3 The Science of Digital Media, Chapter 3: Digital Image Processing[1]

I want to reach that state of condensation of sensations that constitutes a picture.

Henri Matisse

Objectives for Chapter 3:

- Know the important file types for digital image data by name, basic format, and application.
- Understand the difference between fixed-length and variable-length encoding schemes.
- Understand the implementation and application of LZW compression, Huffman encoding, and JPEG compression.
- Understand the notation, motivation, and application of luminance/chrominance downsampling.

[1] This material is based on work support by the National Science Foundation. The first draft of this chapter was written under Grant No. DUE-0127280. The final version was written under Grant No. 0340969. This chapter was written by Dr. Jennifer Burg (burg@wfu.edu).

- Understand the motivation, implementation, and application of a variety of indexed color algorithms and the differences among them.
- Understand the implementation and application of a variety of dithering algorithms and the differences among them.
- Understand how pixel point processing is done in digital image processing.
- Understand how convolutions are applied in filtering for enlarging, reducing, or sharpening images.
- Understand the mathematics and application of histograms.
- Understand the mathematics of resampling.

Mathematics Needed for Chapter 3 (in addition to items listed for previous chapters)

- mean, median, mode, and standard deviation

Programming Background Needed for Chapter 3 (in addition to items listed for previous chapters)

- strings

3.1 Tools for Digital Image Processing

In Chapter 2, we examined how image data is captured and represented. We now turn to digital image processing – what you can *do* with your images to refine them for creative or practical purposes.

Chapter 3 begins with a brief overview of the tools you need to work with digital images – cameras, scanners, printers, and application programs. The focus of the chapter, however, is on the mathematics that make these tools work.

Image processing programs give you great power to alter your bitmap images in interesting and creative ways. These tools do a lot of the work for you, and you don't need to know all the underlying mathematics to use them effectively. Then what motivates us to learn the science and mathematics upon which the tools are based? One motivation is that someone needs to create these tools and application programs to begin with, and knowing the science and mathematics of digital media technology makes it possible for you to contribute to its development. A second, more immediate, motivation for looking behind the scenes is that it gives you the ability to work on digital images at a lower level of abstraction. You can create digital image files "from scratch" using programs that you write yourself, or you take raw image data created with the standard tools and alter it with your own programs.

Let's begin now with your work environment.

To create an original digital image, you can use either a digital camera, a scanner, or a paint or image processing program.

Digital cameras come in three basic types – point-and-shoot, prosumer, and professional level. Point-and-shoot digital cameras are what is called "consumer level" – less expensive, each to use, compact, but with few options. These cameras don't give you much control over pixel dimensions, compression, and file type. It may be that you have only one possible image size, in pixel dimensions, and that the images are all saved in

JPEG format. A prosumer camera – which lies between the consumer and professional levels in quality and expense – has more options of pixel dimensions, file type, compression levels, and settings such as white balance, image sharpening, and so forth. Professional level digital cameras are usually ***single-lens reflex cameras*** (***SLR***). In an SLR camera, when you look through the viewfinder, you're seeing exactly what the lens sees, whereas in a prosumer or point-and-shoot camera your view is offset from the lens's view. SLR cameras have high quality, detachable lenses, so that you can change lenses, using different focal lengths for different purposes. A disadvantage of SLR cameras is the weight and size compared to point-and-shoot.

A digital image makes its way to your computer as a file by means of a physical or wireless connection or by means of a memory card that can be inserted into a card holder on the computer or printer. Common physical connections are USB or IEEe1394 (Firewire). Memory cards – e.g., CompactFlash – can be inserted into adaptors that fit in the PCMCIA port of a computer. The number of images you can fit on a memory card depends on the size of the card, and the pixel dimensions and file type of the images. If you want to take a lot of pictures with high pixel dimensions and little or no compression, you need a large memory card. A gigabyte is not too much.

When you have a choice of pixel dimensions in your camera, you should think about consider how much memory you have in your storage medium, how many pictures you want to take at once, and how you want to use your digital images. High pixel dimensions give you a lot of detail to work with, but the pictures take up a lot of memory. The pixel dimensions offered by our camera might also imply different aspect ratios. The ***aspect ratio*** of a digital image is the ratio of the width to the height, which can be written a:b. Pixel dimensions of 640 x 480, for example, give an aspect ratio of 4:3. If you want to print your image as an 8" × 10", you'll have to adjust the aspect ratio as you edit the image.

You can also capture a digital image by means of a scanner. A scanner is like a copy machine that turns the copy into a file of digital data. Like a digital camera, a scanner takes samples at evenly-spaced points. The number of samples it takes in the horizontal and vertical dimension equates to the pixel dimensions of the image. The object being scanned is a physical object with dimensions that can be measured in inches or centimeters. Thus, we can talk of the resolution of a scanner in pixels per inch or centimeters per inch. (Some people use the term *dots per inch* – DPI – with regard to scanners, but *pixels* is a better term, since the scanner is saving the information as pixels in a file.) A scanner has a maximum resolution, limited by the number of sensors it has. You may be allowed to choose a higher resolution than the scanner can physically produce, but in that case the additional pixels are interpolated. High-quality scanners have resolutions of 1200 × 1200 pixels per inch and higher.

A third way to create a digital image is through a paint , draw, or image processing program. A ***paint program***, sometimes called a ***raster graphics editor***, allows you to create bitmap images with software tools that simulate pencils, brushes, paint buckets, typesetters, and more. Paint Shop Pro and Microsoft Paint are two examples. A ***drawing*** program gives you more facilities for vector graphics (e.g., Illustrator and Freehand). Image processing programs – e.g., Photoshop and GIMP – have many of these same tools as paint and draw programs and even more features for editing digital images you may have created through a camera or scanner. The distinction between

paint, drawing, and image processing programs is not a sharp one since the tools overlap. These programs can be very sophisticated and powerful – sometime expensive – but freeware and shareware versions also exist (e.g., Gimp).

3.2 Digital Image File Types

If you take a picture with a digital camera or scan a photograph with a digital scanner, you'll have a choice of file types in which you can save the image. You'll also have a choice of file types when you save an image in an image processing, paint, or drawing program. Not all color modes can be accommodated in all file types, and some file types imply that the image will be compressed while some do not. Some of the image file types, identified by the suffix on the file name, are *.tiff* (also *.tif*), *.jpg* (also *.jpeg*), *.gif*, *.png*, *.bmp*. Our convention in this book is to refer to these file types using all capital letters, such as GIF, since the suffixes can take multiple forms.

Table 3.1 lists some commonly used file formats, categorized into bitmap images, vector graphics, and a hybrid of the two, sometimes called ***metafiles***. Bitmap images are listed first in the table. The four most important things to know about a bitmap filetype are its color model (e.g., RGB, CMYK, or indexed color); its bit depth; its compression type, if any (e.g., LZW, RLE, or JPEG); and the operating systems, browsers, and application software that support it. The common bit depths for bitmap images are 1, 8, 24, 32, or 48 bits. A 1-bit bitmap images uses only black and white. An 8-bit grayscale image allows 256 shades of gray (including pure black and white). Some file types allow 16-bit grayscale. Bitmap files in RGB and CMYK color mode use 24 bits per pixel – one byte for each of three color channels. Bitmap files in indexed color mode use 8 bits (or more) per pixel to store an index into a color table, called a ***palette***.

Because they can be compressed to a small size, GIF files (Graphics Interchange Format) are commonly used for images presented on the web. GIF files allow only 8-bit indexed color. For this reason, they are most suitable for poster-like or cartoon-like images and for photographic images that don't require more than 256 colors. GIF files use lossless LZW compression (discussed below.) GIF files support transparency in that you can choose a color from the color palette and designate it to be transparent. This is commonly done for the background of an image. Animated GIF files can be created by sequences of single images.

Like GIF files, JPEG files (Joint Photographic Experts Group) are also widely used on the web. They are good for continuous tone photographic images, where colors change gradually from one point to the next and many colors are needed for detail and clarity. By JPEG files, we mean files that are compressed with the JPEG compression algorithm (explained later in this chapter). (The file format is actually called JFIF for JPEG File Interchange Format.) You can often select the level of compression you want when you save and compress the file, making a choice between file size and level of detail. GIF files can be saved in an interlaced format that allows progressive download of web images. In ***progressive download***, a low resolution version of an image is downloaded first, and the image gradually comes into focus as the rest of the data is downloaded.

BMP files are a bitmap format that can be uncompressed or compressed with RLE. BMP files are in 1-bit black and white, 8-bit grayscale, 24- or 32-bit RGB color, RGB color, or 8-bit indexed color. (Indexed color will be discussed below.) BMP files

don't support CMYK color. Transparency is supported for individual pixels as in GIF files. Alpha channels are supported in BMP. An ***alpha channel*** is a channel like R, G, and B – using the same number of bits (which accounts for the 32-bit RGB color version). The bits in the alpha channel indicate the level of transparency of the each pixel.

PNG files (Portable Network Graphics) are similar to GIF files in that the format and compression method used lend themselves to poster-like images with a limited number of colors. PNG works well for photographic images also, although it doesn't achieve the compression rates of JPEG. PNG files allow many variations, including 1, 2, 4, 8 or 16 bits per pixel grayscale; 1, 2, 4, or 8 bits per pixel indexed color; and 8 or 16 bits per channel RGB color. PNG files allow the use of alpha channels. With the addition of an alpha channel, the largest bit depth of a PNG file is 64 bits per pixel (4 channels * 16 bits/channel). PNG uses a lossless compression algorithm that works by predicting the color of a pixel based on previous pixels and subtracting the predicted color value from the actual color. PNG files have an optional interlaced format that allows progressive download. PNG does not support animation.

TIFF files (Tagged Image File Format) allow for a variety of color models, including black and white, grayscale, RGB, CMYK, YCbCr, and CIELab. Either 8 or 16 bits per channel can be used for multi-channel color models. A variety of compression methods can be applied – including LZW, RLE, or JPEG – or a TIFF file can be uncompressed. Multiple images can be included in one image file. TIFF files have other options, which can sometimes create problems in its portability because not all applications that read TIFF files are implemented to support all the variations.

GIF, JPEG, PNG, and TIFF usable within a wide variety of operating systems (e.g., Windows, Mac, and Unix/Linux), web browsers, and application programs (e.g., Photoshop and GIMP). A number of proprietary file types are also listed in the table. These are supported by certain application programs – e.g., PSP for Paint Shop Pro and PSD for Photoshop.

Also listed for Photoshop is its *.raw* file format. This format is useful if you want to work with an image file at a low level of abstraction – perhaps writing a program to implement LZW compression, a convolution for unsharp masking, or indexed color – algorithms described later in this chapter. In Photoshop, you can save the image as a *.raw* file such that you have just pixel values (black and white, grayscale, or RGB color, depending on what you want to do). You can then read these values into a program that you've written yourself and experiment with how the algorithms work.

Be sure to notice that there's a difference between Photoshop's *.raw* file format and RAW files that come from your digital camera. The term RAW image file does not refer to a specific file format. Each digital camera can have its own RAW file format that depends on the camera's engineering. In general, a RAW image file contains unprocessed image data exactly as it is detected by the camera's sensors – without any color interpolation, white balancing, or contrast adjustments. For example, consider a one-CCD camera that uses a demosaicing algorithm like the one described in Chapter 2. For such a camera, if you choose to save an image in the RAW format, you have color information for only one of the three color channels from each of the photosites. The camera doesn't do the demosaicing before saving the data. This generally gives you 12 or 14 bits per photosite. This gives you the rawest data possible, allowing you to do your

own adjustments in whatever fine-tuned way you like. However, because the RAW file is proprietary to the camera, you'll need special software to read the file when you port the file to your computer.

