PREVIOUS CONTENTS NEXT

Chapter 1. An Introduction to PNG

Contents:

PNG,[1] short for ``Portable Network Graphics,'' is a computer file format for storing, transmitting, and displaying images. Similar to the GIF and TIFF image formats--in fact, designed to replace them in many applications--PNG supports lossless compression, transparency information, and a range of color depths. PNG also supports more advanced features such as gamma correction and a standard color space for precise reproduction of image colors on a wide range of systems and embedded textual information for storing such things as a title, the author's name, and explicit copyright.

> [1] PNG is officially pronounced ``ping'' (at least in English) but never spelled that way. Yes, this was a major topic of discussion during its design, and it is explicitly noted in the specification. Believe it or not, in November 1998 the issue once again came under discussion, this time with greater emphasis on non-English pronunciation. Though the ``three-letter'' approach (i.e., *P-N-G* spoken as three separate letters) was not approved for inclusion in the spec, it may be considered an acceptable unofficial alternative.

In this chapter, we'll consider PNG from the perspective of a user who has some familiarity with the process of creating and using computer images, but insufficient knowledge of the technical differences between various formats to be certain when to use what. I won't dwell on features that are mostly of concern to developers; where I do bring up programming issues, it is principally to explain to the *user* why some software may not perform as well as expected. I'll concentrate on two areas to which PNG is particularly well suited: as an intermediate editing format for repeatedly saving and restoring images without loss, and as a final display format for the World Wide Web. And I'll finish up with an in-depth look at one application that has particularly good PNG support: Macromedia's Fireworks 1.0, an image-editing program specifically designed for creating web images.

1.1. Overview of Image Properties

Before we dive right into some of PNG's more interesting features, it might be helpful to introduce (or review) some essential image concepts and take a quick look at a few older image formats. Those who are already familiar with the most basic features of computer images can skip directly to the next section.

There are two main formats for computer images: raster, based on colored dots, which are almost always stored in a rectangular array and are usually packed so close together that individual dots are no longer distinguishable, and vector, based on lines, circles, and other ``primitive'' elements that typically cover a

sizable area and are easily distinguishable from one another. Many images can be represented in either format; indeed, any vector-based image can be approximated by a raster image (lots of dots), and one could easily (though tediously) simulate a raster image in vector format by converting each dot to a tiny box.

The whole point of having two classes of image formats--and, indeed, of having numerous individual file formats--is implicit in the old saying, ``Use the best tool for the job.'' Vector formats are appropriate for simple graphics and text, such as corporate logos, and their advantage is that they can be extremely compact and yet maintain perfect sharpness regardless of the size at which they are reproduced. But with the exception of pen-based plotters and some ancient vector-based displays, the end result is almost always a raster image.

For that reason, plus the fact that raster image formats are more common--and because PNG is one of them--we'll take a closer look at raster features. As I just noted, a raster image is composed of an array of dots, more commonly referred to as *pixels* (short for *picture elements*). One generally refers to a computer image's dimensions in terms of pixels; this is also often (though slightly imprecisely) known as its *resolution*. Some common image sizes are 640 × 480, 800 × 600, and 1024 × 768 pixels, which also happen to be common dimensions for computer displays.

In addition to horizontal and vertical dimensions, a raster image is characterized by depth. The deeper the image, the more colors (or shades of gray) it can have. Pixel depths are measured in *bits*, the tiniest units of computer storage; a 1-bit image can represent two colors (often, though not necessarily, black and white), a 2-bit image four colors, an 8-bit image 256 colors, and so on. To calculate the raw size of the image data before any compression takes place, one needs only to know that 8 bits make a byte. Thus a 320 × 240, 24-bit image has 76,800 pixels, each of which is 3 bytes deep, so its total uncompressed size is 230,400 bytes.

I'll return to the topic of compression in just a moment; first, let's take a closer look at the precise relationship between pixels and colors. Within the broad class of raster formats, there are three main image types: indexed-color, grayscale, and truecolor. The *indexed-color* method, also known as *pseudocolor*, *colormapped*, or *palette-based*, stores a copy of each color value needed for the image in a palette. The main image is then composed of index values referring to different entries in the palette. For example, imagine an image composed entirely of red, white, and blue pixels; the palette would have three entries corresponding to these colors, and each pixel would be represented by the value 0, 1, or 2. (The natural starting point for numbers on a computer is 0, not 1.) Since an image 2 bits deep can represent up to four colors, each pixel in this example would require only 2 bits, even though the precise shades of red, white, and blue might ordinarily require 24 bits each.

Grayscale and truecolor images are simpler in concept; the bytes used by each pixel correspond directly to shades of gray or to colors. In a *grayscale* image of a particular pixel depth, a 0 pixel usually (though not always) means black, while the maximum value at that depth corresponds to white. Intermediate pixel values are smoothly interpolated to shades of gray, though this is often not as straightforward as it might sound--*gamma correction*, a way of adjusting for differences in computer display systems, comes in here. I'll give a brief overview of gamma correction later in this chapter, and I'll discuss it at length in Chapter 10, "Gamma Correction and Precision Color", *Gamma Correction and Precision Color*; for now, I'll merely note that it is a Good Thing, and image formats that provide support for it can be viewed on different platforms without appearing too light on one and too dark on another.

A *truecolor* image uses three separate values for each pixel, corresponding to shades of red, green, and blue. Such images are often also referred to as *RGB*. In Chapter 8, "PNG Basics", I'll talk about human vision and the reasons why mixtures of just three colors can appear to reproduce all colors, or at least a sufficiently large percentage of them that one need not quibble over the difference. I'll also mention some common alternatives to the RGB *color space*. To be considered truly truecolor instead of merely ``high color,'' an image must contain at least 8 bits for each of the three colors in each pixel; thus, at a minimum, a truecolor image has a depth of 24 bits.

Two other concepts--samples and channels--are handy when speaking of images, and RGB images are a good way to illustrate these concepts. A *sample* is one component of a single color value. For example, each pixel in a truecolor image consists of three samples: red, green, and blue. If the image is 24 bits deep, then each sample is 8 bits deep. A 256-shade grayscale image also has 8-bit samples, which means that one can

speak of the ``bits per sample'' for either image type to indicate the level of precision of each shade or color. Note that I have been careful to distinguish between *sample depth* and *pixel depth*. The two terms are directly related in grayscale and truecolor images, but in indexed-color images they can be independent of each other. This is because the sample depth refers to the color values in the palette, while the pixel depth refers to the index values of each pixel (which reference the palette colors). To put it more concretely, the color values in the palette are usually 24-bit values (8 bits per sample), but the pixel indices are usually 8 bits or less. Our previous red, white, and blue example used only two bits per pixel.

A *channel*, on the other hand, refers to the collection of all samples of a given type in an image--for example, the green components of every RGB pixel. Thus a truecolor image has three channels, while a grayscale image has only one. (Ordinarily one does not speak of a palette-based image as having channels.) And when discussing transparency, yet another channel type is often used: the *alpha channel*. This is a special kind of channel in that it does not provide actual color information but rather a level of transparency for each pixel--or, more precisely, a level of *opacity*, since it is most common for the maximum sample value to indicate that the pixel is completely opaque and for zero to indicate complete transparency. A truecolor image with an alpha channel is often called an RGBA image; grayscale images with alpha channels are rarer and don't have a special abbreviation (although I may refer to them as ``gray+alpha'').

Palette-based images almost never have a full alpha channel, but another type of transparency is possible. Rather than associate alpha information with every pixel, one can instead associate it with specific palette entries. By far the most common approach is to specify that a single palette entry represents complete transparency. Then when the image is displayed against some sort of background, any pixel whose index refers to this particular palette entry will be replaced by the background at the pixel's location--or perhaps the pixel simply will not be drawn in the first place. But there is no conceptual requirement that only one palette entry can have transparency, nor that it must be fully transparent. As we'll see shortly, PNG effectively allows any number of palette entries to have any level of transparency.

