Adobe Developers Association

ADA news

In This Issue

Volume 3, Number 12

p. 1
The Adobe Illustrator Plug-in API

p. 2
How to Reach Us

p. 10
Developing with Adobe Fetch

p. 11
Acrobat Column

p. 12
Questions and Answers

The Adobe Illustrator Plug-In API

As many developers of Adobe™ application plug-ins are aware, there are several benefits to creating add-on technologies. These include an established installed base for a target market and the luxury of concentrating on the effect of the plug-in rather than larger issues like saving and printing. A market that is ripe for pursuing is that for Adobe Illustrator™ plug-ins. Many developers are not even aware that an Adobe Illustrator application programming interface (API) exists. The Adobe Illustrator Plug-In SDK provides complete information on programming to the API. The API provides a rich function library and offers a high degree of access to the application's graphic object information, allowing the creation of plug-ins for object creation, special effects, file format filters, and niche market requirements.

Adobe Illustrator plug-ins work only in versions 5.0 and later on the Macintosh® platform. They are stand alone code resources that are located when Adobe Illustrator first starts and are then loaded and run as needed. The API uses a one-way message passing scheme to indicate the desired action of the plug-in and an extensive callback function library to manipulate Adobe Illustrator artwork. The working nature of this system should be apparent from the code example and descriptions in this article.

Example Plug-Ins

The Adobe Illustrator 5.x Plug-In API allows developers to create many different types of plug-ins. Keep in mind that every item under the Filter menu in the Adobe Illustrator application was written using the API. These include the ability to:

- Manipulate the colors of objects. The Adjust Colors filter allows you to increase an object's color attributes by percentages.
- Create objects. The Star filter creates a star shape with a specified radius and number of points.
- Perform special effects on objects. The Roughen filter adds points to an existing path and moves them to create a jagged path. The Bloat filter tweaks bézier points to give an object a bloated effect.
- Select, move and duplicate objects. The Adobe Illustrator alignment tools are plug-in modules that move objects.
- Perform operations on text. The Check Spelling and the Find filters, functions similar to word processing tools, were implemented with the API.
- Read and write file formats. The Acrobat™ and PICT file filters in Adobe Illustrator 5.5 were written using the API.

continued on page 2

Adobe

How To Reach Us

DEVELOPERS ASSOCIATION HOTLINE:

U.S. and Canada:
(415) 961-4111
M–F, 8 a.m.–5 p.m., PDT.
If all engineers are unavailable, please leave a detailed message with your developer number, name, and telephone number, and we will get back to you within 24 hours.

Europe:
+31-20-6511-355

FAX:

U.S. and Canada:
(415) 967-9231
Attention:
Adobe Developers Association

Europe:
+31-20-6511-313
Attention:
Adobe Developers Association

EMAIL:

U.S.
devsup-person@mv.us.adobe.com

Europe:
eurosupport@adobe.com

MAIL:

U.S. and Canada:
Adobe Developers Association
Adobe Systems Incorporated
1585 Charleston Road
P.O. Box 7900
Mt. View, CA 94039-7900

Europe:
Adobe Developers Association
Adobe Systems Europe B.V.
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z.O.
The Netherlands

Send all inquiries, letters and address changes to the appropriate address above.

Illustrator Plug-In API

Plug-In API Organization

The Adobe Illustrator 5.x API is composed of a calling definition and over 200 callback functions. The function definitions follow the Pascal calling conventions, though Adobe Illustrator plug-ins are most often written in the C language. The library includes several types of functions, including functions to add filters to the menus, functions to access the artwork database, functions to work with text as an object or as a text stream, and math functions including working with transformation matrices.

The Adobe Illustrator artwork database is made available to the plug-in. The artwork database is a linked list of art layers, each layer having a tree structure made up of grouped objects and artwork objects. Group objects start a new branch of the tree. There are several types of artwork objects:

```
enum {
        kUnknownArt = 0,
        kGroupArt,              // Has its own tree of artwork
        kPathArt,               // Simple paths
        kCompoundPathArt,       // Compound path, a variation of a group
        kTextArt,               // Three types of text
        kTextPathArt,
        kTextRunArt,
        kPlacedArt              // Placed images
};
```

The Adobe Illustrator artwork database is traversed by the plug-in, or specific objects are requested through the API. API functions are also used to access parts of an object, such as a segment of a path. The filter modifies the artwork to produce the desired result.