File Suffix	Our Abbreviation	File Type	Characteristics
Bitmap Images			
.bmp	BMP	Windows bitmap	1 to 24 bit color depth, 32-bit if alpha channel is used. Can use lossless RLE or no compression. RGB or indexed color.
.gif	GIF	Graphics Interchange Format	Used on the web. Allows 256 RGB colors. Can be used for simple animations. Uses LZW compression. Originally proprietary to CompuServe.
.jpeg or *.jpg*	JPEG	Joint Photographic Experts Group	For continuous tone pictures. Lossy compression. Level of compression can be specified.
.png	PNG	Portable Network Graphics	Designed as an alternative to *.gif* files. Compressed with lossless method. 1 to 64 bit color with transparency channel.
.psd	PSD	Adobe Photoshop	Supports a variety of color models and bit depths. Saves image layers created in photographic editing.
.psp	PSP	Corel Paint Shop Pro	Similar to *.psd*.
.raw		Photoshop	Uncompressed raw file. Could be black and white, grayscale, or RGB color.
.tif or *.tiff*	TIFF	Tagged Image File Format	Often used for traditional print graphics. Can be compressed with lossy or lossless methods, including RLE, JPEG, and LZW. Comes in many varieties.
Vector Graphics			
.ai	AI	Adobe Illustrator	Proprietary vector format.
.swf	SWf	Shockwave Flash	Proprietary vector format; can contain stills, animations, video, and sound.
.cdr	CDR	Corel Draw	Proprietary vector format.
.dxf	DXF	AutoCAD ASCII Drawing Interchange Format	ASCII text stores vector data.
Metafiles			
.cgm	CGM	Computer Graphics Metafile	ANSI, ISO standard.
.emf, .wmf	EMF, WMF	Enhanced metafile and Windows metafile	Windows platform.
.eps	EPS	Encapsulated Postscript	Used for output to Postscript device.
.pdf	PDF	Portable Document Format	An open standard working toward ISO standardization. Windows, MAC, Unix, Linux
.pict	PICT	Picture	Macintosh. Can use RLE or JPEG compression. Grayscale, RGB, CMYK, or indexed color.
.wmf	WMF	Windows metafile	16-bit format. Can be binary or text. Not portable to other platforms.

Table 3.1 Common file types for vector graphics, bitmapped images, and metafiles

We turn now to vector graphic file formats and metafiles. As you recall from Chapter 2, vector graphic files are suitable for images with clear edges and cleanly

separated colors – images that can be described in terms of geometric shapes. A vector graphic file for a poster-type image generally is smaller than a bitmap file for the same image. The size of a vector graphic file is proportional to the number of graphical objects in it, while the size of a bitmap file always depends only on the pixel dimensions, bit depth, color mode, and finally compression. Vector graphic files have the additional advantage of being rescalable without aliasing effects. This is because the image is rendered at the time it is displayed on whatever scale is indicated at that moment and at the maximum resolution of the display device.

Vector graphics files store image data in terms of geometric objects. The objects are specified by parameters like line styles, side lengths, radius, color, gradients, etc. This information can be stored in either binary or text form. If a vector graphics file is text-based, you can look at it in a text editor and read the statements that define the objects. It is possible to create or alter a text-based vector graphic file "by hand" with a text editor if you know the grammar and syntax of the object definition, but working this way requires a lot of attention to detail, and you usually don't need to edit by hand since drawing programs give you such powerful high-level facilities for creating and manipulating vector graphic objects.

A vector graphic file might have statements like those shown in Figure 3.1.

```
LineType 1;
LineWidth 2.0;
LineColr 1;
Line (200,400) (200,600);
Circle (500,600),430;
```

Figure 3.1

Vector graphic files can also be stored in binary form. Typically, these binary files consist of a header identifying the file type and giving global image parameters, a palette (optional), the image data defined in variable-length records, and an end-of-file symbol. Fortunately, editing binary vector graphics files by hand is rarely necessary.

Some file formats combine vector graphics with bitmap images. These are called *metafiles*. The term *metafile* evolved from attempts to create a platform-independent specification for vector graphics. The Computer Graphics Metafile (CGM), originally standardized under the International Standards Organization (ISO) in 1987 and evolving through several revisions, is an example of a standardized metafile format designed for cross-platform interchange of vector graphics, with the optional inclusion of bitmaps. CGM files can be encoded in human-readable ASCII text or compiled into a binary representation. The original CGM was not widely supported by web browsers, but in recent years, the World Wide Web Consortium (W3C) has supported the development of WebCGM, which is designed to incorporate the CGM vector format into Web pages using XML. An alternative to WebCGM for Web vector graphics being developed by W3C is Scalable Vector Graphics (SVG). SVG images can be animated. Generally, WebCGM is considered appropriate for technical graphics and SVG is preferable for graphic arts.

You might want to try opening some vector graphic files in a text editor to see if you can decipher them, or even read them. Because the CGM standard defines both a text and a binary format, your ability to read a CGM file depends on where and how the file was originally made. If you try opening a CGM file in a text editor, you'll probably

discover that it's been encoded in binary, and you won't be able to read it. However, file readers exist for CGM raw files that can give you access to the individual objects.

One of the most widely-used types of metafile is PDF (Portable Document Format). PDF files can be used on all major operating systems – Mac, Windows, Unix, and Linux. PDF documents can contain text, bitmap images, vector graphics, and hyperlinks. The text is searchable.

Microsoft Windows Metafile Format (WMF) is a combined vector/bitmap format. Parameter descriptions of graphical objects in WMF files are stored in 16-bit words. The revised version of WMF, called Enhanced Metafile Format (EMF) uses 32-bit words and has more graphics primitives. WMF and EMF files are stored as binary and are therefore not directly readable.

SWF, proprietary vector graphic format of Flash (formerly produced by Macromedia and bought by Adobe), is currently a very popular file format that is used across a variety of platforms. Its wide use arises from that fact that it allows for inclusion of not only bitmaps but also animated vectors, audio, and video, but within a small, compressed file size. Browser plugins that handle SWF files have become standard. SWF files are stored in binary form and thus are not readable as text.

Among those listed in Table 3.1, the easiest vector graphic files to read as text are the DXF, EPS, and Adobe Illustrator files. Adobe Illustrator files are similar to EPS files, having been designed as a variation of EPS. Both file types can represent either vector objects or bitmaps.

It is possible to compress vector graphic files, and compression is important to file types that include animations, video, and sound. For example, SWF files, which are stored as binary data, use *zlib* compression, a variant of LZW. We will look at LZW and other compression methods in the next section.

3.3 Indexed Color

In image processing programs, it is likely that you will often work in RGB mode and 24-bit color. This corresponds to the color system and bit depth of most current computer displays. However, there are times when you may want to reduce the number of colors used in an image file. You could have a number of motivations for reducing the bit depth – and thus the number of representable colors. It may be the case that your picture doesn't use a large number of colors; slight differences in color may not be important; or you may have constraints on the file size of your picture because of the time it would take to download it or the space it would take to store it. The process of reducing the number of colors in an image file is called ***color quantization***. In image processing programs, the color mode associated with color quantization is called ***indexed color***.

Color quantization begins with an image file stored with a bit depth of n and reduces the bit depth to b. The number of colors representable in the original file is 2^n, and the number of colors representable in the the adjusted file will be 2^b. As an example, let's assume your image is initially in RGB mode with 24-bit color, and you want to reduce it to 8-bit color.

The process of color quantization involves three steps. First, the actual range and number of colors used in your picture must be determined. If your image is stored

initially in RGB mode with 24-bit color, then there are 2^{24} = 16, 777,216 possible colors. The question is, which of these colors appear in the picture?

The second step in color quantization entails choosing 2^b colors to represent those that actually appear in the picture. For our example, the adjusted picture would be limited to 2^8 = 256 colors.

The third step in color quantization is to map the colors in the original picture to the colors chosen for the reduced bit-depth picture. The *b* bits that represent each pixel then become an index into a color table that has 2^b entries, where each entry is *n* bits long. In our example, the table would have 256 entries, where each entry is 24 bits long.

One simple way to achieve a reduction from a bit depth of *n* to a bit depth of *b* is called the ***popularity algorithm***. In the popularity method, the 2^b colors that appear most often in the picture are chosen to be used in the reduced-bit depth picture. A straightforward way to map one of the original colors to the more limited palette is by finding the color which is "closest" using the minimum mean squared distance. More precisely, let the 2^b colors in the reduced color palette be given by their RGB color components such that the i^{th} color has components r_i, g_i, and b_i for $0 \le i < 2^b$. In our example, this means that each of the 256 rows in our color look-up table corresponds to one of the original 16,777,216 colors, decomposed into its three RGB color components. For an arbitrary pixel in the original image with color components *R, G*, and *B*, we want to find the color at index *i* that minimizes $(R - r_i)^2 + (G - g_i)^2 + (B - b_i)^2$.

The disadvantage of the popularity algorithm is that it completely throws out colors that appear infrequently. A picture with one dramatic spot of red in a field of white snow, trees, and sky may lose the red spot entirely, completely changing the desired effect.

The quantization process can also be described graphically, in terms of color spaces. The range of colors in a picture can be seen as a subspace of the RGB cube, and thus the first step in quantization involves finding the smallest "box" that contains all the colors appearing in the image. In the second step, the "box" can be partitioned into 2^bspaces, or in our example, 256 spaces corresponding to the representable colors. A number of methods for achieving this partitioning have been devised.

The ***uniform partitioning algorithm*** divides the subspace containing the existing colors into 2^b blocks of equal size. This can be done by making slices through the initial box in each of the red, green, and blue dimensions. Note that if the quantization is perfectly uniform, the slices in each dimension are equally spaced, but the space between slices in one dimension does not have to equal the space between slices in another. The slices must be made such that they partition the color space into no more than 256 blocks. For example, if we are quantizing to 256 colors, then we could have 16 segments in the red direction, 4 in the green direction, and 4 in the blue direction, yielding $16 \times 4 \times 4 = 256$ blocks; or we could have dimensions of $8 \times 8 \times 4$, or any other combination that gives 256 blocks. We could even have fewer than 256 if we don't mind sacrificing some colors, for example using dimensions of $6 \times 6 \times 6$.

Imagine the color space partitioned uniformly giving eight values of red, eight of green, and four of blue. This is a reduction from 256 values for each red to only eight values. In this partitioning, the red component would map to values between 0 and 7 as follows:

Range of reds in the original image (decimal values)	**Range of red in the original image (binary values)**	**Index to which they map in the color table**
0-31	00000000-00011111	0
32-63	00100000-00111111	1
64-95	01000000-01011111	2
96-127	01100000-01111111	3
128-159	10000000-10011111	4
160-191	10100000-10111111	5
192-223	11000000-11011111	6
224-255	11100000-11111111	7

Essentially, all colors from the original file whose first three bits are 0 map to position 0 in the color table, all whose first three bits are 001 map to position 1, and so forth. Green would map similarly, and blue would be mapped in a coarser-grained fashion to just four values.

The disadvantage of uniform partitioning is that it does not account for the fact that the equal-sized partitions of the color space may not be equally populated. There may be many colors in one partition, and only a few in another. All the colors in a heavily-populated partition will be converted to a single color, and smooth transitions of color will be lost in the image.

A combination of the popularity and the uniform partition algorithms can be achieved as follows: Again assume that n is the bit depth of the original image, and b will be the bit depth of the converted image. Imagine that you want to get a count of how many times each of the 2^n colors appears in a picture. You would probably allocate an array of 2^n integers, read the image file, and increment position i in the array each time the corresponding color is encountered. By this brute force method, if $n = 24$, you could need a $2^{24} = 16{,}777{,}216$ element array. That's pretty large! Instead, you could allocate a 2^k element array, where $b < k < 24$. For example, k could be 12, giving a $2^{12} = 4{,}096$ element array. Then, for each pixel in the image file, consider only the first k/3 bits in each of the color channels. The positions in the color table are incremented each time a color is encountered, but on the basis of the first k/3 bits of each of the color channels only. Then the 2^b most frequently appearing color-ranges from among the 2^k in the color frequency table are chosen for the final color table. As before, the minimum mean squared distance is used to convert each original color to one of the colors in this color table.

Consider how this would work in the case where $n = 24$, $b = 8$, and $k = 12$.

- For each pixel, consider only the first $k/3 = 4$ bits in each of the R, G, and B color channels, for a total of 12 bits total. This is equivalent to taking groups of 4,096 neighboring colors and making them all one "average" color.
- Run through the original image file and count how many of the pixels fall into each of the 4,096 categories.
- Take the $2^b = 256$ most frequently-occurring of these colors and use them in the final indexed color table for your image.

- Finally, to convert your image file based on this color table, take each of the original pixels and map it to the closest of the colors in the color table based on minimum squared distance.

