

CHAPTER 2

Overview

Adobe PDF is a file format for representing documents in a manner independent of the application software, hardware, and operating system used to create them and of the output device on which they are to be displayed or printed. A *PDF document* consists of a collection of *objects* that together describe the appearance of one or more *pages*, possibly accompanied by additional interactive elements and higher-level application data. A *PDF file* contains the objects making up a PDF document along with associated structural information, all represented as a single self-contained sequence of bytes.

A document's pages (and other visual elements) can contain any combination of text, graphics, and images. A page's appearance is described by a PDF *content stream*, which contains a sequence of *graphics objects* to be painted on the page. This appearance is fully specified; all layout and formatting decisions have already been made by the application generating the content stream.

In addition to describing the static appearance of pages, a PDF document can contain interactive elements that are possible only in an electronic representation. PDF supports *annotations* of many kinds for such things as text notes, hypertext links, markup, file attachments, sounds, and movies. A document can define its own user interface; keyboard and mouse input can trigger *actions* that are specified by PDF objects. The document can contain *interactive form* fields to be filled in by the user, and can export the values of these fields to or import them from other applications.

Finally, a PDF document can contain higher-level information that is useful for interchange of content among applications. In addition to specifying appearance, a document's content can include identification and logical structure information

that allows it to be searched, edited, or extracted for reuse elsewhere. PDF is particularly well suited for representing a document as it moves through successive stages of a prepress production workflow.

2.1 Imaging Model

At the heart of PDF is its ability to describe the appearance of sophisticated graphics and typography. This ability is achieved through the use of the *Adobe imaging model*, the same high-level, device-independent representation used in the PostScript page description language.

Although application programs could theoretically describe any page as a full-resolution pixel array, the resulting file would be bulky, device-dependent, and impractical for high-resolution devices. A high-level imaging model enables applications to describe the appearance of pages containing text, graphical shapes, and sampled images in terms of abstract graphical elements rather than directly in terms of device pixels. Such a description is economical and device-independent, and can be used to produce high-quality output on a broad range of printers, displays, and other output devices.

2.1.1 Page Description Languages

Among its other roles, PDF serves as a *page description language*, a language for describing the graphical appearance of pages with respect to an imaging model. An application program produces output through a two-stage process:

1. The application generates a device-independent description of the desired output in the page description language.
2. A program controlling a specific output device interprets the description and *renders* it on that device.

The two stages may be executed in different places and at different times. The page description language serves as an interchange standard for the compact, device-independent transmission and storage of printable or displayable documents.

2.1.2 Adobe Imaging Model

The Adobe imaging model is a simple and unified view of two-dimensional graphics borrowed from the graphic arts. In this model, “paint” is placed on a page in selected areas:

- The painted figures can be in the form of character shapes (*glyphs*), geometric shapes, lines, or sampled images such as digital representations of photographs.
- The paint may be in color or in black, white, or any shade of gray. It may also take the form of a repeating *pattern* (PDF 1.2) or a smooth transition between colors (PDF 1.3).
- Any of these elements may be *clipped* to appear within other shapes as they are placed onto the page.

A page’s content stream contains *operands* and *operators* describing a sequence of graphics objects. A PDF consumer application maintains an implicit *current page* that accumulates the marks made by the painting operators. Initially, the current page is completely blank. For each graphics object encountered in the content stream, the application places marks on the current page, which replace or combine with any previous marks they may overlay. Once the page has been completely composed, the accumulated marks are rendered on the output medium and the current page is cleared to blank again.

PDF 1.3 and earlier versions use an *opaque imaging model* in which each new graphics object painted onto a page completely obscures the previous contents of the page at those locations (subject to the effects of certain optional parameters that may modify this behavior; see Section 4.5.6, “Overprint Control”). No matter what color an object has—white, black, gray, or color—it is placed on the page as if it were applied with opaque paint. PDF 1.4 introduces a *transparent imaging model* in which objects painted on the page are not required to be fully opaque. Instead, newly painted objects are *composited* with the previously existing contents of the page, producing results that combine the colors of the object and its backdrop according to their respective opacity characteristics. The transparent imaging model is described in Chapter 7.