When the plug-in is called, Adobe Illustrator passes a message indicating the action to be taken. The actions are of two general types: initialization and operation. The initialization phase begins when the Adobe Illustrator software is first launched; the operation phase when the plug-in is selected from the Filter menu. The plug-in is passed a parameter block with a variety of information. This includes a pointer to the callback functions, the version of Adobe Illustrator running, an action message and a handle to any parameters needed by the plug-in. This last item is kept so that repeated use of the function is possible.

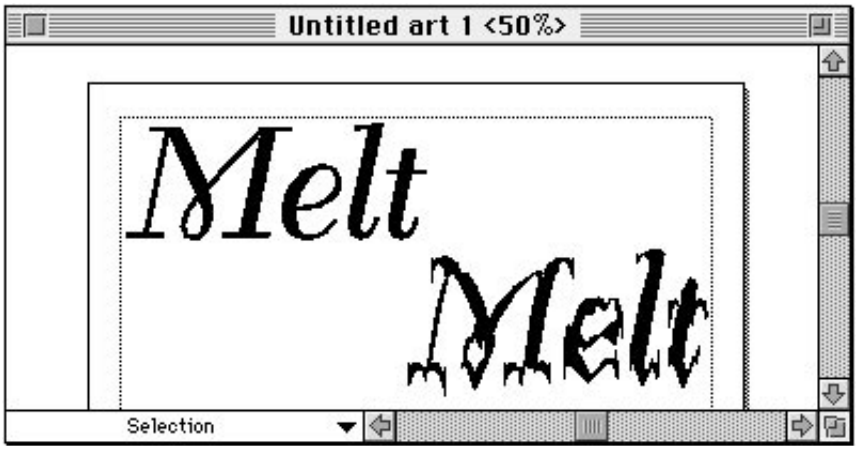
Attention: *Developers!*

Each month, we reserve space in our newsletter for input from our developers. Our Developers Column is a place for members of the ADA to share information with other developers.

If you would like to contribute to our Developers Column, please contact us with your ideas. We look forward to hearing from you.

A Sample Plug-In

At this point it is helpful to look at some actual plug-in code. The Melt plug-in lowers anchor points by a random amount set by the user to produce a melting effect:



The code that follows is complete for the effect with the exception of the header files.

CODE SNIPPET #1

```
//**————————————————————————————————————————————————————————————**
//     melt.c
//     Copyright (c) 1993, 1994 Adobe Systems Incorporated.
//     Lower anchor points of a path to produce a melting effect.
//
//  The first part of the sample code is for initialization.
//**————————————————————————————————————————————————————————————**

//**————————  Headers  ————————**
#include <FixMath.h>
#include "AIPluginInterface.h"    // This file contains the function calls
#include "AIMath.h"

/**——Function & Structure Declarations ——**
typedef struct {
       Fixed meltFactor;
} meltParms;

FXErr go( AIFilterPB *pb );
FXErr get_parms( AIFilterPB *pb );
FXErr melt_path( AIFilterPB *pb, AIArtHandle art );
void  pStrCopy(StringPtr l, StringPtr r );
```

continued on page 4

Illustrator Plug-In API

Code Snippet #1 handles initialization. Notice the structure definition. The message that Adobe Illustrator passes to the plug-in is actually part of a parameter block. The parameter block also includes the callback function list and a handle that you can use to have Adobe Illustrator store any special parameters needed by the plug-in. The structure defined here will store those parameters. In this case there is only one item that affects how the plug-in works, though other plug-ins have many variables. Since this information is generally passed around the plug-in code, it is also a convenient place to store global variables.