The ***median-cut algorithm*** is superior to uniform partitioning in that it does a better balancing of the number of colors in a partition. The first step, as before, is to reduce the RGB color space to the smallest block containing all the colors in the image. The algorithm proceeds by a stepwise partitioning where at each step some sub-block containing $2^n = c$ colors is divided in half along its longest dimension such that $c/2$ of the existing colors from an image are in one half and $c/2$ are in the other. After this halving, the new sub-blocks are placed on a queue, possibly to be divided again later, until there are finally 2^b sub-blocks. When the color space has been partitioned, a color in the center of each sub-block can be chosen as representative of all colors in that sub-block.

The ***octree algorithm*** is similar to the median-cut algorithm in that it partitions the color space with attention to color population. The algorithm is implemented by means of a tree that has a depth of eight where each node can have up to eight children. This yields a maximum of 8^8 leaf nodes. (Note that $8^8 = 2^{24}$.) However, only 2^b leaf nodes are actually created, each leaf node corresponding to a color in the reduced-bit depth image. We assume for this discussion that $n = 24$ and $b = 8$.

The octree algorithm has two major steps – determining the colors to use in the reduced-bit depth image, and converting the original image to fewer colors on the basis of the chosen color table.

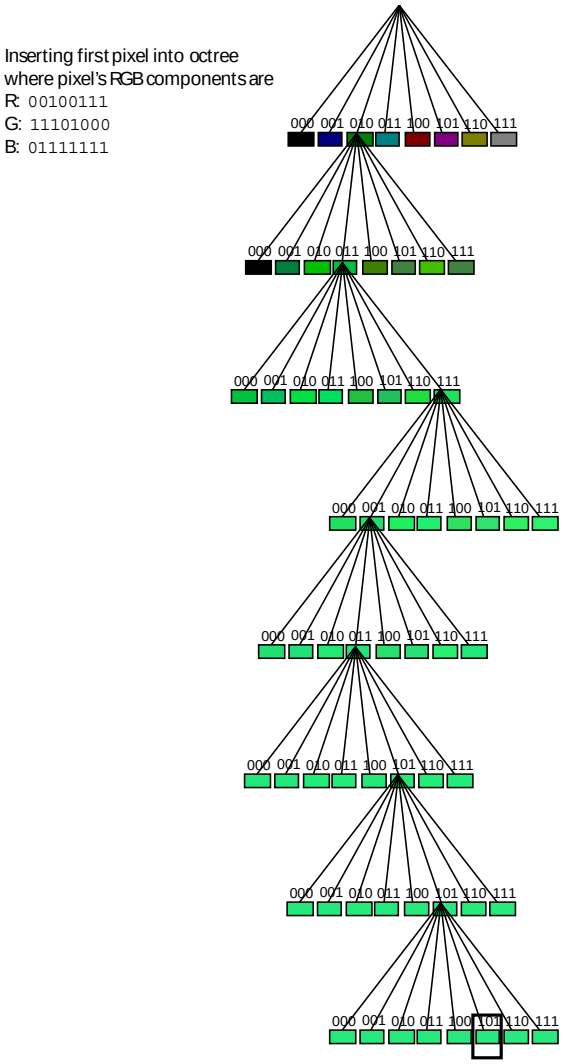


Figure 3.2 Building an octree, one pixel inserted

Let's consider creation of the octree first, which essentially gives us a color table. Initially, a root node is created in the octree with the possibility of having up to eight children. For each pixel in the image, let the R, G, and B components are given in binary as $R = r_1 r_2 r_3 ... r_m$, $B = b_1 b_2 b_3 ... b_m$, and $G = g_1 g_2 g_3 ... g_m$. The algorithm processes each pixel by considering its bits from most significant to least significant, numbering these from 1 to m, matching level numbers of the tree, which is numbered from the root beginning at 0. (Note that $m = n/3$.) For each i, the bits at position i of the pixel being processed through the tree are concatenated as one value, $r_i g_i b_i$. These three bits together can be taken as a value j. Then we create a j^{th} child node of the current node.

For example, say that the pixel currently being inserted into the tree is

$R = 11000110$

$G = 01010101$

$B = 11100011$

Supplements on indexed color and the octree algorithm:

interactive tutorial

programming exercise

worksheet

and $i = 1$. Then $r_i g_i b_i = 101_2$, which has the value 5. Starting at the root node of the tree, we move to the root's 5th child node. If no such child node exists, one is created. It is recorded in this child node that the node has been "visited," i.e., that a pixel has been encountered with this value for the most significant bit of the combined R, G, and B components. The next most significant bits are then considered. They combine to form the decimal value 7, so we move to child 7 of the current node, thus moving down to level 2, and creating the child node if it does not already exist. The child node is marked as having been visited. This continues down the tree to a leaf node, each leaf node corresponding to a 24-bit color. The process repeats for each pixel in the original image file.

We have omitted some details in the description of how a pixel is recorded. Each time a pixel is processed through the tree, there is the possibility that a new leaf node will be created. However, we ultimately want only 2^b leaf nodes representing 2^b colors to be used in the indexed color image. Thus, if processing a pixel results in the creation of a leaf node 2^b+1, some nodes need to be combined. A reducible node – one that has at least two children – must be found at at the lowest possible level. If the reducible node has k children, then after it is reduced there will be room for $k-1$ new leaf nodes in the tree. Reducing a node indicates that, at this point, all the pixels that passed through the reduced node are grouped as a single color. (The tree may grow again later from this node, however.)

As mentioned above, as pixels are processed in each step, the algorithm records at each node in the tree how often the node has been "visited," and by what kind of pixel. Say that variables in each node are as follows:

numVisits /*the number of times the node has been visited*/
RTotal /*the sum of the R components of all the pixels that passed through this node*/
GTotal /*the sum of the G components of all the pixels that passed through this node*/
BTotal /*the sum of the B components of all the pixels that passed through this node*/
isLeaf /*a Boolean indicating if this is a leaf node*/

Then every node in the tree in effect represents a color – the "average" of all the pixels in the image in a certain R, G, and B range, as given by:

$$colorSubstitute = \left(\frac{RTotal}{numVisits}\right), \left(\frac{GTotal}{numVisits}\right), \left(\frac{BTotal}{numVisits}\right)$$

Equation 3.1

After the octree has been created, the original image file is processed a second time. The octree is traversed in a manner reflecting the way it was originally created. For each pixel in a file, going from the most significant to the least significant bits in the R, G, and B components, $0 \geq i \geq m-1$, we move down the octree to the child node corresponding to a grouping of the pixel's i^{th} most significant bits. The leaf node we arrive at by this means contains the color representative for the pixel being processed. This node could represent the pixel's color in the original image, or it could be an average of pixels that have the same bit values in their R, G, and B components up to the i^{th} level of significance.

The octree method has the advantage of using the image's original colors if possible, and averaging similar colors where this is not possible. It is also efficient in its implementation, since the tree never grows beyond a depth of eight.

3.4 Dithering

Supplements on dithering:

interactive tutorial

programming exercise

worksheet

Dithering is a technique for simulating colors that are unavailable in a palette by using available colors that are blended by the eye so that they look like the desired colors. Dithering is helpful when you change an image from RGB mode to indexed color because it makes it possible to reduce the bit depth of the image, and thus the file size, without greatly changing the appearance of the original image.

In an image processing program, if you change an image from RGB mode to indexed mode, you'll probably have a choice of palette sizes and dithering methods. In eight-bit indexed color, if your image is to be placed on the web you may choose to limit the palette to the 216 ***web-safe colors***. All web browsers use the same web-safe palette for eight-bit color, and if you choose this palette you know how your picture will look to others, even those whose browsers or monitors are limited to eight bits. For aesthetic reasons or for reasons having to do with file size, you can also limit your image to a smaller bit depth and palette. Whatever your choice of bit depth, you can choose to dither the image or not. If reducing the bit depth of your image creates undesirable sharply delineated areas of solid color, dithering is a good option.

Dithering algorithms are easy to understand if you consider first how dithering would be done to simulate the look of a grayscale image using only pure black and pure white pixels. Let's look at the three dithering methods that are commonly used in image processing programs – noise, pattern, and error diffusion dithering.

First consider what would happen if you have a grayscale image that uses eight bits per pixel and decide to reduce it to a black and white bitmap that uses one bit per pixel. A sensible algorithm to accomplish this would change pixel values less than 128 to black and values greater than or equal to 128 to white. This is called ***thresholding***.

As you can see from Figure 3.3, thesholding results in large patches of back and white. ***Noise dithering*** (also called ***random dithering***) eliminates the patchiness and high black/white contrast by adding high frequency noise – speckles of black and white that, when combined by the eye, look like shades of gray. The algorithm for a grayscale image proceeds by choosing a random number between 0 and 2^b-1 for each pixel, where b is the bit depth of the indexed image. If the pixel's color value is less than the random number, the pixel is made black; if it's greater, it is white. (Otherwise, choose a new random number.) This algorithm has been applied to the picture in Figure 3.4. It is actually *too* noisy, but the effect can be softened by various means – like not inserting noise with every pixel (using a random number again, to determine when noise will be inserted).

Figure 3.3 Thresholding

Figure 3.4 Noise (i.e., random) dithering

Pattern dithering (also called ***ordered dithering*** or the ***Bayer method***) uses a regular pattern of dots to simulate colors. An $m \times m$ array of values between 1 and m^2 is applied to each $m \times m$ block of pixels in the image file. We call this array a *mask*. The numbers in the mask will determine the appearance of the pattern in the dithered image file. Say that they are as shown below:

1	7	4
5	8	3
6	2	9

For each $m \times m$ block of pixels in the original image, each pixel value p is scaled to a value p' between 0 and m^2. In this case, we can divide all the pixels by 25.6 and drop the remainder. Assuming that the values initially are between 0 and 255, this will result in normalized values between 0 and 9. Then the normalized pixel value is compared to the value in the corresponding position in the mask. If p' is less than that value, the pixel is given the value 0, or black; otherwise, it is white. Pattern dithering is easy and

fast, but it can result in a crosshatched effect (Figure 3.5). There are many variations of this basic algorithm, applying masks of different sizes and types.

Figure 3.5 Pattern dithering

Error diffusion dithering (also called the ***Floyd-Steinberg algorithm***) is a method that disperses the error, or difference between a pixel's original value and the color (or grayscale) value available. In sketch, here's how it works. For each pixel, the difference between the original color and the color available is calculated. Then this error is divided up and distributed to neighboring pixels that have not yet been visited. After all pixels have been visited, the color used for each pixel is the color in the reduced palette closest to the pixel's new value. Details of the algorithm include a determination of the relative fraction of the error distributed to neighboring pixels, the method for choosing the closest available color, and the order in which pixels are processed. Adjustments can be made to reduce color bleeding, i.e., the bleeding of one color into another in the direction in which the error is dispersed.

Consider this example implementation for grayscale images. For each pixel p, the error can be distributed in a manner reflected in the mask below.

	p	7
3	5	1

Imagine laying the mask on top of the image, lining it up over the top left 2×3 pixel area. Each number in the mask pertains to the pixel that it "covers." Call the pixels under the mask $\boldsymbol{f}(r,s)$ where r is the row and s is the column. This gives $p = \boldsymbol{f}(0,1)$. Let e be the error that would be introduced in changing p if the threshold algorithm were applied, defined as follows: If $p < 128$, then $e = p$ (because thresholding would round p down to 0). If $p \geq 128$, then $e = p - 255$ (because thresholding would round p up to 255, which would become a 1 in a black and white image). Now notice that the numbers in the mask add up to 16. The mask symbolizes that if thresholding would make p greater, then, to compensate, error diffusion should subtract 7/16 of the error from the pixel to the right of p; subtract 5/16 of the error from the pixel below p; subtract 3/5 from the pixel below and to the left; and subtract 1/16 from the pixel below and to the right. If

thresholding would make *p* smaller, the corresponding values are added to the neighboring pixels. That is, assignments are made as follows:

$$f(0,2) = f(0,2) + 7/16e$$
$$f(1,0) = f(1,0) + 3/16e$$
$$f(1,1) = f(1,1) + 5/16e$$
$$f(1,2) = f(1,2) + 1/16e$$

(The definition of *e* makes it negative when thresholding would increase *p*.) The equations are assignment statements, denoting that the original pixel values are replaced. Thus, when the mask is moved to the right one pixel, the next step will operate on a pixel that has possibly been changed in a previous step. Pixels can be processed either left to right across each row, or in an alternating motion from left to right and right to left, weaving back and forth. Note that if the error diffusion weaves back and forth, the order of the error multipliers must be flipped accordingly.