While we're on the subject of colormapped images, two other concepts are worth mentioning: quantization and dithering. Suppose one has a 24-bit truecolor image, but it must be displayed on a 256-color, palette-based display. Since truecolor images typically use anywhere from 10,000 to 100,000 colors, the conversion to a colormapped image will involve substituting many of the color values with a much smaller range of colors. This process is known as *quantization*. Because the resulting images have such a limited palette of colors available to them, they often are unable to represent fine color gradients such as the different shades of blue seen in the sky or the range of facial tones in a softly lit portrait. One way around this is to *dither* the image, which is a means of mixing pixels of the available colors together to give the appearance of other colors (though generally at the cost of some sharpness). For example, a checkerboard pattern of alternating red and yellow pixels might appear orange. This effect is perhaps best illustrated with an example. Figure 1-1 shows a truecolor photograph (here rendered in grayscale) together with two 256-color versions of the same image--one simply quantized to 256 colors and the other both quantized and dithered. The insets give a magnified view of one region, showing the relative effects of the two procedures.

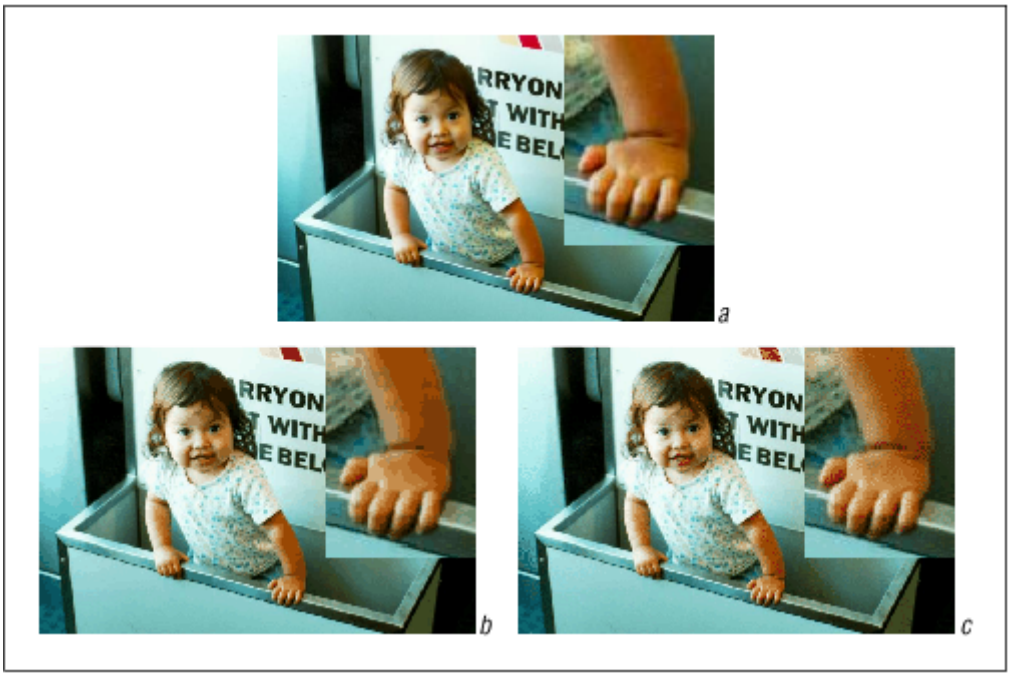

Figure 1-1: *(a) Original, 24-bit image; (b) same image after quantization, and (c) after quantization and dithering.* (Click on images for full-scale, color versions.)

I'll round out our review of image properties and concepts with a quick look at compression. There are really only two flavors: lossless and lossy. *Lossless* compression preserves the exact image data down to the last bit, so that what you get out after uncompressing is exactly the same as what you started with. In contrast, *lossy* compression throws away some of the data in return for much better compression ratios. For photographic images, the best lossless methods may only manage a factor of two or three in compression, whereas lossy methods typically achieve anywhere from 8 to 25 times reduction with very little visible loss of quality. I'll discuss the details of compression, particularly the lossless variety, at greater length in Chapter 9, "Compression and Filtering".

Finally, in describing the advantages of PNG, I will necessarily compare it with some older image formats. Although there are literally hundreds of different formats, we will be most concerned with just three: GIF, JPEG, and TIFF. GIF, short for the Graphics Interchange Format, and JPEG, short for the Joint Photographic Experts Group (which defined the format), are both very common image types often seen on the Web. TIFF, on the other hand, short for Tagged Image File Format, is almost never used on the Web but is quite popular as an output format from scanners and as an intermediate ``save format'' while editing images. I'll touch on the properties of each of these formats as we go.

1.2. What Is PNG Good For?

For image editing, either professional or otherwise, PNG provides a useful format for storing the intermediate stages of an image. Since PNG's compression is fully lossless--and since it supports up to 48-bit truecolor or 16-bit grayscale--saving, restoring, and resaving an image will not degrade its quality, unlike standard JPEG (even at its highest quality settings). PNG also supports full transparency information, unlike JPEG (no transparency at all), GIF (no partial transparency), or even TIFF (full transparency is part of the specification but is not required for minimal conformance). And unlike TIFF, which is probably the most popular intermediate format today, the PNG specification leaves almost no room for implementors to pick and choose what features they'll support. What allowances are made, such as optional support for gamma correction, are tightly constrained. The result is that a PNG image saved in one application is readable and displayable in any other PNG-supporting program.

For the Web, as of early 1999, there are two image formats with ubiquitous support: JPEG and GIF. JPEG is very well suited to the task for which it was designed--namely, the storage, transmission, and display of photorealistic 8-bit grayscale and 24-bit truecolor images with good quality and excellent compression--and PNG was never intended to compete with JPEG on its own terms. But PNG, like GIF, is more appropriate

than JPEG for images with few colors or with lots of sharp edges, such as cartoons or bitmapped text. PNG also provides direct support for gamma correction (loosely speaking, the cross-platform control of image ``brightness'') and transparency. I'll discuss these in more detail shortly.

GIF was the original cross-platform image format for the Web, and it is still a good choice in many respects. But PNG was specifically designed to replace GIF, and it has three main advantages over the older format: alpha channels (variable transparency), gamma correction, and two-dimensional interlacing (a method of displaying images at progressively higher levels of detail). PNG also compresses better than GIF in almost every case, but the difference is generally only around 5% to 25%, which is (usually) not a large enough factor to encourage one to switch on that basis alone. One GIF feature that PNG does *not* try to reproduce is multiple-image support, especially animations; PNG was and is intended to be a single-image format only. A very PNG-like extension format called MNG has been developed to address this limitation; it is discussed in Chapter 12, "Multiple-Image Network Graphics".

1.2.1. Alpha Channels

Also known as a *mask channel*, an alpha channel is simply a way to associate variable levels of transparency (sometimes referred to as ``translucency,'' though that may imply a diffuseness not present with alpha transparency) with an image. Whereas GIF supports simple binary transparency--any given pixel can be either fully transparent or fully opaque--PNG allows an additional 254 levels of partial transparency for ``normal'' images. It also supports a total of 65,536 transparency levels for the special ``deeply insane'' image types, but here we're concentrating on pixel depths that are useful on the Web.

All three of the basic PNG image types--RGB, grayscale, and palette-based--can have alpha information, but currently it's most often used with truecolor images. Instead of storing three bytes for every pixel, now four are required: red, green, blue, and alpha, or RGBA. The variable transparency allows one to create special effects that will look good on *any* background, whether light, dark, or patterned. For example, a photo-vignette effect can be created for a portrait by making a central oval region fully opaque (i.e., for the face and shoulders of the subject), the outer regions fully transparent, and a transition region that varies smoothly between the two extremes. When viewed with a web browser such as Acorn Browse or Arena, the portrait would fade smoothly to white when viewed against a white background or smoothly to black if against a black background. Both cases are shown in Figure 1-2.