The principal graphics objects (among others) are as follows:

- A *path object* consists of a sequence of connected and disconnected points, lines, and curves that together describe shapes and their positions. It is built up

through the sequential application of *path construction operators*, each of which appends one or more new elements. The path object is ended by a *path-painting operator*, which paints the path on the page in some way. The principal path-painting operators are **S** (stroke), which paints a line along the path, and **f** (fill), which paints the interior of the path.

- A *text object* consists of one or more glyph shapes representing characters of text. The glyph shapes for the characters are described in a separate data structure called a *font*. Like path objects, text objects can be stroked or filled.
- An *image object* is a rectangular array of *sample values*, each representing a color at a particular position within the rectangle. Such objects are typically used to represent photographs.

The painting operators require various parameters, some explicit and others implicit. Implicit parameters include the current color, current line width, current font (typeface and size), and many others. Together, these implicit parameters make up the *graphics state*; there are operators for setting the value of each implicit parameter in the graphics state. Painting operators use the values currently in effect at the time they are invoked.

One additional implicit parameter in the graphics state modifies the results of painting graphics objects. The *current clipping path* outlines the area of the current page within which paint can be placed. Although painting operators may attempt to place marks anywhere on the current page, only those marks falling within the current clipping path affect the page; those falling outside it do not affect the page. Initially, the current clipping path encompasses the entire imageable area of the page. It can temporarily be reduced to the shape defined by a path or text object, or to the intersection of multiple such shapes. Marks placed by subsequent painting operators are confined within that boundary.

2.1.3 Raster Output Devices

Much of the power of the Adobe imaging model derives from its ability to deal with the general class of *raster output devices*. These encompass such technologies as laser, dot-matrix, and ink-jet printers, digital imagesetters, and raster-scan displays. The defining property of a raster output device is that a printed or displayed image consists of a rectangular array, or *raster*, of dots called *pixels* (picture elements) that can be addressed individually. On a typical *bilevel* output device, each pixel can be made either black or white. On some devices, pixels can be set to intermediate shades of gray or to some color. The ability to set the colors of

individual pixels makes it possible to generate printed or displayed output that can include text, arbitrary graphical shapes, and reproductions of sampled images.

The *resolution* of a raster output device measures the number of pixels per unit of distance along the two linear dimensions. Resolution is typically—but not necessarily—the same horizontally and vertically. Manufacturers' decisions on device technology and price/performance tradeoffs create characteristic ranges of resolution:

- Computer displays have relatively low resolution, typically 75 to 110 pixels per inch.
- Dot-matrix printers generally range from 100 to 250 pixels per inch.
- Ink-jet and laser-scanned xerographic printing technologies achieve medium-level resolutions of 300 to 1400 pixels per inch.
- Photographic technology permits high resolutions of 2400 pixels per inch or more.

Higher resolution yields better quality and fidelity of the resulting output but is achieved at greater cost. As the technology improves and computing costs decrease, products evolve to higher resolutions.

2.1.4 Scan Conversion

An abstract graphical element (such as a line, a circle, a character glyph, or a sampled image) is rendered on a raster output device by a process known as *scan conversion*. Given a mathematical description of the graphical element, this process determines which pixels to adjust and what values to assign to those pixels to achieve the most faithful rendition possible at the available device resolution.

The pixels on a page can be represented by a two-dimensional array of pixel values in computer memory. For an output device whose pixels can only be black or white, a single bit suffices to represent each pixel. For a device that can reproduce gray levels or colors, multiple bits per pixel are required.

Note: *Although the ultimate representation of a printed or displayed page is logically a complete array of pixels, its actual representation in computer memory need not consist of one memory cell per pixel. Some implementations use other representa-*

tions, such as display lists. The Adobe imaging model has been carefully designed not to depend on any particular representation of raster memory.