After the error has been distributed over the whole image, the pixels are processed a second time. This time for each pixel, if the pixel value is less than 128, it is changed to a 0 in the dithered image. Otherwise it is changed to a 1.

Figure 3.6 Error diffusion dithering

All the dithered images in this section were created "from scratch" through a simple program, one which you could write yourself. An interesting exercise is to try to write these programs and then compare your results to the dithering done by a professional image processing program, one which offers the three dithering options we just examined. (See the programming exercise in the learning supplements, which gives more detail to the algorithms.)

3.5 Channels, Layers, and Masks

Digital image processing tools make it easier for you to edit images by allowing you to break them into parts that can be treated separately. Channels are one such breakdown. A channel is a collection of image data, with separate channels for different color components and opacity information. If you're working in RGB mode, there's a channel for each of the red, green, and blue color components. All channels have the

same pixel dimensions, those of the overall image. You can edit each channel separately, adjusting its contrast, brightness, saturation, shadows and highlights, and so forth. An additional channel called an ***alpha channel*** can be added to an image to store the opacity level for each pixel. Figure 3.7 shows the Channels panel from an image editing program, from which you can select a channel for editing. This image includes an alpha channel, at the bottom.

Figure 3.7 R, G, B, and alpha channel (from Photoshop)

A ***layer*** in image editing is just what the name implies. You can imagine layers to be like sheets of transparent acetate that are laid one on top of another. In areas where there is nothing on a layer, the layer beneath is visible. How can it happen that some areas of a layer have nothing on them? This can happen when you cut something out of one image and paste it into a layer of a second image, as shown in Figure 3.8. The rose was cut out of another image and placed into the image shown. The area around the flower on the layer named *Rose* is transparent, represented by the gray and white checkered pattern. The *Leaves* layer underneath shows through the transparent areas of the *Rose* layer. Notice that each layer has an opacity setting, shown in the Opacity box on the top right of the panel. The *Leaves* layer's opacity is set to 50%, so the solid color background shows through, softening the overall effect and making the rose stand out more.

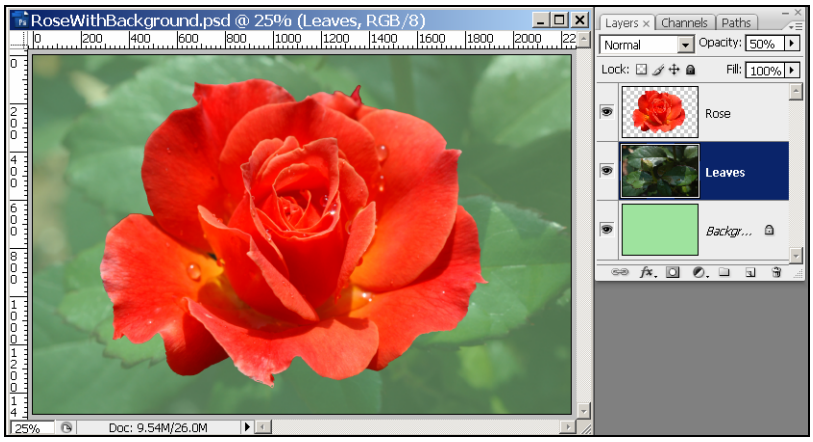


Figure 3.8 Layers (from Photoshop)

If you want to specify the opacity level of a layer pixel-by-pixel, you can do this with either an alpha channel or a layer mask, depending on how your image editing software works. An alpha channel is a type of mask in that it can block the visibility of pixels, either fully or partially. It's different from a physical mask – like one you might put over your face – in that with a digital mask, you can see whatever is on the layer underneath the transparent pixels. The alpha channel has the same pixel dimensions as the overall image. Assuming that the alpha channel is represented by eight bits per pixel, then the closer an alpha channel value is to 255, the more opaque the corresponding pixel in the image. An alpha channel value (alpha value, for short) of 0 corresponds to a fully transparent pixel. (This convention can be reversed in some application programs such that 0 represents fully opaque, so be sure to check.) The alpha values can be normalized to fall between 0 and 1, as shown in Figure 3.9.

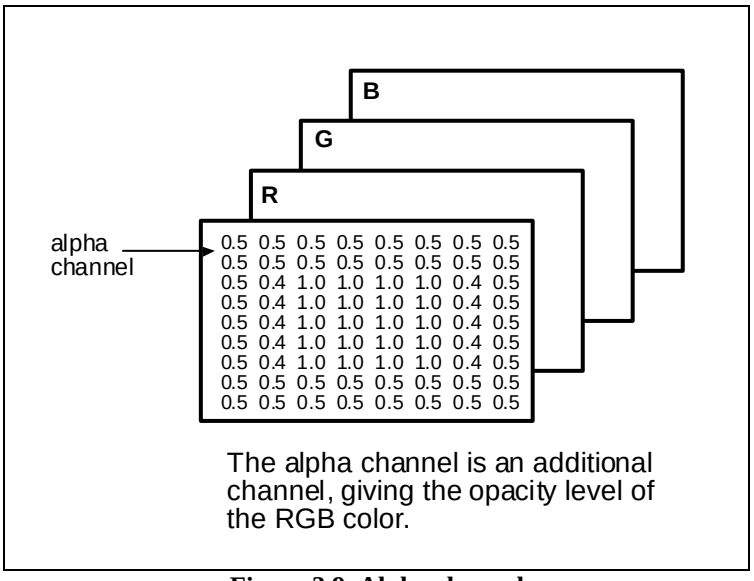


Figure 3.9 Alpha channel

Alpha blending is a mathematical procedure for putting together multiple images or layers with varying levels of opacity. In the simplest case, you have a foreground pixel with a certain opacity level and an opaque background. In that case, alpha blending works as follows:

Given is an RGB image with two layers, foreground and background, the background being 100% opaque. Let a foreground pixel be given by $\boldsymbol{F} = (f_r, f_g, f_b)$ a background pixel be given by $\boldsymbol{B} = (b_r, b_g, b_b)$. Let the alpha value for the foreground, representing the opacity level, be α_f where $0 \le \alpha_f \le 1$. Then for each foreground pixel $\boldsymbol{F}$ and corresponding background pixel $\boldsymbol{B}$ at the same location in the image, the resulting composite pixel color $\boldsymbol{C} = (c_r, c_g, c_b)$ created by alpha blending is defined

key equation

$$c_r = \alpha_f f_r + (1-\alpha_f) b_r$$
$$c_g = \alpha_f f_g + (1-\alpha_f) b_g$$
$$c_b = \alpha_f f_b + (1-\alpha_f) b_b$$

Equation 3.2

Henceforth, we'll condense operations such as this to the form $\boldsymbol{C} = \alpha_f \boldsymbol{F} + (1-\alpha_f)\boldsymbol{B}$, which is meant to imply that the operations are done channel-by-channel.

Let's try an example. If the foreground pixel is (248, 98, 218) with 60% opacity and the background pixel is (130, 152, 74), the result will be (201, 120, 160), as computed below and pictured in Figure 3.10.

0.6*248 + 0.4*130 = 201
0.6*98 + 0.4*152 = 120
0.6*218 +0.4*74 = 160

In the figure, the area where the blocks overlap shows the composite color.

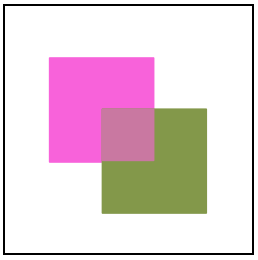


Figure 3.10 Compositing, pink foreground that is 60% opaque with 100% opaque green background

Alpha blending can get more complicated when there are multiple layers, each with its own opacity level. We'll look at this in more detail in Chapter 7, as it applied to the compositing processes used in digital video.

Image processing programs do the mathematics of alpha blending for you, allowing you to create alpha channels or define opacity levels in a number of different ways. One simple method is to add an alpha channel to an image and "paint" on it to indicate which parts of the image you want to be transparent. First you choose a grayscale value between 0 and 255 for your paint color. Then you apply that color to the alpha channel by painting with a brush, creating filled geometric shapes, applying gradients, or using any other methods for applying color to pixels. The closer the color on the alpha channel is to black, the more transparent the corresponding pixels will be in the image.

A second way to make an alpha channel is to select the part of an image that you want to be visible in an image and then convert the selection to an alpha channel. This is the way the alpha channel was created in Figure 3.7. The blue background behind the Demon Deacon puppet was selected with a magic wand tool, which selects similar colors within a given tolerance. The selected was then inverted so that the puppet was selected and the background left out. Then the selection was saved as an alpha channel.

The image shown in Figure 3.7 has an alpha channel that effectively makes the background invisible, but there's no layer underneath to show through. In Chapter 7,

we'll show you how you can import such an image into a video editing program, lay it over a video track, and allow the images on the video track to show through.

If you want to extract an object from its background and put another background behind it, all within the same image, you can do this with a layer mask. A ***layer mask*** is an alpha channel applied to a layer in a multiple-layer image (as opposed to an alpha channel applied to the image as a whole). The concept is still the same – i.e., a channel of pixel data indicates the opacity level of each pixel in a layer. We created another picture of the Deacon walking across campus, this time composing the entire image in an image processing program. This was done by putting the Deacon on the top layer, applying a layer mask to that layer, eliminating the background from that layer, and putting a new background on the bottom layer. In Figure 3.11, you can see a layer mask in the Layers panel and a corresponding alpha channel in the Channels panel.

Figure 3.11 Alpha channel (as a layer mask) on an individual layer

You may want to note that you can have transparent or partially transparent pixels in an image without explicitly having an alpha channel. You've actually seen this already, in the example in Figure 3.8. In this image, part of the rose was selected from its background and the background was erased. Also, the layers themselves have an opacity setting. Thus you see that opacity information can be part of an image file without your having explicit access to the alpha channel. An advantage to explicitly creating an alpha channel is that you don't lose any pixel information by making pixels transparent. The color values are still there, and the alpha channel can easily be edited.

The GIF file type, which are in indexed color mode, can save transparency information without an alpha channel. The transparency information is saved as a pixel color in the color table. You can see this in Figure 3.12. This image, which initially was in RGB mode, was created by first converting the background to a layer, selecting the background, deleting the background, as saving as GIF. You can see in the color table that there is one color reserved to represent transparent pixels. Whether or not this

transparency information is recognized depends on the application program into which you import the GIF file.

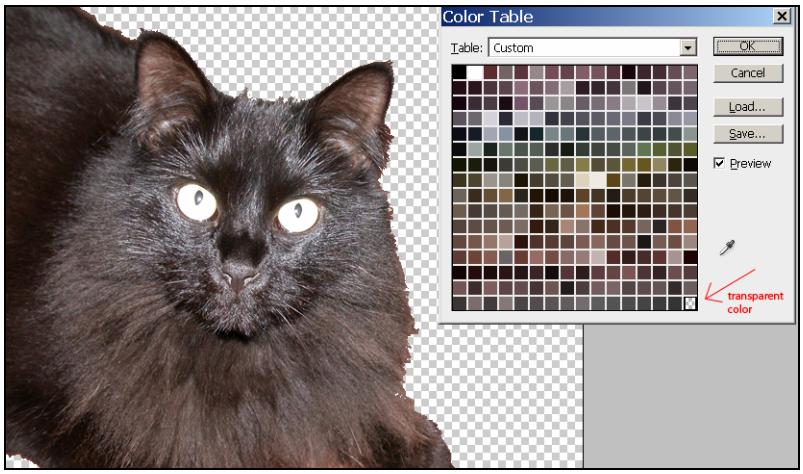


Figure 3.12 GIF file with one color in color table representing transparency

3.6 Blending Modes

In addition to opacity settings, layers have blending modes associated with them. In the figures shown so far in this section, the blending modes have been set to Normal. (Blending modes can be applied to paint tools as well as to layers, indicating how the painted pixels should combine with already-painted pixels.) The Blending Mode setting for a layer is to the left of the Opacity setting on the Layers panel in the figures above.