CODE SNIPPET #2

```
//**————————————————————————————————————————————————————————————————————**
// Main is basically a switch statement to handle the message passed in
// by Adobe Illustrator.
//**————————————————————————————————————————————————————————————————————**
#define kAboutMsg             16000
#define kAlertMsg             16050
long main( AIFilterPB *pb )
{
       AIFunctions *k = pb->functions;
       FXErr error = kNoErr;

       switch ( pb->selector ) {
             case kSelectorPluginInterfaceVersion:
                    // Specify the minimum version of Adobe Illustrator the
                    // filter works with
                    return kPluginInterfaceVersion;
             case kSelectorPluginStartUp: {
                    // Add our menu item to Adobe Illustrator's Filters menu
                    unsigned char FilterMenu[32], FilterName[32];
                    GetIndString( FilterMenu, 16000, 1 );               // Distort
                    GetIndString( FilterName, 16000, 2 );               // Melt...
                    error = k->AddFilter(FilterMenu, FilterName, 0);
                    break;
                    }
             case kSelectorPluginAbout:
                    // Who we are...
                    Alert(kAboutMsg, nil);
                    return kNoErr;
                    break;
             case kSelectorGetParameters:
                    // Get any information we need to do what we do here.
                    error = get_parms( pb );
                    break;
             case kSelectorGo:
                    // And away we go...
                    error = go( pb );
                    break;
       }
```

Illustrator Plug-In API

```
	// Notify the user of any problems.
	if (error)
		Alert(kAlertMsg, nil);

	return error;
}
```

Code Snippet #2 is the entry point of the plug-in, which is a switch statement to handle the various messages passed by Adobe Illustrator. The first two are called at startup. This is your chance to see if your plug-in is compatible with the hardware and the current version of Adobe Illustrator. For example, a plug-in might only work with Adobe Illustrator version 5.5 or later, or might require a floating point processor. After compatibility is confirmed the plug-in name is inserted in the Filter menu or Save dialog.

The Snippet #2 example code adds a filter called Melt... to the Distort menu when the kSelectorPluginStartUp message is passed. AddFilter() is the first example of an API callback. The kSelectorPlugInAbout message is given if the user uses the About Filter menu for your plug-in. File format filters have an additional message to handle; kSelectorCheckFormat is an opportunity to verify that the file is actually something your filter can handle.

Once the initialization has been done, the plug-in waits for the user to select it from the Filter menu. At this point two messages will be passed in succession: kSelectorGetParameters and kSelectorGo. The first is your opportunity to put up a dialog to obtain needed information from the user; the go message is where the filter actually does its thing.

CODE SNIPPET #3

```
//**————————————————————————————————————————————————————————————————————**
// Most plug-ins need some information from the user before they run.
// kSelectorGetParameters is where you can request that information.
// It makes standard dialog calls, so you can make it as fancy as you want.
//**————————————————————————————————————————————————————————————————————**
#define kMeltDLOG               16100
#define kMeltFactorItem         3
FXErr get_parms( AIFilterPB *pb )
{
	AIFunctions  *k = pb->functions;
	DialogPtr    dialog;
	short        hit;
	short        item_type;
	Handle       item_handle;
	Rect         box;
	Str255       text;
	Fixed        value;
	meltParms    **parms;
```

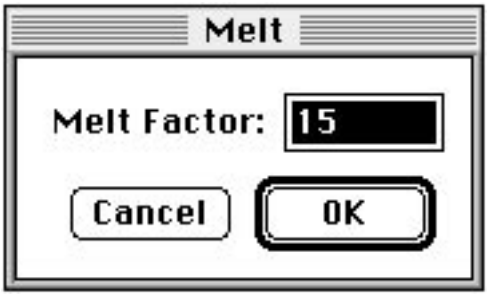
continued on page 6

Illustrator Plug-In API

```
	// Set up our storage for filter parameters if needed.
	// This will happen the first time through, but user values
	// will be stored and used in subsequent calls.
	parms = (meltParms **) pb->parameters;
	if ( !parms ) {
		pb->parameters = NewHandleClear( sizeof(meltParms) );
		if ( !pb->parameters ) return memFullErr;

		parms = (meltParms **) pb->parameters;
		(**parms).meltFactor = Long2Fix(15);
	}

	// Put up and process a dialog to get information from the user
	dialog = GetNewDialog( kMeltDLOG, nil, (WindowPtr) -1 );
	if ( !dialog ) {
		return memFullErr;
	}

	// Most of these calls are standard Macintosh dialog handling
	k->IUFixedToString( (**parms).meltFactor, 1, text );
	GetDItem( dialog, kMeltFactorItem, &item_type, &item_handle, &box );
	SetIText( item_handle, text );
	SelIText( dialog, kMeltFactorItem, 0, 32767 );

	do {
		ModalDialog( k->ModalDialogProc, &hit );
	} while (hit == kMeltFactorItem);

	GetIText( item_handle, text );
	k->IUStringToFixed( text, &value );

	// Clean up and return
	DisposDialog(dialog);

	if (( value != kFixedUnknown) && (hit == ok)) {
		(**parms).meltFactor = _FixedFloor( value );
		return kNoErr;
	}
	else
		return kCanceledErr;
}
```

The function in Code Snippet #3 is executed when the kSelectorGetParameters message occurs. The storage defined and discussed in Code Snippet #1 is allocated here. Notice that the majority of the code here is standard Macintosh dialog handling to setup, show, and process the Melt dialog.