For each graphical element that is to appear on the page, the scan converter sets the values of the corresponding pixels. When the interpretation of the page description is complete, the pixel values in memory represent the appearance of the page. At this point, a raster output process can *render* this representation (make it visible) on a printed page or display screen.

Scan-converting a graphical shape, such as a rectangle or circle, entails determining which device pixels lie inside the shape and setting their values appropriately (for example, to black). Because the edges of a shape do not always fall precisely on the boundaries between pixels, some policy is required for deciding how to set the pixels along the edges. Scan-converting a glyph representing a text character is conceptually the same as scan-converting an arbitrary graphical shape. However, character glyphs are much more sensitive to legibility requirements and must meet more rigid objective and subjective measures of quality.

Rendering grayscale elements on a bilevel device is accomplished by a technique known as *halftoning*. The array of pixels is divided into small clusters according to some pattern (called the *halftone screen*). Within each cluster, some pixels are set to black and others to white in proportion to the level of gray desired at that location on the page. When viewed from a sufficient distance, the individual dots become imperceptible and the perceived result is a shade of gray. This enables a bilevel raster output device to reproduce shades of gray and to approximate natural images such as photographs. Some color devices use a similar technique.

2.2 Other General Properties

This section describes other notable general properties of PDF, aside from its imaging model.

2.2.1 Portability

PDF files are represented as sequences of 8-bit binary bytes. A PDF file is designed to be portable across all platforms and operating systems. The binary representation is intended to be generated, transported, and consumed directly, without translation between native character sets, end-of-line representations, or other conventions used on various platforms.

Any PDF file can also be represented in a form that uses only 7-bit ASCII (American Standard Code for Information Interchange) character codes. This is useful for the purpose of exposition, as in this book. However, this representation is not recommended for actual use, since it is less efficient than the normal binary representation. Regardless of which representation is used, PDF files must be transported and stored as binary files, not as text files. Inadvertent changes, such as conversion between text end-of-line conventions, will damage the file and may render it unusable.

2.2.2 Compression

To reduce file size, PDF supports a number of industry-standard compression filters:

- JPEG and (in PDF 1.5) JPEG2000 compression of color and grayscale images
- CCITT (Group 3 or Group 4), run-length, and (in PDF 1.4) JBIG2 compression of monochrome images
- LZW (Lempel-Ziv-Welch) and (beginning with PDF 1.2) Flate compression of text, graphics, and images

Using JPEG compression, color and grayscale images can be compressed by a factor of 10 or more. Effective compression of monochrome images depends on the compression filter used and the properties of the image, but reductions of 2:1 to 8:1 are common (or 20:1 to 50:1 for JBIG2 compression of an image of a page full of text). LZW or Flate compression of the content streams describing all other text and graphics in the document results in compression ratios of approximately 2:1. All of these compression filters produce binary data, which can be further converted to ASCII base-85 encoding if a 7-bit ASCII representation is required.

2.2.3 Font Management

Managing fonts is a fundamental challenge in document interchange. Generally, the receiver of a document must have the same fonts that were originally used to create it. If a different font is substituted, its character set, glyph shapes, and metrics may differ from those in the original font. This substitution can produce unexpected and unwanted results, such as lines of text extending into margins or overlapping with graphics.

PDF provides various means for dealing with font management:

- The original font programs can be embedded in the PDF file, which ensures the most predictable and dependable results. PDF supports various font formats, including Type 1, TrueType[®], OpenType, and CID-keyed fonts.
- To conserve space, a font subset can be embedded, containing just the glyph descriptions for those characters that are actually used in the document. Also, Type 1 fonts can be represented in a special compact format.
- PDF prescribes a set of 14 standard fonts that can be used without prior definition. These include four faces each of three Latin text typefaces (Courier, Helvetica*, and Times*), as well as two symbolic fonts (Symbol and ITC Zapf Dingbats[®]). These fonts, or suitable substitute fonts with the same metrics, are required to be available in all PDF consumer applications.
- A PDF file can refer by name to fonts that are not embedded in the PDF file. In this case, a PDF consumer can use those fonts if they are available in its environment. This approach suffers from the uncertainties noted above.
- A PDF file contains a *font descriptor* for each font that it uses (other than the standard 14). The font descriptor includes font metrics and style information, enabling an application to select or synthesize a suitable substitute font if necessary. Although the glyphs' shapes differ from those intended, their placement is accurate.

