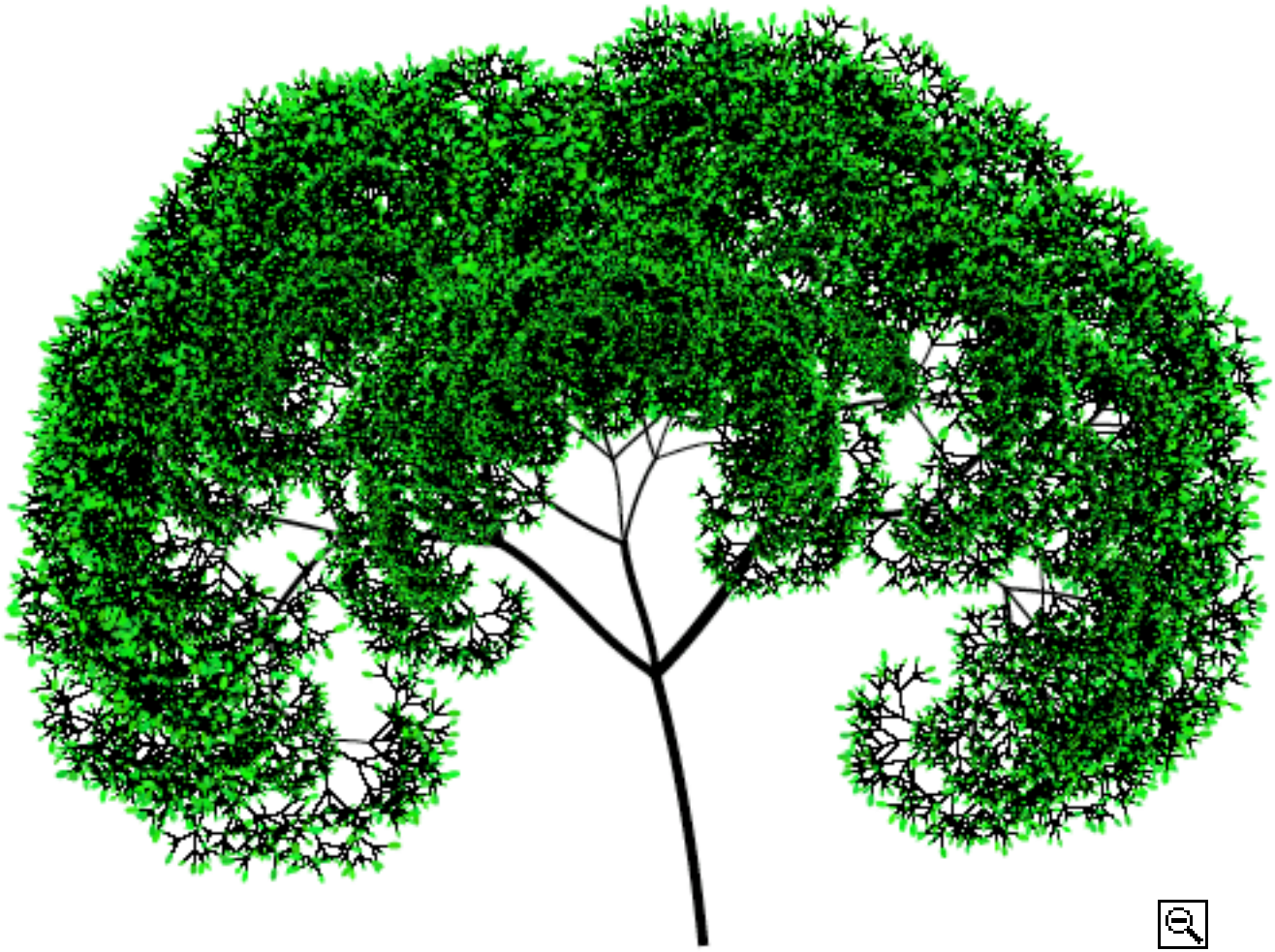
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

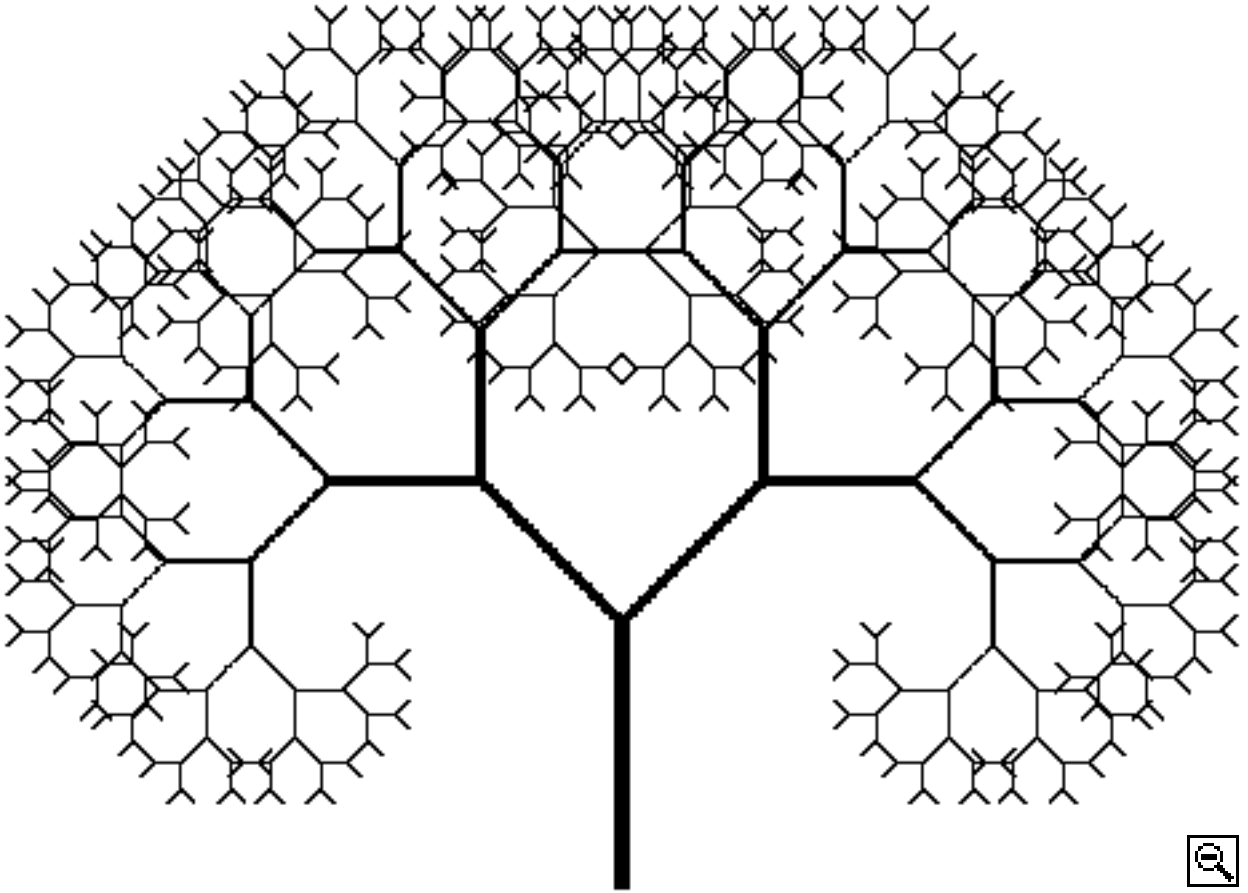
Questions and Answers. Do you have any