Figure 1-2: *Portrait with an oval alpha mask (a) against a white background and (b) against a black background.* (Click on images for full-scale versions.)

This feature is especially important for the small web graphics that are typically used on web pages, such as colored (circular) bullets and fancy text. To avoid the jagged artifacts that really stand out on such images, most applications support *anti-aliasing*, a method for creating the illusion of smooth curves on a rectangular grid of pixels by smoothly varying the pixels' colors. The problem with anti-aliasing in the absence of variable transparency is that it must be done against a predetermined background color, typically either white or black. Reusing the same images on a different background usually results in an unpleasant ``halo'' effect, as shown in Figure 1-3. The standard approach is to create separate images for each background color used

on a site, but this has negative implications both for the designer, who wastes time creating and maintaining multiple copies of each image, and for visitors to the site, who must download those copies.

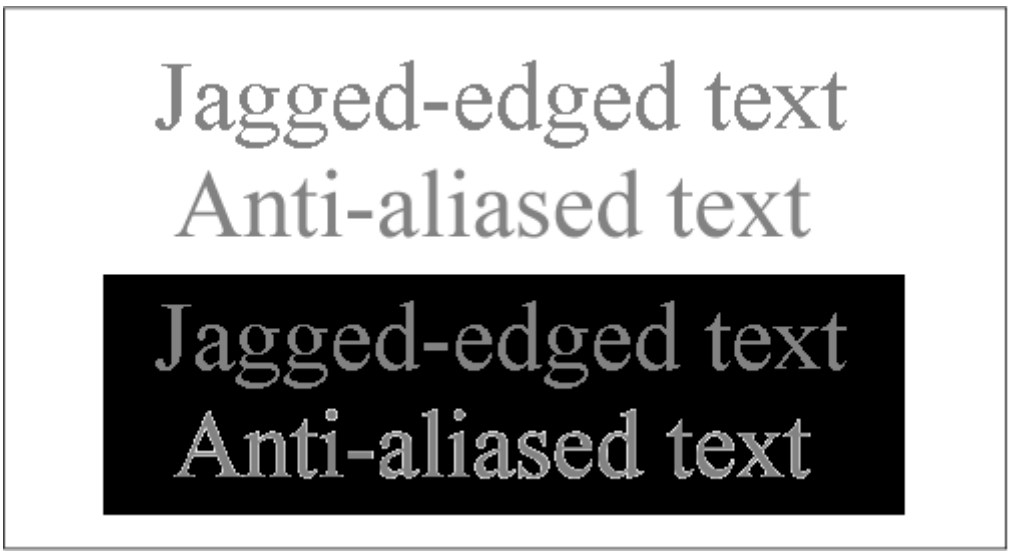

Figure 1-3: *Gray text anti-aliased against a white background, displayed against both white and black backgrounds.*

Alpha blending, on the other hand, effectively uses transparency as a placeholder for the background color. Fully transparent regions will inherit the background color as is; fully opaque regions will show up as the foreground images. This is no different from the usual case, exemplified by transparent GIFs. But the anti-aliased regions in between the fully transparent and fully opaque areas are no longer pre-mixed with an assumed background color; instead, they are partially transparent and can be mixed with whatever background on which the image happens to be placed.

Of course, effective replacements for GIF buttons and icons must not only be more useful but also of comparable or smaller size, and that mostly rules out truecolor RGBA images. Fortunately, PNG supports alpha information with palette images as well; it's just harder to implement in a smart way. A PNG alpha-palette image is just that: an image whose palette also has alpha information associated with it, not a palette image with a full alpha mask. In other words, each pixel corresponds to an entry in the palette with red, green, blue, *and* alpha components. So if you want to have bright red pixels with four different levels of transparency, you must use four separate palette entries to accommodate them--all four entries will have identical RGB components, but the alpha values will differ. If you want all of your colors to have four levels of transparency, you've effectively reduced your total number of available colors from 256 to 64. In general, though, only some of the colors need more than one level of transparency, and recognizing which ones do is where things get tricky for the programmer.[2]

> [2] As it happens, the same algorithm that allows one to quantize a 24-bit truecolor image down to an 8-bit palette image also allows one to reduce a 32-bit RGBA image to an 8-bit palette-alpha image. So it's not really that tricky for programmers; it's just not how they're used to thinking about such things.

1.2.2. Gamma and Color Correction

Gamma correction basically refers to the ability to correct for differences in how computers (and especially computer monitors) interpret color values. Web authors in particular are probably aware that Macintosh-generated images tend to look too dark on PCs, and PC-generated images tend to look too light and washed out on Macs. An image that looks good on an SGI workstation won't look right on either a Macintosh or a PC, and even a PC-created image won't look right on all PCs.

Gamma information is a partial solution. It's a means of associating a single number with a computer display system, in an attempt to characterize the tricky physics lurking within a graphics card's digital-to-analog converter (RAMDAC) and within a monitor's high-voltage electron gun and display phosphors. Gamma is

only a first approximation that accounts for overall ``brightness,'' but it is generally sufficient for casual users. More demanding users will additionally want to adjust for differences in the individual red, green, and blue channels--the so-called *chromaticity* values, which are also supported by PNG. Even this is merely a second approximation, however.

The absolute best solution currently available is to use a complete *color management system*, which allows one to take into account things like the viewing environment (a ``dim surround,'' for example) and its interaction with the human visual system. The International Color Consortium has defined a profile format that describes the relationship between an input color space (say, a digital camera or scanner) and the output color space that the user sees. This is the most general way to account for cross-platform differences (and, of course, PNG supports it via the iCCP chunk), but its flexibility comes at a cost: it tends to add at least 250 bytes and often 2,000 bytes or more to every image.

Fortunately, a new proposal for operating systems and physical devices avoids the overhead of a complete ICC profile. Called *sRGB*, for Standard RGB color space, it defines just that: a standard, unified color space that devices can support, thereby allowing true color management with minimal file overhead and no need for the user to wade through a complicated end-to-end calibration procedure. As of January 1999, the sRGB proposal was in ``Committee Draft for Voting,'' and it should be approved as an international standard[3] by mid-1999; conformant devices should start appearing shortly thereafter. PNG supports sRGB via a chunk called, logically enough, sRGB.

> [3] sRGB is Part 2 of IEC 61966 (*Colour Measurement and Management in Multimedia Systems and Equipment*), a proposed standard of Technical Committee 100 of the International Electrotechnical Commission. The IEC is a standards body similar to the International Organization for Standardization (ISO); in fact, international standards such as MPEG, VRML97, and the Latin-1 character set are all joint ISO/IEC standards, and PNG is on track to join them.

Gamma, chromaticity, and color management are described in more detail in Chapter 10, "Gamma Correction and Precision Color"; PNG's basic structure, including the means by which it can be officially or unofficially extended, is covered in Chapter 8, "PNG Basics" and Chapter 11, "PNG Options and Extensions".

1.2.3. Interlacing and Progressive Display

By now, just about everyone has seen interlaced GIFs in action; they first show up with a very stretched, blocky appearance and gradually get filled in until the full-resolution image is displayed. Their big advantage is that an overall impression of the image is visible after only one-eighth of the image data has been transferred; gross features such as embedded buttons or large text are often recognizable (and clickable) even at this stage.

But as useful as GIF's interlacing is, it has one big disadvantage: it is not symmetric. In other words, while GIF's first pass consists of one-eighth of the image data, that factor of eight comes entirely at the expense of vertical resolution. Horizontally, every line is at full resolution as soon as it is displayed, which means that each pixel in the first pass is stretched by a factor of eight. Needless to say, this does make text and other features much harder to recognize than they really need to be.