Illustrator Plug-In API



There are a few calls to Adobe Illustrator that take advantage of some utility functions for string conversions. Some of the Adobe Illustrator API calls are there strictly for convenience, for example, conversion and math functions. Notice how the API calls are made indirectly, using a pointer from the parameter block. This mechanism allows Adobe to add new calls to the API, like those added in Adobe Illustrator 5.5.

CODE SNIPPET #4

```
//**————————————————————————————————————————————————————————————————————**
// When the kSelectorGo message is received, the filter does its work.
// Usually, this involve retrieving some artwork and processing it.
//**————————————————————————————————————————————————————————————————————**
FXErr go( AIFilterPB *pb )
{
       AIFunctions       *k = pb->functions;
       AIArtHandle       **matches = nil;
       long              i, count;
       Fixed             selectionHeight = kFixedZero;
       FixedRect         bounds;
       FXErr             error = noErr;
       AIMatchingArtSpec spec[1];

       // Get any selected paths.  If there are none, exit.
       // The MatchingArtSpec array can contain as many conditions as needed.
       // Once filled, this is passed to GetMatchingArt().
       spec[0].type = kPathArt;
       spec[0].whichAttr = kArtSelected;
       spec[0].attr = kArtSelected;

       error = k->GetMatchingArt( spec, 1, &matches, &count );
       if ( error ) return error;

       if ( !count ) {
              error = kBadParameterErr;
              goto error;
              }
```

continued on page 8

```
	// The matching art is returned as an array of art objects.
	// Walk the list of paths and *melt* each one.
	for ( i = 0; i < count; ++i ) {
		AIArtHandle path;

		path = (*matches)[i];
		error = melt_path( pb, path );
		if ( error ) goto error;
		}

	// Clean up and return
	DisposeHandle( (Handle) matches );
	return kNoErr;
error:
	if ( matches )
		DisposeHandle( (Handle) matches );

	return error;
}
```

When the kSelectorGo message is received the function in Code Snippets 4 and 5 are used. In Snippet #4 artwork is extracted from the artwork database using the GetMatchingArt() callback. You can specify the type of art with which your plug-in works and Adobe Illustrator will retrieve it. This is done by setting the type and whichAttr fields of the AIMatchingArtSpec array. In this case, the plug-in works with selected path art. A plug-in could as easily request all placed images or text art. When a simple request for artwork is insufficient, the artwork can be accessed by having your filter walk the artwork tree directly. Once the artwork is retrieved, the plug-in processes each object.

CODE SNIPPET #5

```
//**————————————————————————————————**
// Where the actual melting effect is applied to each object.  The paths
// are broken down to segments and then segments are processed.
//**————————————————————————————————**
FXErr melt_path( AIFilterPB *pb, AIArtHandle path )
{
	AIFunctions	*k = pb->functions;
	meltParms	**parms = (meltParms **) pb->parameters;
	short		count, i;
	Fixed		meltFactor;
	FXErr		error;

	// Retrieve the user specified information
	meltFactor = (**parms).meltFactor;

	// The path is treated as a list of segments
	error = k->GetPathSegmentCount( path, &count );
	if ( error ) goto error;
```

```
	for ( i = 0; i < count; ++i ) {
		AIPathSegment segment;
		Fixed meltAmount;

		error = k->GetPathSegments( path, i, 1, &segment );
		if ( error ) goto error;

		// We process each segment, moving it a certain amount to produce
		// a melting effect.
		if (i%2) k->FixedMul( meltFactor, Long2Fix(2) );
		meltAmount = Long2Fix(_FixedTruncToShort(k->FixedMul( meltFactor,
							k->FixedRnd() )));
		segment.p.v -= meltAmount;

		// The melted segment is then applied to the path.
		error = k->SetPathSegments( path, i, 1, &segment );
		if ( error ) goto error;
	}

	return kNoErr;
error:
	return error;
}
```

The final routine in Code Snippet #5 is where the artwork is actually manipulated. The GetPathSegment( ) function is used to obtain the current segment information. For each segment in the path, the anchor point is moved by the meltAmount. After it is modified using the math library functions, Adobe Illustrator is told to set the artwork with the SetPathSegment( ) function.