Blending modes create a variety of effects in making a composite of a foreground and background image. For example, the darken mode replaces a background pixel with the foreground pixel only if the foreground pixel is darker. The lighten mode does an analogous lightening process. The multiply mode, as the name implies, multiplies the foreground and background pixel values, thus combining the darkest elements from both images. The blending operations are done channel-by-channel. Table 3.2 lists the equations for a number of common blending modes. The pixel values in these equations are assumed to be scaled to a range of 0 to 1.

Supplement on blending modes:

programming exercise

Given is an RGB image with two layers, foreground and background, with 100% opacity on each layer. Let a foreground pixel be given by $\boldsymbol{F} = (f_r, f_g, f_b)$ and a background pixel be defined by $\boldsymbol{B} = (b_r, b_g, b_b)$ where $0 \le f_r, f_g, f_b, b_r, b_g, b_b \le 1$. Then for each foreground pixel $\boldsymbol{F}$ and corresponding background pixel $\boldsymbol{B}$ at the same location in the image, the resulting composite pixel color $\boldsymbol{C} = (c_r, c_g, c_b)$ is defined for each blending mode in Table 3.2. The equations are applied channel-by-channel, and results are clipped to a range of 0 to 1.

blending mode	**equation**
normal	$\boldsymbol{C} = \boldsymbol{F}$
multiply	$\boldsymbol{C} = \boldsymbol{F} * \boldsymbol{B}$
divide	$\boldsymbol{C} = \left(\frac{\boldsymbol{B}}{\boldsymbol{F} + \frac{1}{255}} \right) \left(\frac{256}{255} \right)$
screen	$\boldsymbol{C} = 1 - ((1 - \boldsymbol{F})(1 - \boldsymbol{B}))$
overlay	$\boldsymbol{C} = \boldsymbol{B}(\boldsymbol{B} + 2\boldsymbol{F}(1 - \boldsymbol{B}))$
dodge	$\boldsymbol{C} = \left(\frac{\boldsymbol{B}}{\frac{256}{255} - \boldsymbol{F}} \right) \left(\frac{256}{255} \right)$
burn	$\boldsymbol{C} = 1 - \left(\left(\frac{1 - \boldsymbol{B}}{\boldsymbol{F} + \frac{1}{255}} \right) \left(\frac{256}{255} \right) \right)$
hard light	If $\boldsymbol{F} > 0.5$ then $\boldsymbol{C} = 1 - 2(1 - \boldsymbol{B})(1 - \boldsymbol{F})$ If $\boldsymbol{F} \le 0.5$ then $\boldsymbol{C} = 2(\boldsymbol{F} * \boldsymbol{B})$
soft light	$\boldsymbol{C} = 2(\boldsymbol{F} * \boldsymbol{B}) + \boldsymbol{B}^2 - 2(\boldsymbol{F} * \boldsymbol{B}^2)$
grain extract	$\boldsymbol{C} = \boldsymbol{B} - \boldsymbol{F} + 0.5$
grain merge	$\boldsymbol{C} = \boldsymbol{B} + \boldsymbol{F} - 0.5$
difference	$\boldsymbol{C} = \lvert \boldsymbol{B} - \boldsymbol{F} \rvert$
addition	$\boldsymbol{C} = \boldsymbol{B} + \boldsymbol{F}$
subtraction	$\boldsymbol{C} = \boldsymbol{B} - \boldsymbol{F}$
darken only	$\boldsymbol{C} = \min(\boldsymbol{B}, \boldsymbol{F})$
lighten only	$\boldsymbol{C} = \max(\boldsymbol{B}, \boldsymbol{F})$

Table 3.2 Blending modes

The equations in the table were adapted from a GIMP website. Revised versions of GIMP and other image processing programs may use different equations.

Dissolve, hue, saturation, value are sometimes listed as blending modes as well, although they are not implemented by single channel equations. Dissolve mode operates by randomly dithering the alpha channel of the foreground to black and white. Hue takes the hue of the foreground (where the hue is defined) and the saturation and value of the background. Saturation and value modes operate analogously. Color works as a combination of hue and saturation blending modes.

The blending mode equations given in Table 3.2 do not take into account the alpha channels of the foreground and background layers. How blending and alpha channels are combined is implementation-dependent in different application programs.

Generally, the foreground's alpha affects only the strength of the blend, while the background's alpha sets the opacity level.

3.7 Pixel Point Processing

3.7.1 Histograms

When you work with bitmap images, you generally have something you want to communicate or an effect your want to create. It may be that you simply want your image to be as clear and detailed as possible. If you're working on the image as art, you might want to provoke a certain mood or alter colors and shading that affect the aesthetics of the image. If you're using the image to sell something, you may want to change the focus or emphasis. Whatever your goals, you will find yourself doing operations like adjusting the contrast or brightness, sharpening lines, modifying colors, or smoothing edges. Each of these operations is an example of an image transform – a process of changing the color or grayscale values of image pixels. The following discussion divides image transforms into two types: ***pixel point processing*** and ***spatial filtering***. In pixel point processing, a pixel value is changed based only on its original value, without reference to surrounding pixels. Spatial filtering, on the other hand, changes a pixel's value based on the values of neighboring pixels. (You've already seen an example of spatial filtering – dithering.) This section looks at pixel point processing, and in the next section we'll move on to more examples of spatial filtering.

Supplements on histograms:

interactive tutorial

worksheet

One of the most useful tools for pixel point processing is the histogram. A ***histogram*** is a discrete function that describes frequency distribution; that is, it maps a range of discrete values to the number of instances of each value in a group of numbers. More precisely,

key equation

> Let v_i be the number of instances of value i in the set of numbers. Let min be the minimum allowable value of i max be the maximum. Then the ***histogram function*** is defined as
>
> $$h(i) = v_i \text{ for } \min \le i \le \max .$$

A simple histogram is shown below for a group of 32 students identified by what class they are in (1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior). You can see that a histogram is graphed as vertical lines (or sometimes vertical bars are used). There are ten freshmen, six sophomores, five juniors, and 11 seniors.

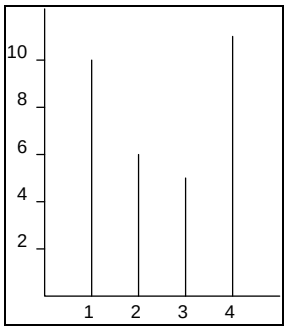


Figure 3.13 A histogram

An image histogram maps pixel values to the number of instances of each value in the image. A grayscale image and its corresponding histogram are shown in Figure 3.14. Grayscale levels extend from 0 to 255 across the horizontal axis. Each of these values has a vertical line above it indicating the number of pixels in the image that have this value. In practice, image histograms are usually normalized by dividing by the maximum number of instances of a pixel value within the range of values in the image. This is illustrated in the figure below. The value that occurs most frequently in the image is the statistical ***mode***. In this example, there are two modes – 141 and 143 occurring with the same frequency. The tallest vertical line is at this point, and it goes to the top of the histogram. Normalizing histograms in this way makes it easier to compare the histograms of two images that don't have the same total number of pixels, since all normalized histograms have a height of one unit.

One of the most important things to observe from a histogram is the extent to which the pixel values cover the possible range. Consider the example in Figure 3.14. The darkest pixel is 58, and the lightest is 210. In other words, the image does not have a wide dynamic range. Consequently, it doesn't have a lot of contrast. All the values are crowded together in the center of the histogram. This generally isn't what you want in an image. Contrast makes the image look sharper and more interesting.

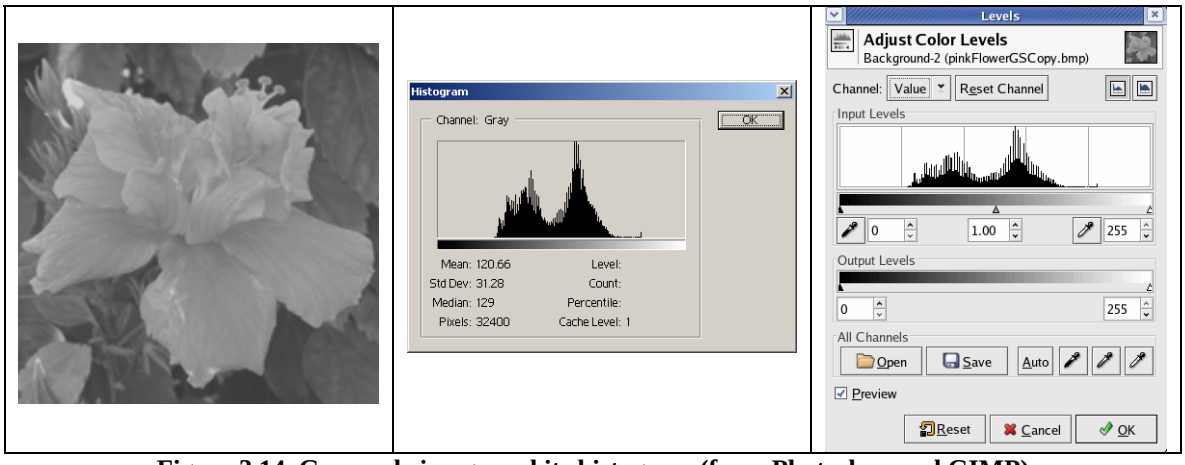


Figure 3.14 Grayscale image and its histogram (from Photoshop and GIMP)

Some histograms, like the one in Figure 3.14, give statistical information about the distribution of pixel values, including the mean, median, and the standard deviation.

key equation

Assume that you have *n* samples in a data set. For our purposes, a sample is a pixel value.

The ***mean***, or average, $\bar{x}$ is defined as:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Equation 3.3

A ***median*** is a value *x* such that at most ½ of the values in the sample population are less than *x* and at most ½ are greater. If the number of sample values is even, the median may not be unique. For example, with [1,3,5,7], any value between 3 and 5 inclusive is a median. Commonly, however, the mean of the two values in the middle is considered the median.

key equation

The ***standard deviation*** measures how much the samples vary from the average. It is defined as

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}$$

where x_i is the i^{th} sample in a population of *n* samples.

Equation 3.4

A large standard deviation implies that most of the pixel values are relatively far from the average, so the values are pretty well spread out over the possible range. Generally, this means that a higher standard deviation is indicative of more contrast in the image. However, you have to be careful how you interpret this. If you had only the standard deviation of an image file's histogram and didn't see the image, you might be fooled. You have to keep in mind that the histogram gives you an overview of the pixel values that exist in an image, but it doesn't tell you anything about *where* they are. Think about this: What type of grayscale image would give you the largest standard deviation? One that has an equal number of purely black and white pixels in it, right? The only pixel values in such an image would be at both extremes – 0 and 255. But both of the images in Figure 3.15 fit this description. The one on the left is perceived as having more contrast because the black and white pixels are spatially separated. The image on the left has alternating black and white pixels, so it looks like a shade of gray.

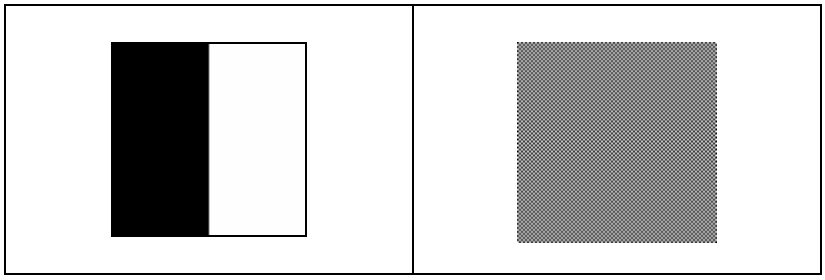


Figure 3.15 Two figures with same the histogram and standard deviation

Histograms are available in scanner software, digital cameras, and image processing programs. You should check your hardware and software to see what type of histograms they offer and how they are used. In scanner software or digital cameras, a histogram can help you adjust lighting or make settings when you capture an image so

that you'll get the maximum amount of appropriate pixel data. With a scanner, you can prescan a digital image and then view the histogram to see what adjustments are necessary before the final scan. With a digital camera, you can take a picture and then view a histogram to decide if you need to take the picture again with different settings or lighting conditions. If most of the pixel values are grouped in the center of the histogram, you may want to adjust your lighting to increase contrast. If most of the values to the right of center, the image is probably too light. If most are to the left, it's probably too dark.