PNG's approach to interlacing is two-dimensional and involves no stretching at all on more than half of its passes. Even-numbered passes are stretched, but only by a factor of two--similar to the effect after GIF's third pass. Some applications display only the odd-numbered PNG passes, so their pixels always appear square. In addition, PNG's interlacing consists of seven passes, as opposed to GIF's four. This means that the user will see an overall impression of the image after only one- sixty-fourth of the data has arrived, eight times faster than GIF.[4] In the time it takes GIF to display its first pass, PNG displays four passes--and keep in mind that PNG's fourth pass is only one-quarter as stretched as GIF's first pass, with ``pixels'' that are basically 2 × 4 blocks instead of 1 × 8. As a general rule, text embedded in an interlaced PNG image becomes readable roughly twice as fast as in the identical interlaced GIF, as shown in Figure 1-4. The rows show the respective appearance after one-sixty-fourth, one-thirty-second, one-sixteenth, one-eighth, one-fourth, half, and all of the data has arrived. The first column shows GIF interlacing; the others show PNG

interlacing, rendered in various styles: standard blocky rendering, interpolated rendering, and sparse rendering, respectively. Note that the word *Interlacing* has roughly the same readability in the fifth GIF row, the fourth blocky PNG row, and the third interpolated PNG row. In other words, the GIF text takes two to four times as long to become readable.

[4] I am implicitly assuming that one-sixty-fourth of the compressed data (the stuff that can be said to ``arrive'') corresponds to one-sixty-fourth of the *uncompressed* image data (what the user actually sees). This is not quite true for either PNG or GIF, though the difference is likely to be small in most cases--and other factors, such as network buffering, will tend to wash out any differences that do exist. See Chapter 9, "Compression and Filtering" for more details.

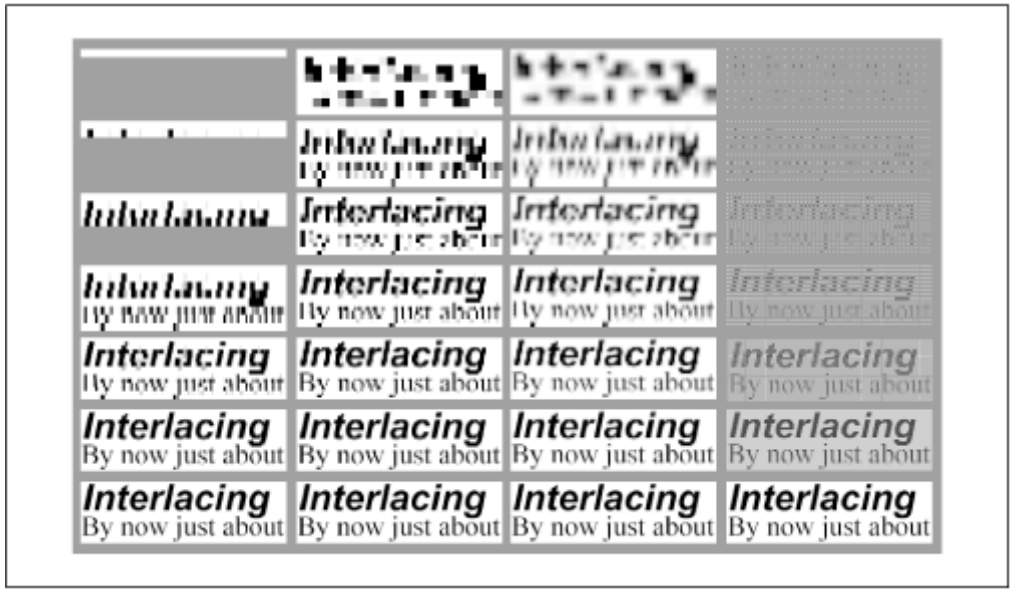

Figure 1-4: *Comparison of GIF interlacing (far left), normal PNG interlacing (second from left), PNG with interpolation (second from right), and PNG with sparse display (far right).* (Click on image for full-scale version.)

JPEG doesn't support interlacing, per se, but it does support a method of progressive display that has been implemented in most browsers since late 1996. In fact, progressive JPEG is a two-dimensional scheme that is not only visually similar to interlaced PNG but also somewhat superior. Loosely speaking, progressive JPEG uses the ``average'' color for any given block of pixels, whereas PNG uses the color of a single pixel in the corner of the block. Early JPEG passes also tend to be somewhat softer (smoother) than early PNG passes; some users find that effect more pleasing.

Finally, I should at least mention TIFF's potential for interlacing. Although no major browser supports TIFF as a native image format, it does offer a very general, random-access approach to image layout. Based either on groups of rows (``strips'') or on rectangular blocks of pixels (``tiles''), a properly constructed TIFF could be used for some form of progressive display. But aside from complete lack of browser support (and very little interest from users), TIFF's compression works only within individual strips or tiles, not across them. So either the interlacing effect would be horrible or the compression would be (or quite possibly both), which is probably why no one seems to have tried it.

1.2.4. Compression

PNG's compression is among the best that can be had *without losing image data* and without paying patent or other licensing fees.[5] Patents are primarily of concern to application developers, not end users, but the decision to throw away some of the information in an image is very much an end-user concern. This information loss generally happens in two ways: in the use of a lesser pixel depth than is required to represent all of the colors in the image, and in the actual compression method (hence ``lossy'' compression).

[5] The ``Burrows-Wheeler block transform coding'' method used in the bzip2 utility is also unpatented and achieves somewhat better compression than PNG's low-level engine, but it

wasn't publicly known at the time and is far, far slower for decoding. JPEG-LS, the new lossless JPEG standard, is fairly fast and performs somewhat better than PNG on natural images, but it does much worse on ``artistic'' ones. It's covered by patents held by Hewlett-Packard and Mitsubishi, but both companies are waiving license fees (i.e., allowing free use). And BitJazz has a new lossless technique called ``condensation''; it appears to compress images 25% to 30% better than PNG, but it is patented and completely proprietary.

PNG supports all three of the main image types discussed earlier: truecolor, grayscale, and palette-based. TIFF likewise supports all three; JPEG only the first two; and GIF only the third, although it can fake grayscale by using a gray palette. Both GIF and PNG palettes are limited to a maximum of 256 colors, which means that full-color images--which usually have tens of thousands or even hundreds of thousands of colors--cannot be stored as GIFs or palette-based PNGs without loss.[6] On the other hand, an image that does fit into a 256-color palette requires only one byte per pixel, which leads to an immediate factor-of-three reduction in file size over a full RGB image before any ``real'' compression is done at all. This fact alone is an important issue for PNG images, since PNG allows an image to be stored either way.

[6] Technically that's not *quite* true in the case of GIF; it supports the concept of multiple subimages, each of which may have its own palette and may be tiled side by side with other subimages to form a truecolor mosaic. This mode is not widely supported, however, particularly on 8-bit displays. Even where it is supported as intended by its proponents, it is an incredibly inefficient way to store and display truecolor image data.

It is worth mentioning that TIFF palettes support up to 65,536 colors, which is sufficient to handle many full-color images without loss. Any palette with more than 256 colors will require two bytes per pixel, eliminating much of the benefit of a palette-based image, but applications that support TIFF are usually more concerned with reading and writing speed than with file sizes.

So let's assume that the image type has been decided; that brings us to the compression method itself. Both GIF and PNG use completely lossless compression engines, and all but the most recently specified forms of TIFF do so as well. Standard JPEG compression is always lossy, however, even at the highest quality settings.[7] Because of this, JPEG images are usually three to ten times smaller than the corresponding PNG or TIFF images. This makes JPEG a very appealing choice for the Web, where small file sizes are important, but JPEG's compression method can introduce visible artifacts such as blockiness, color shifts, and ``ringing'' or ``echos'' near image features with sharp edges. The upshot is that JPEG is a poor choice for intermediate saves during editing, and for web use it is best suited to smoothly varying truecolor images, especially photographic ones, at relatively high quality settings. It is not well suited to simple computer graphics, cartoons, and many types of synthetic images. Figure C-3 in the color insert demonstrates this: notice the dirty (or ``noisy'') appearance of the blue-on-white text, the faint yellow spots above and below it, the darker blue spots in the upper half, and the hints of pink in the white-on-blue text.