And that is a plug-in. When it is run, the Melt filter produces a simple but useful effect. The implementation of the Melt effect is perhaps overly simplified, but effects can be as involved as you want them to be. Certain aspects of the filter, like error handling, should be improved. Error handling and other filter development issues are addressed in more detail in the Adobe Illustrator Plug-Ins SDK.

Conclusion

The Adobe Illustrator Plug-In API provides an opportunity to extend the functionality of Adobe Illustrator 5.x for Macintosh. It allows full access to the Adobe Illustrator artwork database and a set of routines to manipulate it, so only your creativity and know-how need to be added. File formats, special effects, and vertical market needs are just some of the possible types of plug-ins. A tutorial, a full function reference, and more sample code with header files are included in the Adobe Illustrator Plug-In SDK. Contact the Adobe Developers Association if you would like more information about writing plug-ins for Adobe Illustrator. §

Developing With

Adobe Fetch

Q What is Fetch™, and as a developer, what opportunities are there for writing plug-ins or additions to the product?

A Fetch is a multimedia database product for the Macintosh that allows customers to catalog, browse, search, retrieve, and reuse graphic, movie, and sound files. Fetch can be used as a stand-alone product on an individual graphic design workstation or in a work group situation with literally hundreds of users accessing the same database at the same time.

There are a series of add-on products for Fetch 1.2 that were written by add-on developers like you. Fetch 1.2 does not have any application programming interfaces, yet, but does take advantage of certain technologies, such as Macintosh Easy Open and Apple® events, to allow developers to add their own functionality to the product. You can find more information on this topic in the Fetch Compatibility Toolkit.

The Fetch Compatibility Toolkit is comprised of a document and sample files that explain how to make applications "Fetch Aware"—that is how to add support for Apple's 'pnot' resource. The 'pnot' resource is a resource defined by Apple to provide a standard way of attaching keywords, descriptions, thumbnails, and previews of files, so that a program, like Fetch, can extract the information for use in its own context.

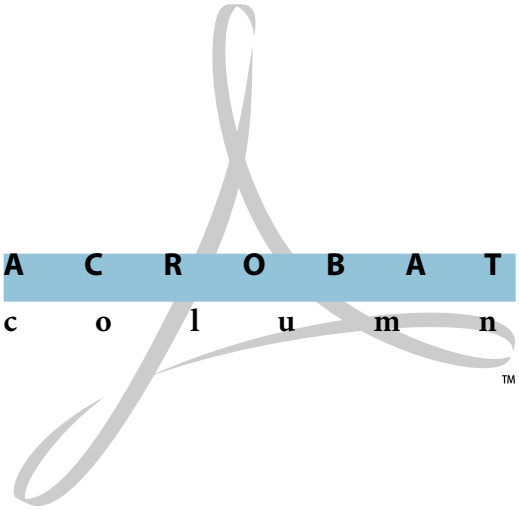
This toolkit also describes the Apple events supported by Fetch and explains how Fetch uses the Macintosh Easy Open (MEO) technology to allow developers to write their own filters for Fetch. Fetch supports a limited number of file formats, and will call a Macintosh Easy Open Translation Extension for an unsupported file. We are relying on developers like you to provide the translation extension for file formats that we do not support. So far, developers have written Macintosh Easy Open Translation Extensions for Scitex® CT, Scitex LW, and SuperPaint™ formats. Other MEO Translation Extensions are on their way!

As a developer, you may also license the Fetch Browser to use as the organizing tool for your clip art or stock photography CD-ROM products. The Fetch Content Publisher's Toolkit is available for developers who license the Fetch Browser.

The Fetch Content Publisher's Toolkit is a CD-ROM that contains a version of Fetch you can use to create a catalog of your images. It also contains the Fetch Browser application that lets your customer browse through the catalog of images you have provided. This toolkit also contains documentation that walks you through the steps for preparing your images, cataloging your images with Fetch, and even burning a CD-ROM of your images.