Font management is primarily concerned with producing the correct appearance of text—that is, the shape and placement of glyphs. However, it is sometimes necessary for a PDF application to extract the meaning of the text, represented in some standard information encoding such as Unicode. In some cases, this information can be deduced from the encoding used to represent the text in the PDF file. Otherwise, the PDF producer application should specify the mapping explicitly by including a special object, the **ToUnicode CMap**.

2.2.4 Single-Pass File Generation

Because of system limitations and efficiency considerations, it may be necessary or desirable for an application program to generate a PDF file in a single pass. For example, the program may have limited memory available or be unable to open temporary files. For this reason, PDF supports single-pass generation of files. Although some PDF objects must specify their length in bytes, a mechanism is provided allowing the length to follow the object in the PDF file. In addition, in-

formation such as the number of pages in the document can be written into the file after all pages have been generated.

A PDF file that is generated in a single pass is generally not ordered for most efficient viewing, particularly when accessing the contents of the file over a network. When generating a PDF file that is intended to be viewed many times, it is worthwhile to perform a second pass to optimize the order in which objects occur in the file. PDF specifies a particular file organization, *Linearized PDF*, which is documented in Appendix F. Other optimizations are also possible, such as detecting duplicated sequences of graphics objects and collapsing them to a single shared sequence that is specified only once.

2.2.5 Random Access

A PDF file should be thought of as a flattened representation of a data structure consisting of a collection of objects that can refer to each other in any arbitrary way. The order of the objects' occurrence in the PDF file has no semantic significance. In general, an application should process a PDF file by following references from object to object, rather than by processing objects sequentially. This is particularly important for interactive document viewing or for any application in which pages or other objects in the PDF file are accessed out of sequence.

To support such random access to individual objects, every PDF file contains a *cross-reference table* that can be used to locate and directly access pages and other important objects within the file. The cross-reference table is stored at the end of the file, allowing applications that generate PDF files in a single pass to store it easily and those that read PDF files to locate it easily. By using the cross-reference table, the time needed to locate a page or other object is nearly independent of the length of the document, allowing PDF documents containing hundreds or thousands of pages to be accessed efficiently.

2.2.6 Security

PDF has two security features that can be used, separately or together, in any document:

- The document can be *encrypted* so that only authorized users can access it. There is separate authorization for the owner of the document and for all other

users; the users' access can be selectively restricted to allow only certain operations, such as viewing, printing, or editing.

- The document can be digitally *signed* to certify its authenticity. The signature may take many forms, including a document digest that has been encrypted with a public/private key, a biometric signature such as a fingerprint, and others. Any subsequent changes to a signed PDF file invalidate the signature.

2.2.7 Incremental Update

Applications may allow users to modify PDF documents. Users should not have to wait for the entire file—which can contain hundreds of pages or more—to be rewritten each time modifications to the document are saved. PDF allows modifications to be appended to a file, leaving the original data intact. The addendum appended when a file is incrementally updated contains only those objects that were actually added or modified, and includes an update to the cross-reference table. Incremental update allows an application to save modifications to a PDF document in an amount of time proportional to the size of the modification rather than the size of the file.

In addition, because the original contents of the document are still present in the file, it is possible to undo saved changes by deleting one or more addenda. The ability to recover the exact contents of an original document is critical when digital signatures have been applied and subsequently need to be verified.