[7] There are two forms of truly lossless JPEG, which are discussed briefly in Chapter 8, "PNG Basics", but currently they are almost universally unsupported. There is also a relatively new TIFF variant that uses ordinary (lossy) JPEG compression, but it is likewise supported by very few applications.

Among the popular lossless image-compression engines, PNG's engine is demonstrably the most effective--even leaving aside the issue of prefiltering, which I'll discuss in the next section. TIFF's best classic compression method and GIF's (only) method are both based on an algorithm known as *LZW* (Lempel-Ziv-Welch), which is quite fast and was used in the Unix utility *compress* and in the early PC archiver ARC. PNG's method is called *deflate*, and it is used in the Unix utility *gzip* (which supplanted *compress* in the Unix world) and in PKZIP (which replaced ARC in the early 1990s as the preeminent PC archiver). Unlike LZW, deflate supports different levels of compression versus speed--a dial, if you will. At its lowest setting,[8] deflate is as fast as or faster than LZW and compresses roughly the same; at its highest setting, deflate is considerably slower but achieves noticeably better compression. (Decompression speed is essentially unaffected by the compression level, except insofar as a less compressed image may take more time to read from network or disk.) The deflate algorithm is described in more detail in Chapter 9, "Compression and Filtering".

[8] Actually I'm referring to deflate's second-lowest compression setting (``level 1''); the very lowest setting (``level 0'') is uncompressed. Sadly, the dial only goes to 9, not 11.

1.2.4.1. Compression filters

Compression filters are a way of transforming the image data (without loss of information) so that it will compress better. Each row in the image can have one of five filter types associated with it; choosing which of the five to use for each row is almost more of a black art than a science. Nevertheless, at least one reasonably good algorithm is not only known but is also described in the PNG specification and is implemented in freely available software. Other algorithms are likely to perform even better, but so far this has not been an active area of research.

By way of example--admittedly an extreme case--a 512 × 32,768 image containing all 16,777,216 possible 24-bit colors compressed over 300 times better with filtering than without. The uncompressed image was 48 MB in size; the compressed but unfiltered version was around 36 MB; but the filtered version (using the ``reasonably good algorithm'' referred to earlier) was only 115,989 bytes (0.1 MB). And a version created by trying multiple filtering approaches was a mere 91,569 bytes, for a total compression ratio of 550:1 and an improvement over the unfiltered version of more than 400 times. Keep in mind that we're talking about *completely lossless compression* here. Yow.

Filtering is also described in more detail in Chapter 9, "Compression and Filtering".

1.2.4.2. Compression oopers

Despite PNG's potential for excellent compression, not all implementations take full advantage of the available power. Even those that do can be thwarted by unwise choices on the part of the user.

The most harmful mistake from the perspective of file size and apparent compression level is mixing up PNG image types. Specifically, forcing an application to save an 8-bit (or smaller) palette image as a 24-bit truecolor image is *not* going to result in a small file. This may be unavoidable if the original has been modified to include more than 256 colors (for example, if a continuous gradient background has been added or another image pasted in), but many images intended for the Web have 256 or fewer colors. These should almost always be saved as palette-based images.

Another simple mistake is creating interlaced images unnecessarily. Interlacing is a great benefit to users waiting for large images to download, but on small ones such as buttons and icons, it makes little difference. From a compression perspective, on the other hand, interlacing can have a significant impact, especially for small images. Compression works best where pixels are similar or identical, which is often the case in localized regions, but PNG's two-dimensional interlacing scheme mixes up pixels in an ``unnatural'' order that can destroy any compressor-friendly patterns.

Another ``unnatural'' image modification is standard JPEG compression. The echoes (or ringing) I mentioned earlier are almost never a good thing from PNG's point of view, regardless of their visual effect. For example, a blue image with white text could be saved natively as a two-color (1-bit) palette PNG. After JPEG compression, however, there will be a whole range of blues and whites in the image, and possibly even hints of some other colors. The image would then have to be saved as an 8-bit or even a 24-bit PNG, with obvious consequences for the file size. Bottom line: don't convert JPEGs to PNGs unless there is absolutely no alternative. Instead, start over with the original truecolor or grayscale image and convert *that* to PNG.

On the programmer's side, one common mistake is to include unused palette entries in a PNG image, which again inflates the file size. This error is most noticeable when converting tiny GIF images (bullets, buttons, and so on) to PNG format; these images are typically only 1,000 bytes or so in size, and storing 256 3-byte palette entries where only 50 are needed would result in over 600 bytes of wasted space. PNG's support for transparent palette images, which involves a secondary ``palette'' of transparency values that mirrors the main color palette, can also be misused in this way. Because all palette colors are assumed to be opaque unless explicitly given transparency, well-written programs will reorder the palette so that any transparent entries come first. That allows the remainder of the transparency chunk, containing only opaque entries, to be omitted.

Another common programmer mistake is to use only one type of compression filter, or to vary them incorrectly. As noted earlier, compression filters can make a dramatic difference in the compressibility of the image. However, this is not a feature that users need to know much about. For applications such as Adobe Photoshop that do allow users to play with filters, the best approach is to turn off filters for palette-based images and to use dynamic filters for all other types.

Finally, the low-level compression engine itself can be tweaked to compress either better or faster. Usually ``best compression'' is the preferred setting, but an implementor may choose to use an intermediate level of compression in order to boost the interactive performance for the user. In general, the difference in file size is negligible, but there are rare cases in which such a choice can make a big difference.

A more detailed list of compression tips for both users and programmers is presented in Chapter 9, "Compression and Filtering".

1.2.5. Summary of Usage

Table 1-1 summarizes the sorts of tasks for which PNG, JPEG, GIF, and TIFF tend to be best suited; question marks indicate debatable entries. (Keep in mind that there are always exceptions, though.)

Table 1-1. *Comparison of Typical Usage for Four Image Formats*

	PNG	GIF	JPEG	TIFF
Editing, palette image, fast saves	✓	✓		✓
Editing, truecolor image, fast saves	✓			✓
``Final'' edit, best compression	✓			
Editing, maximal editor portability	?	?		?
Web, truecolor image, no transparency			✓	
Web, palette image, no transparency	✓	✓		
Web, image with ``on/off'' transparency	✓	✓		
Web, image with partial transparency	✓			
Web, cross-platform color consistency	✓			
Web, animation		✓		
Web, maximal browser portability	?	✓	✓	
Web, smallest possible images	✓		✓	

Several things are worth noting here. The first is that TIFF is not at all suited as a web format, simply because it is not supported by any major browser. (This will not be a big surprise to the web designers in the audience.) Even as an editing format, TIFF's main strength is its speed. With regard to portability between image-editing apps, the facts are a little murkier, however. GIF traditionally has been the best-supported format due to its simplicity, but a number of shareware and freeware applications have dropped support due to patent-licensing issues. TIFF has been widely supported, too, but it has also been widely cursed for its incompatibilities among apps. And PNG, of course, is still relatively new. By now it is supported by most of the main image editors, but some of its features (such as 48-bit truecolor) are often supported as read-only capabilities or ignored altogether.

The choice of a web format depends almost entirely on what features are required in the image. Transparency automatically rules out JPEG; partial transparency rules out GIF, as well. For animation, GIF is the only choice. For opaque, photographic images, JPEG is the only reasonable choice--its compression can't be beat. The truly critical issue, however, is portability across browsers. GIF and JPEG are relatively safe bets, but what about PNG? By late 1997, it was supported (at least minimally) in virtually all browsers; Microsoft's Internet Explorer 4.0 and Netscape's Navigator 4.04 finally got native PNG support in October and November 1997, respectively.[9] But gamma correction was supported only by Internet Explorer, and PNG transparency was almost unusable. At the time of this writing, Navigator 5.0 is still unreleased, and IE 5.0 for

Windows is unchanged from version 4.0. But there are strong indications that the Big Two will finally support both gamma and full alpha-channel transparency in their next major releases.