For more information on the Fetch toolkits, contact the Adobe Developers Association.

The Adobe Fetch gallery window with a preview of a particular file.



This month, we'll describe enhancements made to the two special PostScript™ operators supported by the Adobe™ Acrobat™ Distiller™ application. Version 2.0 of the Distiller application provides expanded versions of the **pdfmark** and **setdistillerparams** operators. **pdfmark** allows your application to generate advanced Acrobat features such as bookmarks and links. **setdistillerparams** allows your application to control various Distiller parameters such as image compression and font embedding. **pdfmark** enhancements allow your application to create:

- articles, by specifying the article's name and a sequence of rectangles that enclose the sections of the article. Articles allow users to read a section of a document that spans multiple columns and pages by simply pressing the Return or down arrow key to go to the next screen.
- links more conveniently. In version 1.0, the **pdfmark** command for creating a link had to execute within the PostScript language commands describing the link's source page, because the source page was determined implicitly from the command's location in the file. In version 2.0, a link **pdfmark** can specify a source page explicitly, allowing it to be placed anywhere in the PostScript file.
- enhanced bookmarks and links. In version 1.0, a link's destination could only be another location within the same file; you could not create a link to another file. Support in version 2.0 has been expanded to allow links to other files, including files that are not PDF files.
- named destinations. Bookmarks and links can now specify a destination by name, rather than by page number and location on the page. This allows you to decouple the updating of documents in a cross-linked collection. For example, suppose you created links whose destination is Chapter 1 in another PDF file (let's call it *dest.pdf*). The page on which Chapter 1 is located may change each time *dest.pdf* is modified. Using a version 1.0-style link, each time you modified *dest.pdf* you would have to modify each link in every file whose destination is Chapter 1 in *dest.pdf*. Using named destinations, you can make the destination of each link be the name Chapter 1 in *dest.pdf*. The file *dest.pdf* itself contains the information specifying where the location named Chapter 1 is within the file.
- document info dictionary entries. The document info dictionary contains information such as title, creation date, author, and keywords, and other optional strings. These can be used by the Acrobat Search plug-in to restrict the documents searched, for example, to search only those with a particular author.

setdistillerparams enhancements allow control over:

- antialiasing. You can enable or disable antialiasing. Antialiasing retains some of the information that would otherwise be lost by downsampling. It does so by increasing the number of bits per color component in an image (color, grayscale, or monochrome) when downsampling the image.
- font subsetting, which embeds a partial font instead of a complete font. The partial font contains only those glyphs that were used in the PDF file, instead of the entire font. You can enable or disable subsetting, and specify the maximum percentage of glyphs in a font that can be used before the entire font is embedded instead of only a subset.
- the conversion of CMYK images to RGB images. Images that are specified in the four-component CMYK color space can optionally be converted to the three-component RGB color space. This reduces the size of a file, and is generally worthwhile unless the file is specifically intended to print on a CMYK printer.

With these enhancements, you can specify almost any PDF feature in the PostScript file your application produces. This allows your application to produce full-featured electronic documents that take advantage of the features of the Acrobat products. For more information on **pdfmark** and **setdistillerparams**, see the technical notes that are contained in the Acrobat SDK and Acrobat Plug-Ins SDK.

Questions Answers

Q Section 5.7.1 of the *PostScript Language Reference Manual, Second Edition* states that "Before executing the graphics operators that describe the character, BuildGlyph **must execute one of the following operators to pass width and bounding box information to the PostScript interpreter:** **setcachedevice ... setcachedevice2 ... setcharwidth..." In the** BuildGlyph **procedure for the space character where no graphics operators are executed, can I omit calling one of these operators?**

A The answer to your question is "no". This is not clear from reading section 5.7.1; however, you must always execute one of the operators listed above in your BuildGlyph procedure. The same applies to BuildChar.

The reason for this requirement is that a call to **setcachedevice** or **setcharwidth** is necessary to tell the font machinery what the character's width is. There is no explicit default width provided, so the result of not making this call would be unpredictable. And furthermore, the character won't get cached.

Q I observed that when a PostScript language file is converted to PDF format by Acrobat Distiller, the resulting PDF file is usually smaller than the original PostScript language file. I am curious why that is the case.

A In general, PDF files are smaller than the PostScript language files that were used to generate them, sometimes many times smaller.