Sometimes a histogram is displayed along with the image just photographed or pre-scanned, and the areas that are ***clipped*** in the image are outlined for you in the image. Clipped values are those that fall outside the sensitivity range of the camera or scanner. If an area is too dark, all the pixels there register as black (0), and if the area is too light they become white (255 in grayscale). Some scanners allow you to increase the ***analog gain*** – the luminance of the scanner's light source – so that enough light can pass through the source image's darker tones, allowing them to be measured.

Changing the lighting and adjusting analog gain are examples of ways to use a histogram to capture good image data from the outset. Gathering the best data possible from the outset is always the best plan. But even with the best planning, you'll still have many situations where you want to alter the captured image in some way. Histograms are very useful in image processing programs for situations like these.

Before showing you how to adjust contrast or brightness with a histogram, we need to clarify how different color modes are handled. In RGB mode, each pixel has three color channels where the values in each channel range from 0 to 255. (Larger bit depths are possible, but eight bits per channel is common.) A separate histogram can be created for each color channel. The R histogram, for example, shows how many pixels have 0 for their R component, how many have 1 for their R component, and so forth up to 255. A composite RGB histogram can also be constructed, which for $0 \le j \le 255$ shows the total number of pixels that have j as their R component plus the pixels that have j as their G component plus the pixels that have j as their B component.

A disadvantage of the composite RGB histogram is that it doesn't correspond directly to the perceived brightness of an image. Among the three color channels, the human eye is most sensitive to green and least sensitive to blue. This fact is reflected in the way that RGB color is transformed to grayscale. Recall from Chapter 2 that a three-byte RGB color can be converted to a one-byte grayscale pixel with value L using

$$L = 0.30R + 0.59G + 0.11B$$

Equation 3.5

Some scanners or image processing programs give you access to a luminance histogram (also called a *luminosity histogram*) corresponding to a color image. A luminance histogram converts the RGB values into luminance, as shown above, and then creates the histogram from these values.

Figure 3.16 shows a histogram that can be manipulated to alter the brightness or contrast of a grayscale image. Notice the triangles on either side of the histogram, pointed to with the red arrows. These triangles can be slid left and right and correspond to values labeled Input Levels. Say that you moved the slider on the left to 55 and the one on the right to 186. Assuming that you don't move the middle slider, this would map the input pixel values to output values such that the pixel that was originally 55 would

become 0, the pixel that was previously 186 would become 255, and everything in between would be "spread out" proportionately. This would increase the contrast and allow for more differences between the grayscale values in the image. If you applied this transform and then looked at the histogram again, the new histogram would look like Figure 3.17. (Actually, you can open the Histogram window and watch the histogram change there as you move the sliders in the Levels window.) Notice that now the whole range of possible grayscale values is being used.

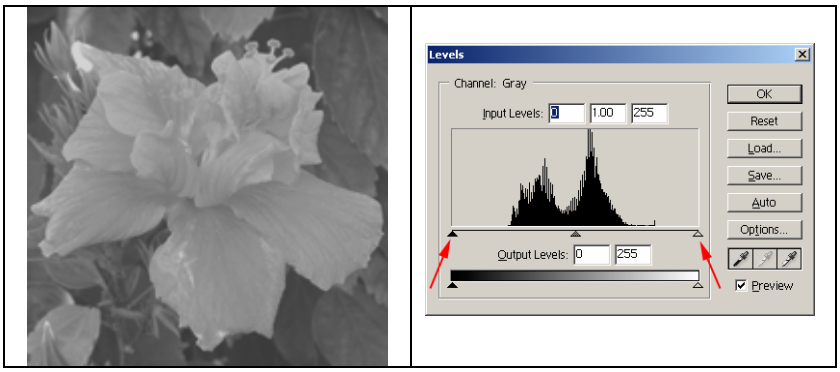


Figure 3.16 A histogram that can be manipulated to adjust brightness and contrast (from Photoshop)

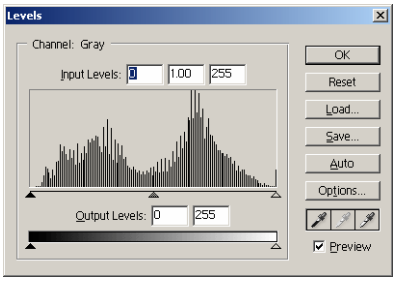


Figure 3.17 Histogram of image in Figure 3.16 after contrast adjustment (from Photoshop)

You might be wondering what the middle value is in the Input Levels input fields. It is called the ***gamma level*** and relates to the midtones of the image. You can manipulate this value by moving the center triangle-shaped slider on the histogram. If you move the middle slider and leave the left and right sliders where they are, you can manipulate the midtones without affecting the lightest and darkest values – the highlights and shadows – in the image. If you do this, you'll see that the gamma value ranges from 0.10 to 9.99. In the next section, we'll look at the mathematics to see why this range of gamma values makes sense.

3.7.2 Transform Functions and "Curves"

Supplements on curves:

[interactive tutorial]

Image processing programs sometimes give you another mathematical/graphical view of image data from which you can perform pixel point processing. Programs like Photoshop and GIMP, the Curves feature allows you to think of the changes you make to pixel values as a transform function. Let's look at pixel point processing from this point of view now.

We define a ***transform*** as a function that changes pixel values. Expressed as a function in one variable, it is defined as

$$\boldsymbol{g}(\boldsymbol{p}) = T(\boldsymbol{f}(\boldsymbol{p}))$$

Equation 3.6

p is a pixel position with coordinates (x,y), ***f(p)*** is the pixel value at that point in the original image, T is the transformation function, and ***g(p)*** is the transformed pixel value. For simplicity, let's abbreviate ***f(p)*** as p_1 and ***g(P)*** as p_2. This gives us

worksheet

$$p_2 = T(p_1)$$

Equation 3.7

Transforms can be applied to pixel values in a variety of color modes. If the pixels are grayscale, then ***f(p)*** and ***g(p)*** are values between 0 and 255. If the color mode is RGB or some other three-component model, ***f(p)*** and ***g(p)*** imply three components, and the transform may be applied either to each of the components separately or to the three as a composite. For simplicity, let's look at transforms as they apply to grayscale images first.

An easy way to understand a transform function is to look at its graph. A graph of transform function T for a grayscale image has p_1 on the horizontal axis and p_2 on the vertical axis, with the values on both axes going from 0 to 255 – that is, from black to white.

Consider the graphs in Figure 3.18. In each case, what would the corresponding transform do to a digital image?

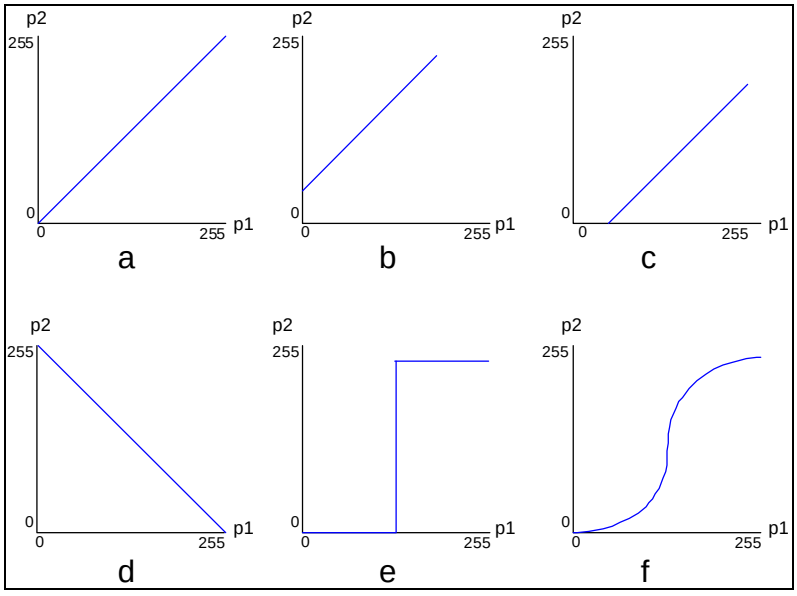


Figure 3.18 Curves for adjusting contrast

a. This transform doesn't change the pixel values. The output equals the input.
b. This transform lightens all the pixels in the image by a constant amount.
c. This transform darkens all the pixels in the image by a constant amount.
d. This transforms inverts the image, reversing dark pixels for light ones.
e. This transform is a threshold function, which makes all the pixels either black or white. A pixel with a value below 128 becomes black, and all the rest are white.
f. This transform increases contrast. Darks become darker and lights become lighter.

Image processing programs have features to automatically adjust the contrast or color levels for you. Many also give you greater control by allowing you to manipulate a graph of the transform function. Figure 3.19 shows the Curves dialog box from Photoshop and GIMP, where you can select points and drag them to adjust contrast or brightness. This makes it possible to create functions like the ones shown in Figure 3.18. In Figure 3.20, we've applied the functions shown in Figure 3.18 so you can see for yourself what effect they have.

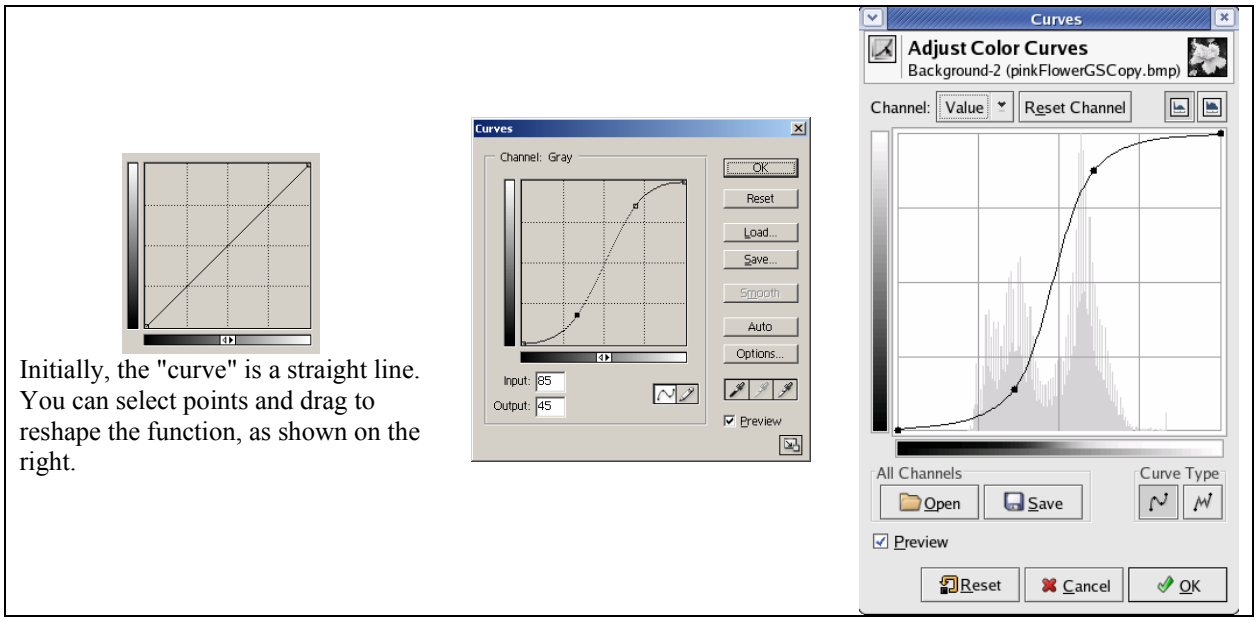


Figure 3.19 Adjusting the curve of the contrast function (from Photshop and GIMP)

a. original image | b. lighten | c. darken

d. invert e. threshold f. increase contrast

Figure 3.20 Adjusting contrast and brightness with curves function

The adjustment we made in contrast using the histogram could be done with a graphical view of the transform function as well. Say that you want to adjust the image in Figure 3.16 in a way that would be equivalent to moving the histogram's left slider to 55 and its right slider to 186, as described above.

Moving the histogram slider on the left from 0 to 55 and on the right from 255 to 186 is equivalent to the transform function pictured in Figure 3.21. If you start with two identical images, perform the histogram adjustment on one, perform the curves adjustment on the other, and then look at the new histogram of each, you'll see that you end up with identical histograms and identical images. Both histograms will look like the one in Figure 3.17.