[9] Most other web browsers have supported PNG natively since 1995 or 1996.

Of course, that begs the question of when it is safe to start using PNG on the Web. In theory, the extended **OBJECT** tag in HTML 4.0 provides the means to do so immediately. **OBJECT** is a ``container'' in HTML parlance, similar to **FONT** tags or **BLOCKQUOTE**; it affects the stuff inside it, between the **<OBJECT>** and **</OBJECT>** tags--including other (nested) **OBJECT**s. Unlike most container tags, however, **OBJECT**s refer to their own data (as part of the **<OBJECT>** tag itself), and this can include images. In fact, one can think of an **OBJECT** as an extremely enhanced **IMG** tag. Whereas **IMG** refers to a single datatype (just images) and can display a small amount of plain text if the image can't be rendered (via the **ALT** attribute), **OBJECT**s can refer to numerous datatypes (images, VRML, Shockwave, Java applets, and so on) and can display arbitrary HTML if their main datatype cannot be rendered (via the contents of the **OBJECT** container). Thus, browsers peel **OBJECT** blocks like onions, first trying to render the outermost layer and moving inward until they find something they can handle. As soon as they find something to render, the remainder of the block is discarded. (This is the sense in which the inner stuff is ``affected'': it may be completely ignored. Indeed, only one layer is *not* ignored...at least according to the HTML 4.0 specification.)

So the preferred approach for PNG images is simply to wrap an **OBJECT** tag around an old-style **IMG** tag, where the **OBJECT** refers to the PNG and the **IMG** refers to a JPEG or GIF version of the same image. I'll provide some concrete examples of this in Chapter 2, "Applications: WWW Browsers and Servers", *Applications: WWW Browsers and Servers*. Newer browsers that support both PNG and **OBJECT** will render the PNG in the outer **OBJECT**, ignoring the **IMG** tag. Older browsers will either ignore **OBJECT** as an unknown tag or else parse it but recognize that they cannot render the PNG; either way, they will use the GIF or JPEG from the inner **IMG** tag, or the text in the ALT attribute if they do not support images.

At least, that's the theory. The main problem with this approach is that no version of Navigator or Internet Explorer up through the latest 4.x releases handles **OBJECT** tags correctly. Both browsers will attempt to find a plug-in to handle an **OBJECT** image; lacking that, they will either render the inner **IMG** or fail entirely. I'll look at this in more detail in Chapter 2, "Applications: WWW Browsers and Servers".

But plug-in oddities notwithstanding, the **IMG**-within-an-**OBJECT** approach works moderately well now and will only get better as browsers improve their conformance with WWW standards and as the need for external PNG plug-ins diminishes. Indeed, most of the images on the Portable Network Graphics home site are referenced in this manner. As for referring to PNG images directly in old-style **IMG** tags, which is more commonly thought of as ``using PNG on the Web''--that depends on the images and on the target audience. For example, the Acorn home site already uses PNG images in places; their audience is largely Acorn users, and Acorn Browse has perhaps the best PNG support of any browser in the world. But sites targeted at the average user running Navigator or Internet Explorer must keep in mind that any given release of the Big Two browsers achieves widespread use only after a year or so, and even then, a large percentage of users continue to use older versions. From a PNG perspective, this means that late 1998 was about the earliest it would have been reasonable to begin using **IMG**-tag PNGs on general-purpose sites. Sites that would like to make use of PNG transparency or gamma support will have to wait until about a year after the 5.0 releases occur, which presumably means sometime in the year 2000. (PNG as the Image Format of the New Millennium[10] has a nice ring to it, though.)

[10] That would be the millennium of four-digit years beginning with the numeral ``2,'' which, of course, is what everyone will be celebrating on New Year's Eve, 1999. (The Third Millennium is the one that starts on January 1, 2001.)

1.3. Case Study of a PNG-Supporting Image Editor

Software development tends to be a dynamic and rapidly changing field, and even periodicals have trouble keeping up with what is current. To attempt to do so in a book--even one that uses the phrase ``at the time of this writing'' as often as I have here--borders on the ridiculous. Nevertheless, given PNG's unique feature set and its unfamiliarity to many of those who could make the best use of those features, I feel that it is worth the risk to explore in depth an application that appears to have, as of early 1999, the best PNG support of

anything on the market: Macromedia's Fireworks 1.0, available for 32-bit Windows and Macintosh. (Version 2.0 was released while this book was in the final stages of production; information about it is noted wherever possible, but I did not have time to test it.)

Fireworks is an image editor with a feature set that rivals Adobe Photoshop in many ways, but with far more emphasis on web graphics and less on high-end printing support. In this, it is closer to Adobe ImageReady, a web-specific application intended to tune image colors and optimize file sizes. I'll come back to Photoshop and ImageReady in Chapter 4, "Applications: Image Editors".

1.3.1. PNG Feature Support in Fireworks

Fireworks 1.0 supports a good range of PNG features and image types, and it truly shines in its handling of transparency--indeed, its native internal format is 32-bit RGBA (truecolor with a full 8-bit alpha channel) for all images, and it can save this format, too. In addition, ordinary single-color (GIF-like) transparency is supported in both palette-based and RGB image types, and PNG's unique ``RGBA palette'' mode is also supported. Nor is this support limited to recognizing when an image contains 256 or fewer color-transparency combinations; with a suitable choice of export options, Fireworks can (within limits) quantize and optionally dither even a truecolor image with a nontrivial alpha channel to an 8-bit RGBA-palette image.

There are a couple of notable omissions from Fireworks's list of PNG features, however. The most painful is the lack of support for gamma and color correction; images created by the application will vary in appearance between different display systems just as much as any old-style GIF or JPEG image would, appearing too bright and washed out on Macintosh, SGI, and NeXT systems or too dark on just about everything else. Version 1.0 also cannot write interlaced PNGs, even though it provides a seemingly valid checkbox option for some PNG output types. Version 2.0 addresses this problem, but only in a very limited way: the original plans were to include a ``hidden'' preference that can be changed so that all exported PNG images are interlaced (instead of none of them).[11]

> [11] A tight release schedule was the main reason for the lack of a real fix in version 2.0; Macromedia engineers were fully aware of the deficiencies in the workaround and are expected to address them in the next release.

As one would expect of a graphics application targeted at the Web, Fireworks doesn't preserve 16-bit samples, although it will read 16-bit PNG images (for example, from a medical scan) and convert the samples to 8 bits. Slightly more surprising is its lack of support for true grayscale PNGs; Fireworks saves these as palette-based files, with a palette composed entirely of grayscale entries. This is a perfectly valid type of PNG file, but the required palette adds up to 780 bytes of unnecessary overhead, a distinct liability for icons and other tiny images. On the other hand, a palette-based grayscale image with transparency can include a colored palette entry to be used as the background color, something that PNG does not support for true grayscale files.

In addition to supporting PNG as an output format, Fireworks actually uses PNG as its native file format for day-to-day intermediate saves. This is possible thanks to PNG's extensible ``chunk-based'' design, which allows programs to incorporate application-specific data in a well-defined way. Macromedia has embraced this capability, defining at least four custom chunk types that hold various things pertinent to the editor. Unfortunately, one of them (pRVW) violates the PNG naming rules by claiming to be an officially registered, public chunk type, but this was an oversight and should be fixed in version 2.0.