questions about Acrobat, PDF or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

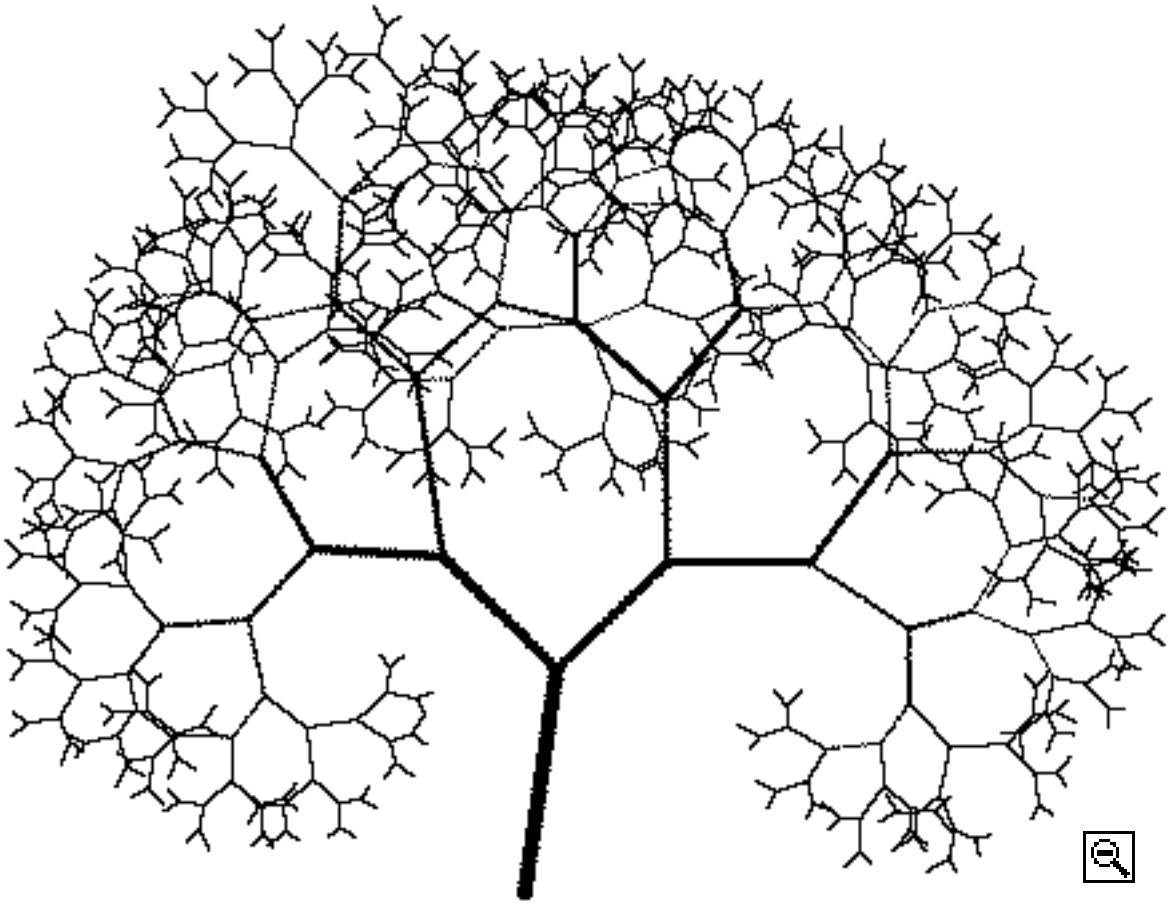
Please send any comments, questions, or problems to:

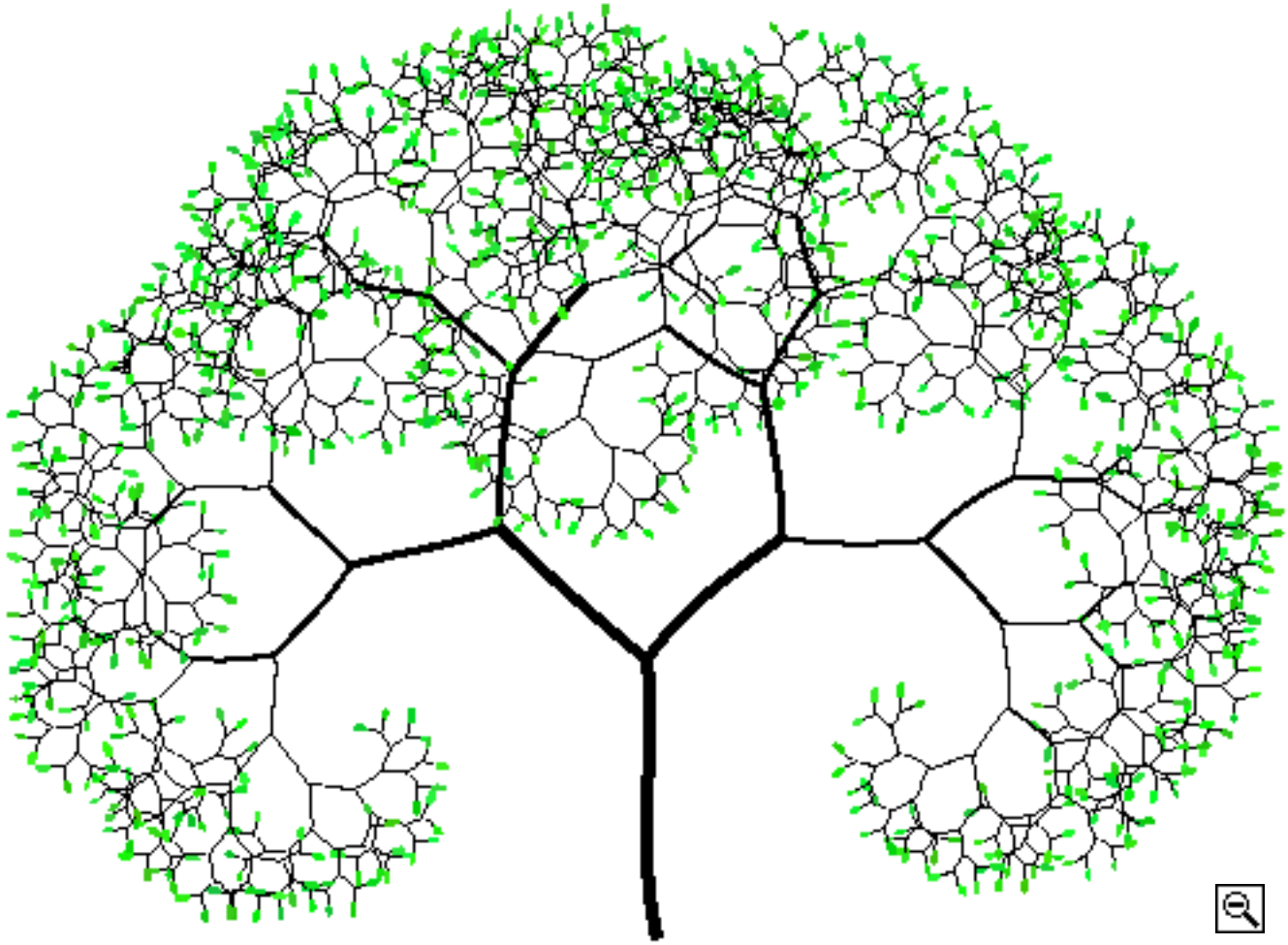
journal@acumentraining.com

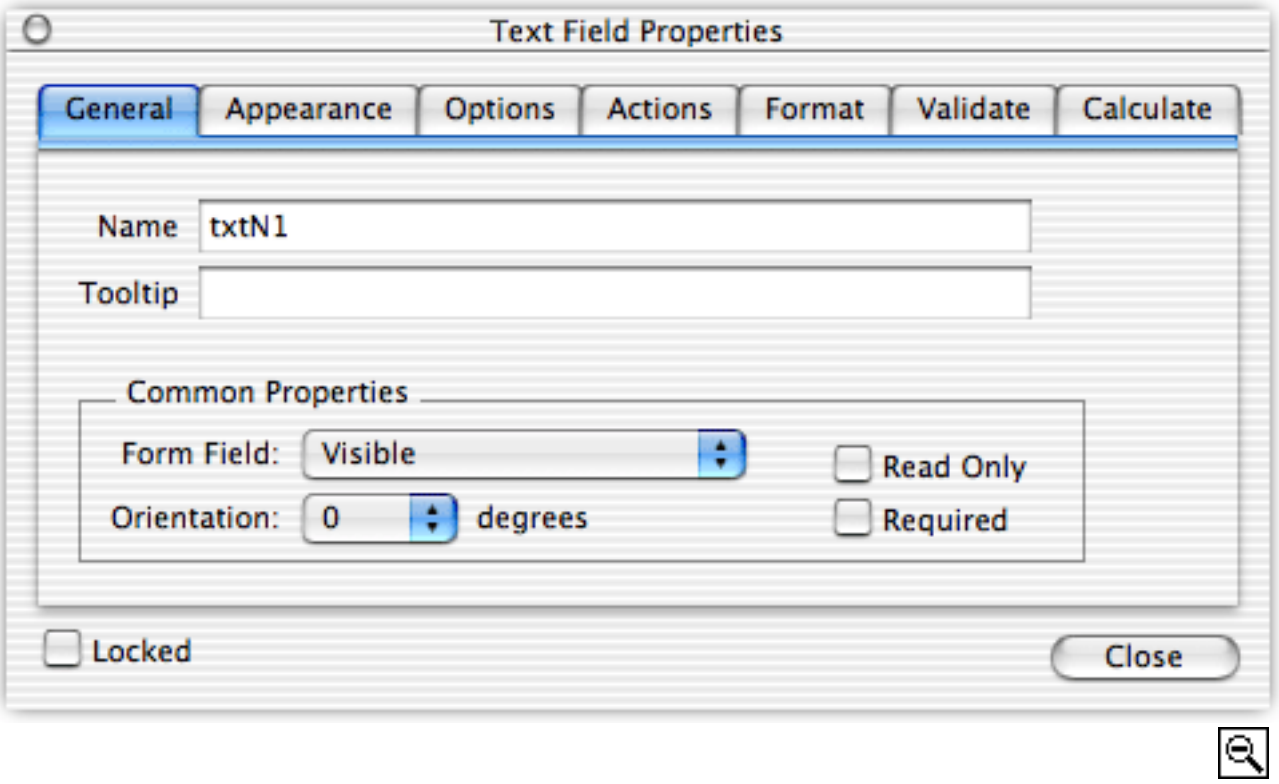
Return to Menu

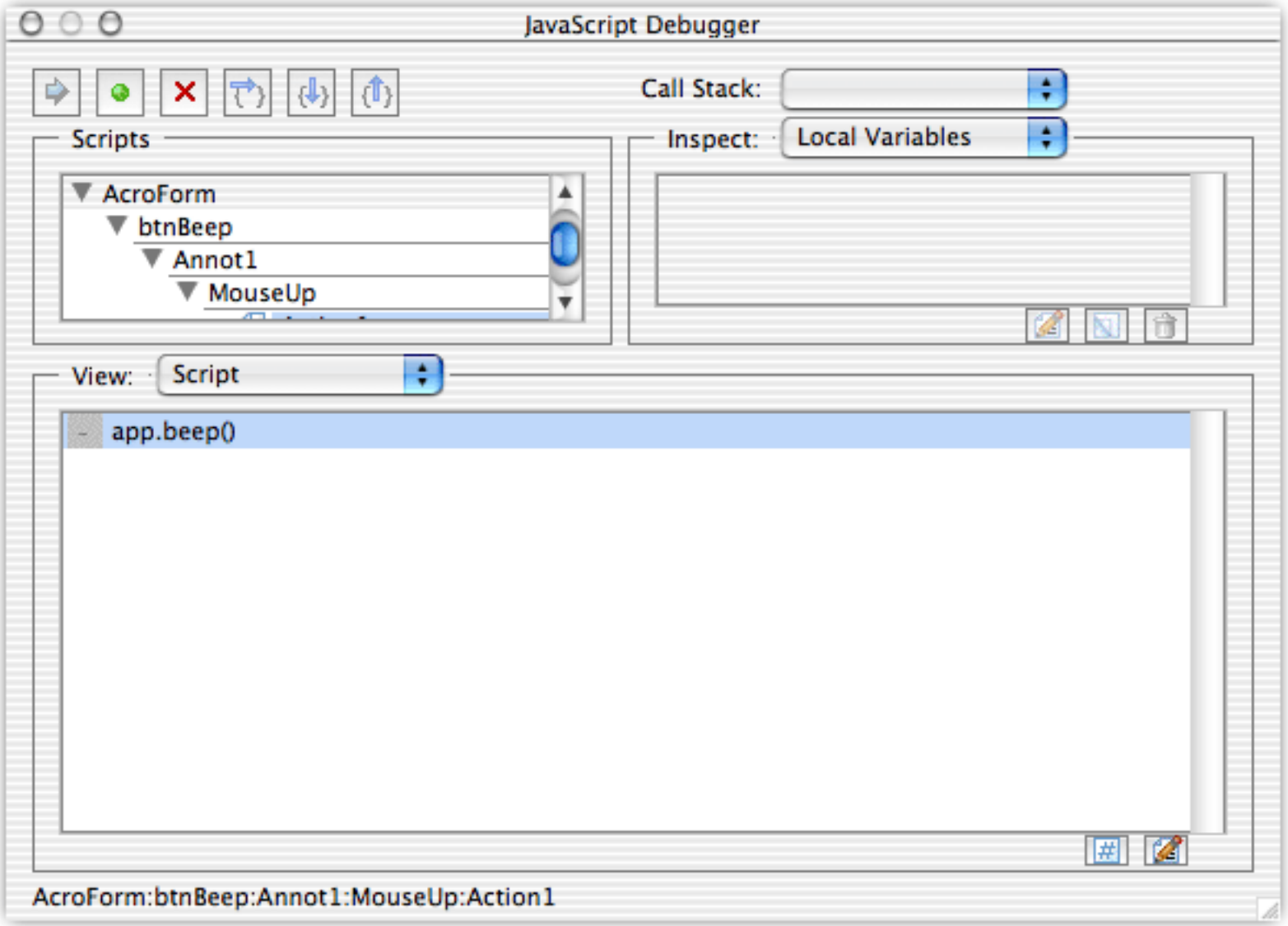









Text Field Properties
General
Appearance
Options
Actions
Format
Validate
Calculate
Name
txtN1
Tooltip
Common Properties
Form Field:
Visible
Read Only
Orientation:
0
degrees
Required
Locked
Close


JavaScript Debugger
Call Stack:
Scripts
Inspect:
Local Variables
AcroForm
btnBeep
Annot1
MouseUp
View:
Script
app.beep()
AcroForm:btnBeep:Annot1:MouseUp:Action1