2.2.8 Extensibility

PDF is designed to be extensible. Not only can new features be added, but applications based on earlier versions of PDF can behave reasonably when they encounter newer features that they do not understand. Appendix H describes how a PDF consumer application should behave in such cases.

Additionally, PDF provides means for applications to store their own private information in a PDF file. This information can be recovered when the file is imported by the same application, but it is ignored by other applications. Therefore, PDF can serve as an application's native file format while its documents can be viewed and printed by other applications. Application-specific data can be stored either as *marked content* annotating the graphics objects in a PDF content stream or as entirely separate objects unconnected with the PDF content.

2.3 Creating PDF

PDF files may be produced either directly by application programs or indirectly by conversion from other file formats or imaging models. As PDF documents and applications that process them become more prevalent, new ways of creating and using PDF will be invented. One of the goals of this book is to make the file format accessible so that application developers can expand on the ideas behind PDF and the applications that initially support it.

Many applications can generate PDF files directly, and some can import them as well. This direct approach is preferable, since it gives the application access to the full capabilities of PDF, including the imaging model and the interactive and document interchange features. Alternatively, applications that do not generate PDF directly can produce PDF output indirectly. There are two principal indirect methods:

- The application describes its printable output by making calls to an application programming interface (API) such as GDI in Microsoft® Windows® or QuickDraw in the Apple® Mac® OS. A software component called a *printer driver* intercepts these calls and interprets them to generate output in PDF form.
- The application produces printable output directly in some other file format, such as PostScript, PCL, HPGL, or DVI, which is converted to PDF by a separate translation program.

Although these indirect strategies are often the easiest way to obtain PDF output from an existing application, the resulting PDF files may not make the best use of the high-level Adobe imaging model. This is because the information embodied in the application's API calls or in the intermediate output file often describes the desired results at too low a level. Any higher-level information maintained by the original application has been lost and is not available to the printer driver or translator.

Figures 2.1 and 2.2 show how Acrobat products support these indirect approaches. The Adobe PDF printer (Figure 2.1), available on the Windows and Mac OS platforms, acts as a printer driver, intercepting graphics and text operations generated by a running application program through the operating system's API. Instead of converting these operations into printer commands and transmitting them directly to a printer, Adobe PDF converts them to equivalent PDF operators and embeds them in a PDF file. The result is a platform-independent file that can be viewed and printed by a PDF viewer application, such as Acrobat,

running on any supported platform—even a different platform from the one on which the file was originally generated.