Although it is entirely possible to use the intermediate Fireworks PNG files in other applications, including on the Web (in fact, one of the ``frequently asked questions'' on the Fireworks web site specifically mentions Netscape, Internet Explorer, and Photoshop), they are not really appropriate for such usage. One reason is that the native PNG format reflects Fireworks's internal storage format, which, as mentioned earlier, is 32-bit RGBA. Even if the image contains only two colors and no transparency, it is saved as a 32-bit PNG file. That certainly doesn't help the old compression ratio any, but the potential for expansion due to the image depth is often overshadowed by that due to the custom chunks, several of which are huge.[12] Thanks to these chunks (which are meaningless to any application but Fireworks), the intermediate PNG files can easily be larger than a completely uncompressed RGBA image would be.

[12] In a 590k tutorial image from Macromedia's web site, 230k is due to image data; 360k is due to custom chunks.

Of course, Macromedia never intended for users to treat the native Fireworks PNG files as the final output format. The fully editable ``fat'' PNGs are produced by the Save menu option; to make final, highly compressed PNGs for web usage, use the Export option. While this might seem like an odd approach to someone unfamiliar with modern image editors, its only real difference from that of applications like Photoshop or Paint Shop Pro is the fact that the intermediate format is widely readable even by low-end apps and browsers (which is not the case for Photoshop's native *.psd* format or Paint Shop Pro's *.psp* format). For an in-house network with high-speed links--for example, in a design studio--this allows images to be easily browsable over the intranet, yet retain all of their object-level editing attributes.

1.3.2. Invoking PNG Features in Fireworks

Because Fireworks's internal format is 32-bit (i.e., truecolor plus a full alpha channel), working with transparency is as easy as opening an image and applying the Eraser tool to its background. For example, suppose you have a photograph of someone and want to focus on the face by making everything else transparent, leaving behind an oval (or at least roundish) portrait shot with a soft border. There are several ways to accomplish this, but the following prescription is one of the simplest:

1. Open the original image (**File → Open**).
2. Pick the background image (**Modify → Background Image**).
3. Double-click on the **Lasso** tool (right side of tool palette, second from top).
4. In the **Tool Options** pop-up, pick **Feather** and a radius, perhaps 25.
5. Draw a loop around the face of the subject.
6. Invert the lasso selection so that the part *outside* the loop gets erased (**Select → Inverse**).
7. Erase everything outside the loop via **Edit → Clear** (or do so manually with the Eraser tool).

Note that the Lasso tool's feathering radius is subtly different from that available via the Select menu. The latter is a smoothing factor for the Lasso's *boundaries*/; in this example, with an inverted selection so that the image's rectangular boundary is also lassoed, changing the value through the menu will round off the corners of the dashed Lasso boundary and may merge separated parts of it together. The feathering radius on the Tool Options pop-up affects only the width of the partially transparent region generated along the Lasso's boundary.

In any case, that's all there is to creating an image with transparency. The next step is to save it as a PNG file. As I just noted, the Save and Save As... menu items save the complete Fireworks ``project,'' retaining information about the objects in the image and the steps used to create them, at a considerable cost in file size. It is generally worthwhile to save a copy that way in case further editing is needed later. But for publishing the image on the Web, it must be exported, and this is where it can be converted into a palette-based image with or without transparency--or left as a 32-bit RGBA image, but without all of the extra editing information included.

First let's consider the case of exporting the image as a full RGBA file. Here are the available options in the Export dialog box:

- **Format:** PNG
- **Bit Depth:** Millions +Alpha (32 bit)

Fireworks 1.0 provides no option to interlace the image, so the preceding steps represent the complete list of possibilities for this case. Things get more interesting when it comes to palette-based (or *indexed-color*) images. Then one has the option of choosing either single-color transparency or the nicer RGBA-palette

transparency, in addition to a number of other palette-related options. Here are the options for the RGBA-palette case:

- **Format:** PNG
- **Bit Depth:** Indexed (8 bit) (this is the default)
- **Palette:** WebSnap Adaptive (default) or Adaptive
- **Dither:** Check on or off
- **Transparency:** Alpha Channel
- **Interlaced:** Checkbox may be checked but does nothing in version 1.0

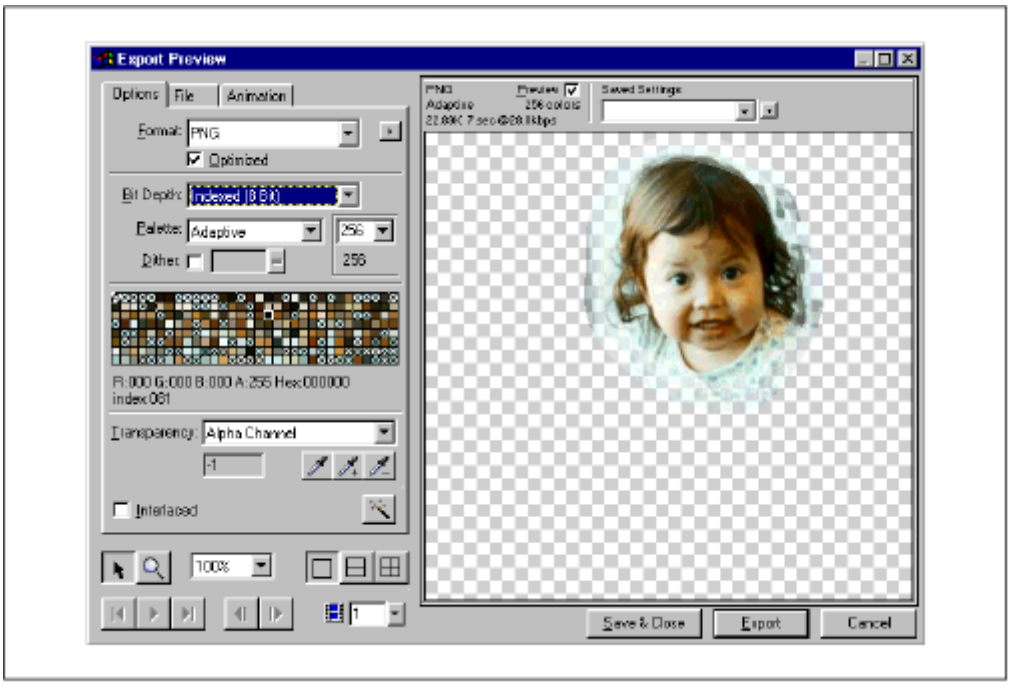

Figure 1-5: *Fireworks Export Preview window showing RGBA-palette options.* (Click on image for full-scale version.)

Note that the effects of the current options are reflected in the preview image to the right (as in Figure 1-5), which shows a limitation in Macromedia's original implementation of RGBA-palette mode. In particular, only four levels of alpha are used, two of which are either complete transparency or complete opacity (the other two represent one-third and two-thirds transparency), which results in very noticeable banding effects in Figure 1-6.

Figure 1-6: *Example of Fireworks RGBA-palette image showing strong banding.*

The four-level approach works quite well for anti-aliasing (that is, preventing ``jaggies" on curved elements such as circles or text), which effectively involves a one-pixel-wide band of variable transparency lying between regions of complete transparency and complete opacity. But the previous example uses a 25-pixel-wide feathering radius, and the two partial-transparency bands both show up extremely well and have sharply defined edges even if dithering is turned on. Unfortunately, that rather defeats the purpose of alpha transparency in this case; the 32-bit version is the only alternative. Fortunately this was one of the areas that got fixed in version 2.0, and judging by one test image, the results are spectacular.

Very nearly the same procedure works if you want to save the image with single-color, GIF-like transparency; instead of picking Alpha Channel from the list of options in the Transparency pull-down box, this time pick **Index Color**. Doing so once will allocate a single palette entry, not used elsewhere in the image, to act as the fully transparent color. A strange feature of version 1.0 is that the Transparency pull-down will still indicate Alpha Channel the first time Index Color is chosen. Choosing it again will cause it to ``stick," but at a cost: the entry chosen for transparency, which generally seems to be the last one (usually black), may now be used in the opaque parts of the image as well as the transparent regions. It is not clear whether this is a bug or an intentional feature of some sort, but it is fully reproducible. Figure 1-7 shows an example.