Here are the reasons why a PDF is usually smaller than the original PostScript language file:

a) The bitmap images in PostScript language files are usually for high resolution output devices, ranging from 300 dpi up to a couple thousand dpi, while the PDF files are often intended mainly for display on computer screens with resolution around 100 dpi . An option is provided in the Distiller for authors to downsample images when generating PDF. This can dramatically decrease file size, at the cost of removing high-resolution detail.

b) PDF supports PostScript Level 2's image compression features, and uses them by default. If the original PostScript language file was Level 1, or if Level 2 compression features were not used, the resulting PDF file can be dramatically smaller even if downsampling is not done.

c) By default, the body text of a PDF file is also compressed. This is almost never true of PostScript language files. This is independent of whether the file's images are compressed and/or downsampled.

d) The Distiller application optimizes incoming PostScript language files. For example, many applications generate PostScript language files in which a **show** is done for individual letters or for short word fragments. To the extent possible, the Acrobat Distiller program combines these short **show** strings into larger strings with fewer calls to the **show** operator.

In addition, some applications generate unnecessary sequences of graphics state operations; for example, setting the same color or line style more than once. The Distiller application omits many of these unnecessary calls to graphics state operators.

e) PDF operator names are shorter than standard PostScript operator names; for example, PDF uses 'm' for **moveto**. The significance of this depends on the prolog used by a given application and platform. For example, the Adobe Illustrator software's prolog also defines short operator names, so the file size reduction would not be as pronounced in the case of an Illustrator-generated file.

f) PostScript language files can contain a number of procedure definitions; however, many of the procedures are often not used. The Distiller program omits the unused PostScript language code.

A commonly-cited example that shows the size reduction from the PostScript language to PDF is one of Adobe's annual reports. It was produced in a page layout program, saved as a PostScript language file, and distilled into a PDF file. The original file was around 60 megabytes; the PDF file was under 1 MB and fit on a diskette. The reason is largely

image downsampling; only screen-resolution images were retained in the PDF file. This "60:1" result is probably only typical of image-rich files which are heavily downsampled.

All of the above factors play a role in decreasing the file size of distilled documents; however, PDF files can sometimes be bigger than the original PostScript files. A file size reduction is not guaranteed for all cases for the following reasons:

a) Inappropriate image compression—Each file compression algorithm is appropriate for a particular type of data. JPEG works best on low frequency images, such as smooth images with few edges; fax compression methods are optimized for 1-bit images with large areas of white space. If an inappropriate compression method is chosen, the resulting image can be substantially larger than the original.

b) Loop unwinding—PostScript is a programming language; PDF is only a "marking language". PostScript programs may use loops to describe illustrations composed of repeated objects. For example, a PostScript program might say in effect "draw a star 100 times, moving a little right and down each time". PDF would use individual commands to draw each of the 100 stars.

c) Another important factor that can make a PDF file larger or smaller than the original PostScript language file is font embedding. Depending on whether the PDF or the PostScript language file embeds fonts, either format may be larger or smaller. Each font may occupy up to 50 Kbytes in a file. PDFWriter and Distiller give users the choice of embedding fonts in a PDF file. If a font is not embedded, its metrics are included instead. The size of a font's metrics is about 4 Kbyte.

In addition, the Acrobat viewers assume the existence of the base 13* fonts, so PDFWriter and the Distiller application never embed these when generating PDF.

To summarize, a PDF file can be bigger or smaller than the original PostScript language file depending on the contents of the original file and the settings used to distill it.

* *The base 13 fonts are Times-Roman, Times-Italic, Times-Bold, Times-BoldItalic, Helvetica, Helvetica-Oblique, Helvetica-Bold, Helvetica-BoldOblique, Courier, Courier-Oblique, Courier-Bold, Courier-BoldOblique, and Symbol.*

Colophon

All proofs and final output for this newsletter were produced using Adobe PostScript software. The document review process was accomplished via electronic distribution using Adobe Acrobat software. Typefaces used are from the Minion™ and Myriad™ families from the Adobe Type Library.

Managing Editors:
Nicole Frees, Debi Hamrick

Technical Editor:
Jim DeLaHunt

Art Director:
Karla Wong

Designer:
Lorsen Koo

Contributors:
Tim Bienz, Matt Foster, Michelle Sellars, Sun-Inn Shih

Part Number ADA0053 11/94

Adobe PostScript