Notice that the transform function we applied to the midtones is linear. When you move the Input Level sliders on the left and right of the histogram but don't move the middle slider, the grayscale values in between are adjusted by a linear function. The histogram's middle slider corresponds to midtones. When the histogram is adjusted, grayscale values are spread out linearly on either side of this value.

A non-linear function like the S-curve in Figure 3.19 usually does a smoother job of adjusting the contrast. You can create a smooth S-curve like this by clicking on points in the graph and dragging to a new location. Equivalently, you can change the gamma value in the histogram. We promised earlier to look at the math of the gamma value, and we can do it now with reference to the graph of the transform function.

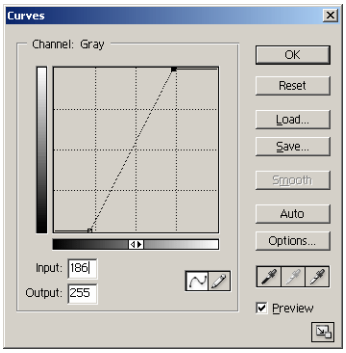


Figure 3.21 Curve to accomplish the same contrast adjustment as done with the histogram (from Photoshop)

key equation

The ***gamma value γ*** is an exponent that defines a non-linear relationship between the input level p_1 and the output level p_2 for a pixel transform function. In particular,

$$p_1 = p_2^{\gamma} \text{ where } p_1 \in [0\ 1]$$

Equation 3.8

This equation may seem a little strange because the input level p_1 is on the left-hand side of the equation; it seems more natural to think of p_2 as a function of p_1. We could rewrite the equation to express p_2 as a root of p_1, but it amounts to the same thing. We give the equation in the form above because this is the way you'll see it in most sources. What's important is the relationship between the values.

Think about what the graph of $p_1 = p_2^{\gamma}$ would look like for various values of gamma, with p_1 defining the horizontal-axis and p_2 defining the vertical. The graph for $x = y^{0.5}$ is given in Figure 3.22. The histogram view allows you to use gamma values between 0.1 and 9.99. A gamma value of 0.1 causes nearly all the pixel values to be changed to black. A value of 9.99 causes nearly all pixels to be changed to white. In general, gamma values less than 1 darken the image while values greater than 1 lighten it.

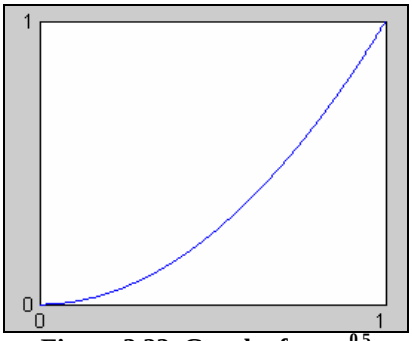


Figure 3.22 Graph of $x = y^{0.5}$

Changing the gamma value in the histogram view of Figure 3.16 is equivalent to pulling on a point in the graph of the transform function, represented in the Curves view of Figure 3.19. As you pull a chosen point, the graph is "stretched" as smoothly as possible to accommodate the change, as if you're pulling on an elastic band. The values of lightest and darkest pixels are anchored where they are, but the midtones are lightened or darkened.

Adjustments such as this can be done on color images as well, but the operations are more complicated. If you're working in RGB color, you can adjust the curve function for the full RGB color values, or you can adjust the curve for each color channel separately. One thing that makes it more difficult to work with color than with grayscale, however, is that RGB is not a luminance/chrominance model. For the three color components, equal changes of values do not create equal changes in luminosity or perceived brightness or contrast of the image. The green component contributes the most to luminosity, red the second most, and blue the least. Thus, linear changes to the RGB color levels do not produce linear changes in brightness. Also, the three color components combine in a non-linear way to create the colors we perceive. In any case, it is possible to work with the color channel curves and visually adjust the colors by moving the graphs. The important thing to understand when you are adjusting a single

color channel is that you are changing only that channel's contribution to each pixel in the image.

3.8 Spatial Filtering

3.8.1 Convolutions

A ***filter*** is an operation performed on digital image data to sharpen, smooth, or enhance some feature, or in some other way modify the image. A distinction can be made between filtering in the spatial versus the frequency domain. ***Filtering in the frequency domain*** is performed on image data that is represented in terms of its frequency components. ***Filtering in the spatial domain*** is performed on image data in the form of the pixel's color values. We'll look at the latter type of filtering in this section.

Spatial filtering is done by a mathematical operation called ***convolution***, where each output pixel is computed as a weighted sum of neighboring input pixels. Consider a grayscale image as a matrix of grayscale values. (Generalizing to three color channels is straightforward.) Convolution is based on a matrix of coefficients called a ***convolution mask***. The mask is also sometimes called a ***filter***. (We will use these terms interchangeably.) The mask dimensions are less than or equal to the dimensions of the image. Let's call the convolution mask $\boldsymbol{c}$ and the image $\boldsymbol{f}$. Assume the image is of dimensions $M \times N$ and the mask is $m \times n$. You can picture the mask being placed over a block of pixels in an image, starting in the upper left corner. (You may notice that the filter numbering is "flipped" with regard to the image. This is by convention.) The pixel that is to receive a new grayscale value lies at the center of the mask, which implies that the dimensions of the mask must be odd. Assume that pixel $\boldsymbol{f}(x,y)$ is the one to receive the new value.

Supplements on convolution:

interactive tutorial

worksheet

key equation

Let $\boldsymbol{f}(x,y)$ be an $M \times N$ image and $\boldsymbol{c}(v,w)$ be an $m \times n$ mask. Then the equation for a linear convolution is

$$\boldsymbol{f}(x,y) = \sum_{v=-i}^{i} \sum_{w=-j}^{j} \boldsymbol{c}(v,w)\boldsymbol{f}(x-v, y-w)$$

where $i = (m-1)/2$ and $j = (n-1)/2$. Assume m and n are odd. This equation is applied to each pixel $\boldsymbol{f}(x,y)$ of an image, for $0 \le x \le M-1$ and $0 \le y \le N-1$.

Equation 3.9

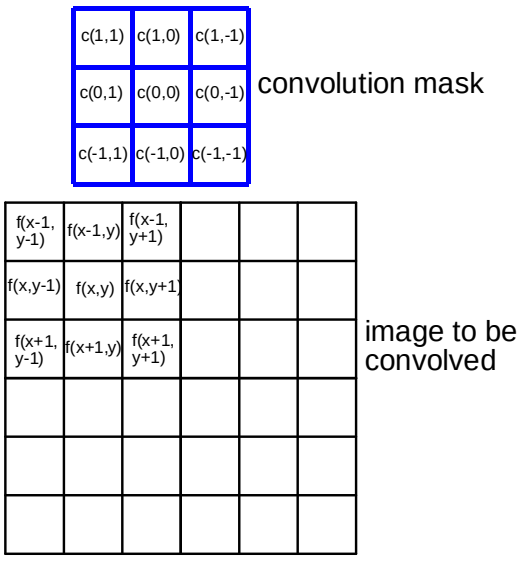


Figure 3.23 Convolution

We have written the convolution equation such that it replaces values in the original image. If you do this, then the new values will be used in the convolution of neighboring pixels. Sometimes this is what you want, and sometimes it isn't, depending on the purpose of the convolution. You should note that sometimes you want to create an entirely new image, and write the new values there so that they aren't used in later computations.

You may wonder what happens at the edges of the image. If we place the mask over the image such that the upper left pixel is the one to receive a new value, then part of the mask is "hanging off" the edge of the image. Different ways to handle the edge pixels are summarized in Figure 3.24.

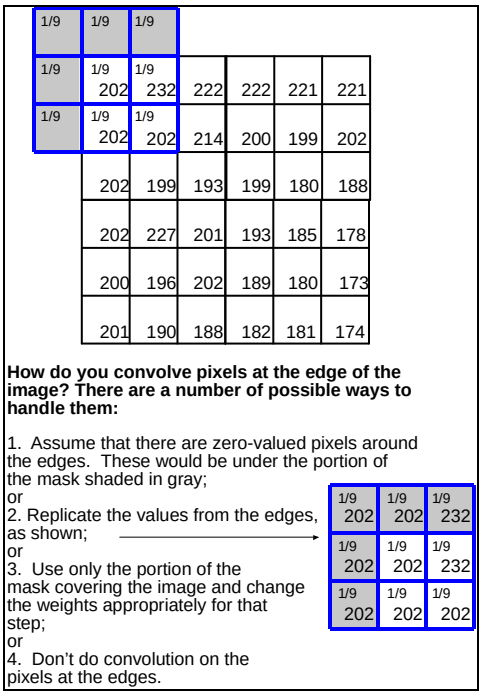


Figure 3.24 Handling edges in convolution

Filters are sometimes used for smoothing or blurring an image. This is done by means of an averaging convolution mask like the one shown in Figure 3.25. You might wonder what would ever motivate you to blur an image, but there are a number of situations in which it is useful. Blurring can be used as a preprocessing step to "pull objects together" so that the main objects in an image can then be detected and extracted. It can be helpful in removing ***image noise***, which is manifested as unwanted speckles, which on a grayscale image can look like sprinkles of salt and pepper. It can soften jagged edges or remove moiré patterns in an undersampled image. It can smooth over the blockiness that can be caused by JPEG compression done at a high compression rate. You also might want to smooth an image for aesthetic reasons. Smoothing convolutions are sometimes referred to as ***low-pass filters*** because their effect is to remove high-frequency components of an image.

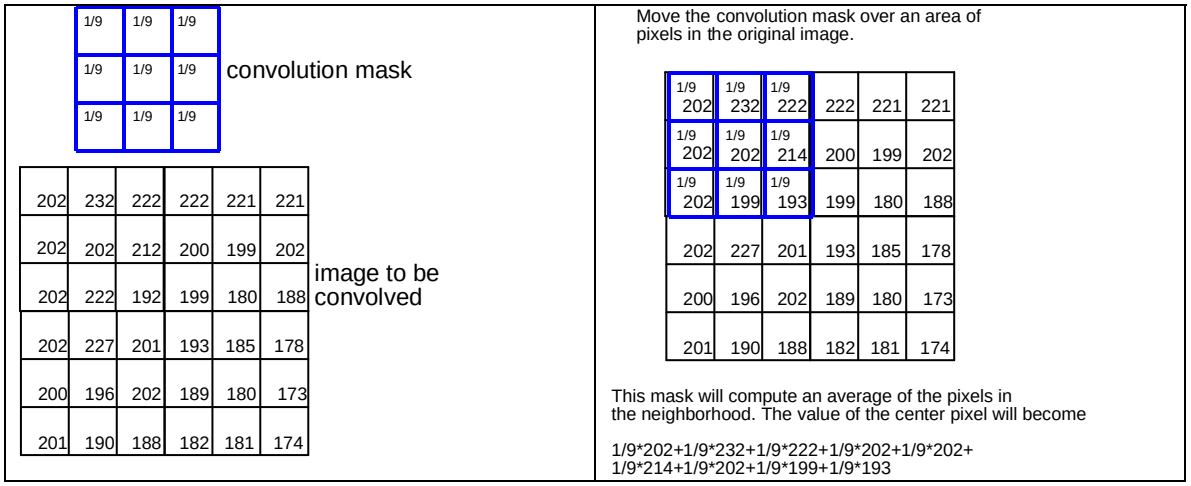


Figure 3.25 Convolution for averaging pixels in a 3 × 3 neighborhood

The mask shown in Figure 3.25 takes an average of the pixels in a 3 × 3 neighborhood. An alternative for smoothing is to use a ***Gaussian blur***, where the coefficients in the convolution mask get smaller as you move away from the center of the mask. It is called a *Gaussian blur* because the magnitudes of the coefficients in both the horizontal and vertical directions take the shape of a Gaussian bell curve. The coefficients result in a weighted average of neighboring pixels. In a pure average, all the coefficients are the same, and they sum to 1. In a weighted average, the coefficients sum to 1, but they are not all the same. For a Gaussian blur, the coefficients closer to the center have more weight than those farther away. A 3 × 3 mask for a Gaussian blur is shown in Figure 3.26. In practice, larger masks are often used for better effect.