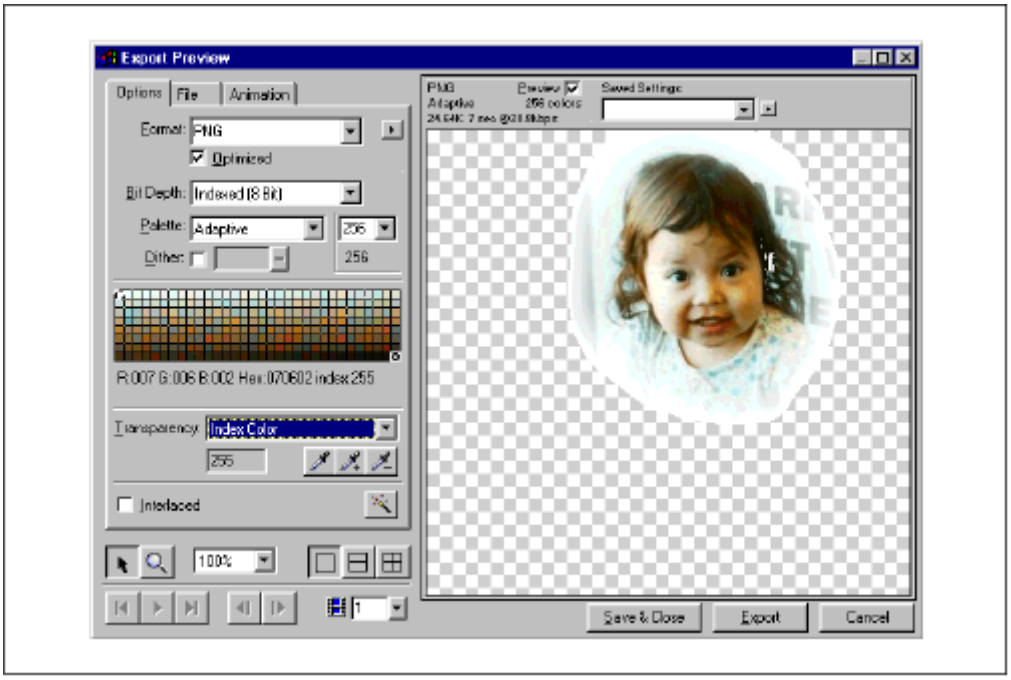

Figure 1-7: *Fireworks Export Preview after choosing Index Color transparency twice, showing transparency (white artifacts) in opaque regions.*
(Click on image for full-scale version.)

As with transparent GIFs, single-color PNG transparency requires that the image be displayed against a suitable background color--white, in our example--to look good. The opposite case, displaying against black, is shown in Figure 1-8.

Figure 1-8: *Example of a Fireworks image with single-color transparency, displayed against the ``wrong'' background.*

1.3.3. Analysis of Fireworks PNG Support

I should note a few caveats about the implementation of indexed-color images and transparency in Fireworks 1.0. For example, the dither checkbox seems to have very little effect in any of the palette examples, and no effect at all on the alpha channel in RGBA images; in fact, the export ``wizard'' explicitly notes this and actually recommends against its use. And the palette-size pull-down seems to have been borrowed from the GIF user interface--it allows only power-of-two palette sizes (e.g., 64, 128, 256) even though PNG's palette chunk can have any number of entries from 1 to 256. The final jump is particularly abrupt; it may happen that 160 colors is the perfect trade-off between quality and image size, but such an image would have to be saved with either 128 or 256 colors.

With regard to transparency, the placement of transparent entries in the Export window's palette view is directly reflected in the PNG file's palette, whether Alpha Channel or Index Color is selected. This is regrettable, since the transparent colors are scattered all over the palette in the alpha case. The single-color case is even worse--the transparent color is the very last entry in the palette. As noted earlier, the preferred approach is to put all of the transparent entries at the beginning of the palette so that the redundant information about opaque colors can be eliminated from the transparency chunk. For a photographic image saved in palette format with single-color transparency, the cost is 127 or 255 bytes of wasted space.

PNG also supports a single-color (or single-shade), ``cheap'' transparency mode that works with truecolor and grayscale images and avoids the need for a full alpha channel, but there is no way to invoke this feature in Fireworks. The lack of any grayscale support other than palette-based means that a gray image with an alpha channel must be saved either as RGBA, doubling its size, or as an indexed image with transparent palette entries, generally with some data loss. (The loss comes about because there are only 256 possible gray+alpha combinations in palette mode, whereas a full gray+alpha image supports up to 65,536 combinations.) There is also no support for a PNG background-color chunk.

Images that already have transparency are preserved quite well (recall that everything is stored internally as 32-bit RGBA), and Fireworks provides quite a number of options beyond what described earlier for adding or modifying transparency. One in particular that could be used for unsharp masking and other special effects is invoked via the **Xtras** menu. With the background image selected, choose **Other → Convert to Alpha**, which first converts the image to grayscale and then to an alpha mask. The lightest parts of the image become the most transparent, while the black parts remain opaque.

Fireworks's compression is reasonably good. Even though there are no user options to adjust the compression level, the default level is a good trade-off between speed and size. Truecolor images tend to be compressed within a few percent of the best possible size, while indexed-color images may see upward of 15% improvement when run through an optimization tool such as *pngcrush* (discussed in Chapter 5, "Applications: Image Converters").

Fireworks also does a good job preserving PNG text annotations, albeit with a quirk: it removes all of the line breaks (``newlines''), for some reason. (Oddly enough, GIF and JPEG comments are not preserved.) The program adds its own Software text chunk; as one might expect, any incoming image that already includes such a chunk will find it replaced. This is a minor breach of PNG etiquette, but one that helps keep tiny image files from getting noticeably bigger because of text comments.

Fireworks 1.0 also adds a Creation Time text chunk to most images it exports. This is not really a problem, per se; what is unusual is that the chunk's contents are invariably ``Thu, May 7, 1998''--a date that has nothing to do with any of the images or even with the release of Fireworks 1.0. See also Chapter 11, "PNG Options and Extensions" for a discussion of why ``creation time'' is a fuzzy concept. Version 2.0 was to have corrected this, replacing the Creation Time text chunk with PNG's officially defined timestamp chunk, tIME, but I did not have a chance to verify that. The tIME chunk indicates the time of last modification, which is a more precisely defined concept and one that is appropriate for an image editor.

As noted earlier, the ability to save interlaced PNG images will first be implemented as a global preference setting. As of January 1999, the plan was for this to require editing version 2.0's preferences file. Under Windows, this file is called *Fireworks Preferences.txt* and is in the Fireworks installation directory (*C:\Program Files\Macromedia\Fireworks*, by default); on the Macintosh, it is called *Fireworks Preferences* and is found in the *System Folder:Preferences* folder. Open the file in any text editor and find the line:

```
(ExportPngWithAdam7Interlacing) (false)
```

Change this to the following to make all exported images interlaced:

```
(ExportPngWithAdam7Interlacing) (true)
```

This change will take effect only after Fireworks 2.0 is restarted. Fortunately, later releases are expected to have a normal checkbox option.

1.3.4. Concluding Thoughts on Fireworks

Lest the preceding detailed list of caveats and oddities leave the reader with the impression that Fireworks's PNG support is not as good as I initially suggested, let me reiterate that it is, in fact, quite good overall. Version 2.0's improved support for RGBA-palette images puts Fireworks far ahead of any other image editor. The inability to set PNG interlacing is regrettable but is being addressed; lack of gamma support is the only truly unfortunate design choice, particularly for a product with both Windows and Macintosh versions. With luck, both gamma and color correction will become core features of the next major release.

◄ PREVIOUS CONTENTS NEXT ►