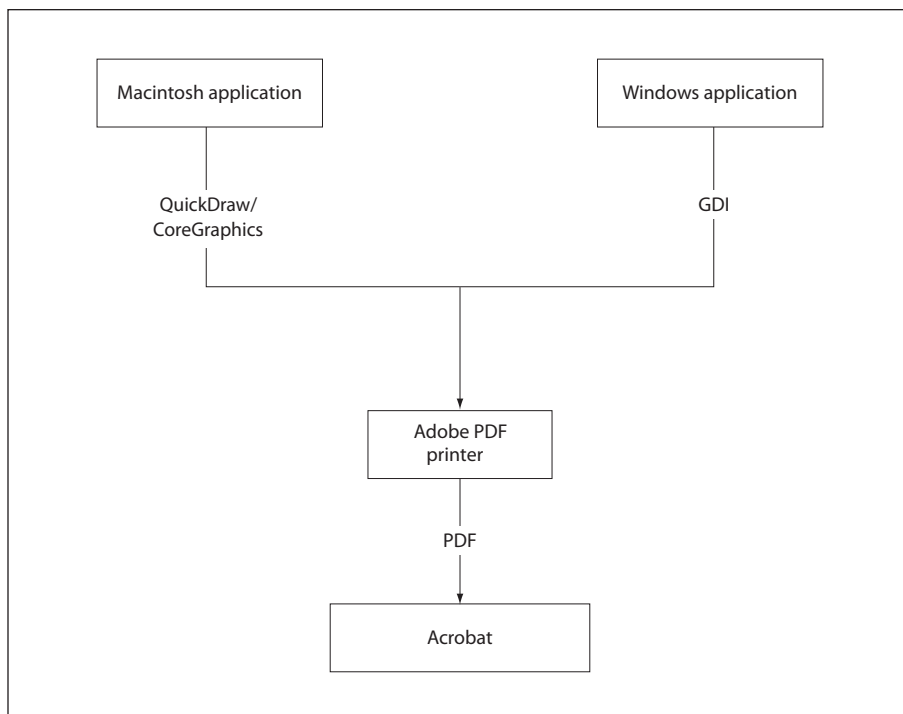


FIGURE 2.1 *Creating PDF files using the Adobe PDF printer*

Instead of describing their printable output through API calls, some applications produce PostScript page descriptions directly—either because of limitations in the QuickDraw or GDI imaging models or because the applications run on platforms such as DOS or UNIX®, where no system-level printer driver exists. PostScript files generated by such applications can be converted to PDF files using the Acrobat Distiller® application (see Figure 2.2). Because PostScript and PDF share the same Adobe imaging model, Distiller can preserve the exact graphical content of the PostScript file in the translation to PDF. Additionally, Distiller supports a PostScript language extension, called **pdfmark**, that allows the producing application to embed instructions in the PostScript file for creating hypertext links, logical structure, and other interactive and document interchange features of PDF.

Again, the resulting PDF file can be viewed with a viewer application, such as Acrobat, on any supported platform.

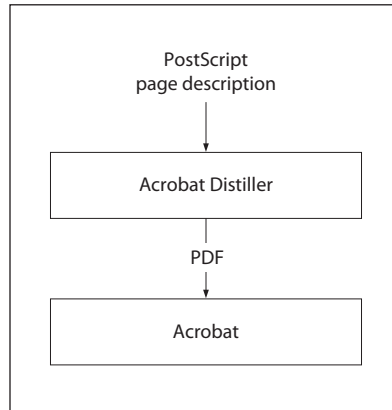


FIGURE 2.2 *Creating PDF files using Acrobat Distiller*

2.4 PDF and the PostScript Language

The PDF operators for setting the graphics state and painting graphics objects are similar to the corresponding operators in the PostScript language. Unlike PostScript, however, PDF is not a full-scale programming language; it trades reduced flexibility for improved efficiency and predictability. PDF therefore differs from PostScript in the following significant ways:

- PDF enforces a strictly defined file structure that allows an application to access parts of a document in arbitrary order.
- To simplify the processing of content streams, PDF does not include common programming language features such as procedures, variables, and control constructs.
- PDF files contain information such as font metrics to ensure viewing fidelity.
- A PDF file may contain additional information that is not directly connected with the imaging model, such as hypertext links for interactive viewing and logical structure information for document interchange.

Because of these differences, a PDF file generally cannot be transmitted directly to a PostScript output device for printing (although a few such devices do also

support PDF directly). An application printing a PDF document to a PostScript device must follow these steps:

1. Insert *procedure sets* containing PostScript procedure definitions to implement the PDF operators.
2. Extract the content for each page. Each content stream is essentially the script portion of a traditional PostScript program using very specific procedures, such as **m** for **moveto** and **l** for **lineto**.
3. Decode compressed text, graphics, and image data as necessary. The compression filters used in PDF are compatible with those used in PostScript; they may or may not be supported, depending on the LanguageLevel of the target output device.
4. Insert any needed resources, such as fonts, into the PostScript file. These can be either the original fonts or suitable substitute fonts based on the font metrics in the PDF file. Fonts may need to be converted to a format that the PostScript interpreter recognizes, such as Type 1 or Type 42.
5. Put the information in the correct order. The result is a traditional PostScript program that fully represents the visual aspects of the document but no longer contains PDF elements such as hypertext links, annotations, and bookmarks.
6. Transmit the PostScript program to the output device.