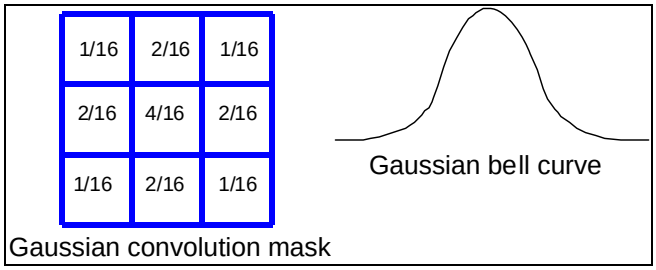


Figure 3.26 Convolution mask for Gaussian blur

3.8.2 Filters in Digital Image Processing Programs

Digital image processing programs like Photoshop and GIMP have an array of filters for you to choose from, for a variety of purposes. Some are corrective filters for sharpening or correcting colors. Some are destructive filters for distorting or morphing images. Some filters create special effects such as simulated brush-strokes, surfaces, and textures. These filters operate by applying convolutions to alter pixel values. If you understand how different convolution masks will affect your images, you can apply the predefined filters more effectively, even using them in combination with each other. You can also create your own customized masks for creative special effects.

One way to design a convolution mask or predict how a predefined mask will work is to apply it to three simple types of image data: a block of pixels that contains a sharp edge, a block where the center value is different from the rest of the values, and a block were all pixel values are the same. To illustrate these different cases, we'll change our representation of the convolution mask slightly, to make it look more like the way custom filters are presented in Photoshop and GIMP. These application programs allow you to create a custom filter that performs a weighted sum of pixels with a scaling factor and an offset. The pixel values are multiplied by the weights in the mask, the product is divided by the scaling factor, and to this total the offset is added. The custom mask window is shown in Figure 3.27. Assume that blank spaces in the mask are 0s. Since the custom mask varies in its shape and the number of values it contains, and it also involves scaling and offset factors, we need to rewrite Equation 3.9 to describe how each new pixel value is derived from the mask. Let's just assume there are n weights in the mask, numbered in row-major order. Let the new pixel value be called q and the weights be given by w_i for $1 \le i \le n$. The pixel value corresponding to weight w_i is p_i (that is, the

pixel "under" that weight in the mask.) The scale factor is s and the offset is c. Then the equation for the new pixel value derived from the mask is

$$q = \left[\left(\sum_{i=1}^{n} w_i p_i \right) / s \right] + c$$

Equation 3.10

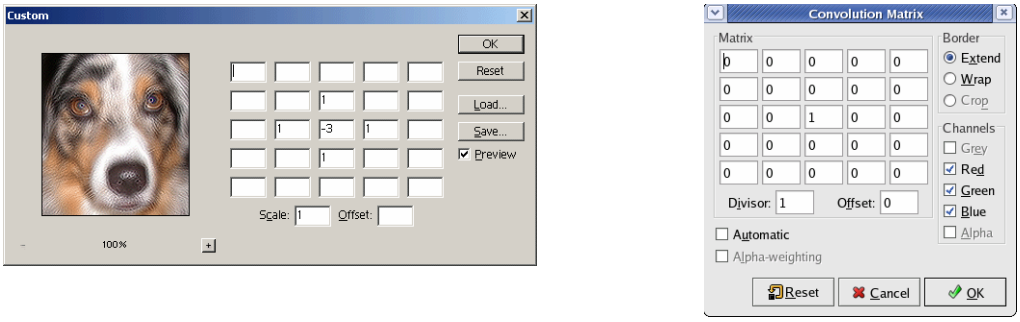


Figure 3.27 Custom filter (from Photoshop and GIMP)

In order to preserve the brightness balance of an image, then $\sum_{i=1}^{n} w_i / s$ should equal 1. You can easily see this in the case where all the pixels under the mask have the same value. If $\sum_{i=1}^{n} w_i / s > 1$, then the center pixel will be become lighter, and if $\sum_{i=1}^{n} w_i / s < 1$ the center pixel will become darker. You can arrange it so that $\sum_{i=1}^{n} w_i / s = 1$ by making some weights positive and some weights negative, by changing the scaling factor, or by a combination of both methods. Each variation you try has a slightly different effect on how you sharpen (or blur) the entire image or affect primarily the edges in the image.

Let's look at a couple of filters that affect edges. Consider the convolution mask in Figure 3.28 and how it would operate on the block of pixels labeled block **a**. (Assume that for block **a**, the pixels values extend infinitely to the left and right with the same values in each row.) If you do the math, you'll see that along a horizontal edge that goes from light to dark as you move from top to bottom, the filter detects the edge, making that edge white while everything else is black. The effect is illustrated in Figure 3.29. If you swap the row of 1s for the row of –1s, the filter will detect horizontal edges that go from black to white rather than from white to black as you move down the image. This would make a white edge along the girl's bangs rather than along the top of her hair in Figure 3.29. If you arrange the 1s and –1s vertically rather than horizontally, the filter detects vertical edges moving one direction or the other.

1	1	1
0	0	0
-1	-1	-1

mask

255	255	255	255
255	255	255	255
255	255	255	255
0	0	0	0
0	0	0	0
0	0	0	0

block **a**

0	0	0	0
0	0	0	0
255	255	255	255
255	255	255	255
0	0	0	0
0	0	0	0

block **b**

The mask above applied to block **a** of pixels yields block **b**.

Figure 3.28 An edge-detection filter

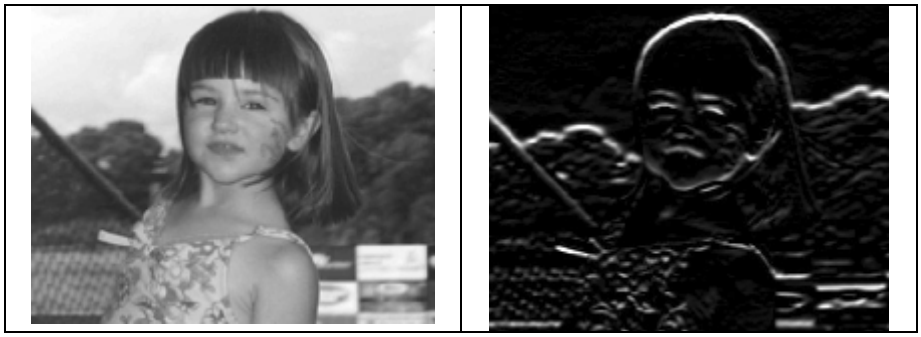
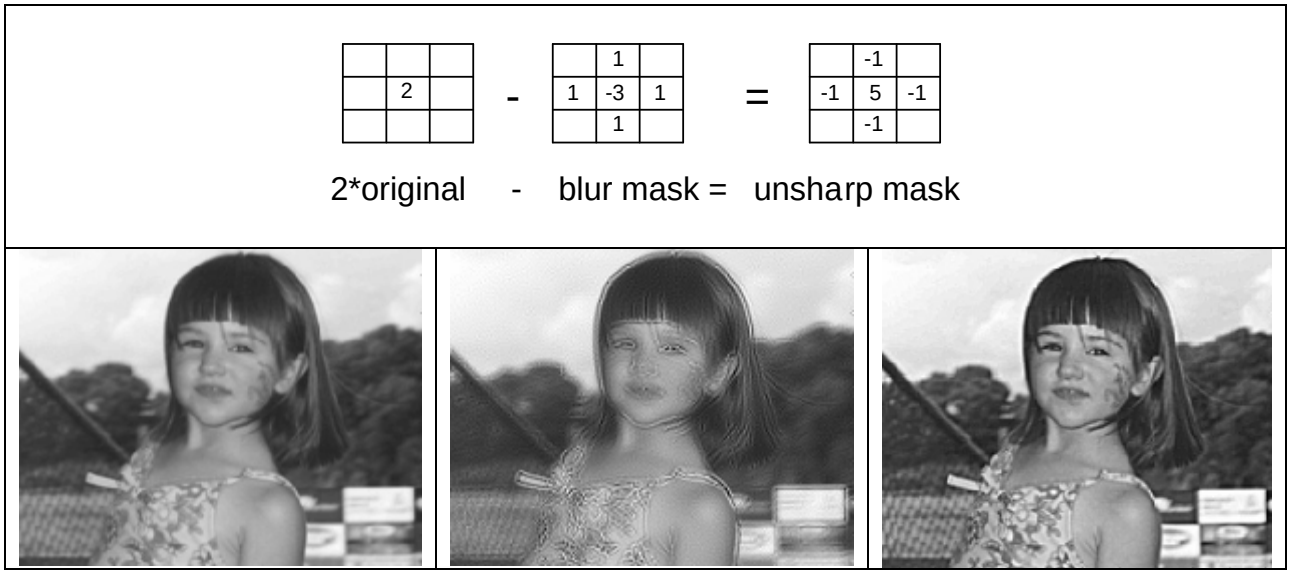


Figure 3.29 Applying an edge-detection filter to a grayscale image

One of the most useful filters in the repertoire is called the ***unsharp mask***. The name is misleading because this filter actually sharpens images. The name is derived from the way the mask is constructed. The idea is that first a blurred ("unsharp") version of the image is produced. Then the pixel values in the original image are doubled, and the blurred version of the image is subtracted from this. The result is a sharpened image, as shown in Figure 3.30. The blur mask shown in Figure 3.30 is different from the one we showed above. Variations of the unsharp mask can be created using different sizes of masks or types of blur masks.



original image	image with blur filter applied	image with unsharp mask applied

Figure 3.30 Unsharp mask

3.9 Resampling and Interpolation

Resampling is a process of changing the total number of pixels in a digital image. There are a number of situations in which resampling might be needed. To understand these, you need to understand the relationship between resolution, the size of an image in pixels, and print size (discussed in Chapter 2).

Here are four scenarios where you might want to change the print size, resolution, or total pixel dimensions of a digital image. See if you can tell which require resampling:

1. You scanned in an 8 ×10 inch photograph at a high resolution (300 pixels per inch, abbreviated *ppi*). You realize that you don't need this resolution since your printer can't print in that much detail anyway. You decide to decrease the resolution, but you don't want to change the size of the photograph when it is printed out. If you ask your image processing program to change the image size from 8 ×10 inches and 300 ppi to 8 ×10 inches and 200 ppi, does the image have to be resampled?

2. You scanned in a 4 × 5 inch image at a resolution of 72 ppi, and it has been imported into your image processing program with these dimensions. You're going to display it on a computer monitor that has 90 ppi, and you don't want the image to be any smaller than 4 × 5 on the display. Does the image have to be resampled?

3. You scanned in an 8 ×10 inch photograph at 200 ppi, and it has been imported into your image processing program with these dimensions.. You want to print it out at a size of 4 × 5 inches. Does the image have to be resampled?

4. You click on an image on your computer display to zoom in closer. Does the image have to be resampled?

So let's see how well you did in predicting when resampling is necessary. The key point to understand is that resampling is required whenever the total number of pixels in a digital image is changed. If the resolution – the ppi – is not changed but the print size *is*, resampling is necessary. Resampling is also required if the print size is not changed but the resolution is. Compare your answers to the answers below.

1. The image has to be resampled in this case. If you have 300 ppi and an image that is 8 × 10 inches, you have a total of 300*8 *10 = 24,000 pixels. You want 200*8 *10 = 16,000 pixels. Some pixels have to be discarded, which is called ***downsampling***.

2. Again, the image has to be resampled. The 4 × 5 image scanned at 72 ppi has pixel dimension of 288 × 360. A computer display that can fit 90 pixels in every inch (in both the horizontal and vertical directions) will display this image at a size of 3.2 × 4 inches. Retaining a size of at least 4 × 5 inches on the computer display requires ***upsampling***, a process of inserting additional samples into an image.

3. The image doesn't have to be resampled to decrease its print size, although you can resample if you choose to. If you specify that you want to decrease the image print size without resampling, then the total number of samples will not change. The resulting image file will have exactly the same total number of pixels as before. However, without resampling, the number of pixels per inch will be greater because you're decreasing the number of inches. If, on the other hand, you specify that you want to decrease the size and resample, then the resolution will not change, but the total number of pixels *will*.

Keeping the same resolution while decreasing the print size implies that you'll have fewer samples in the final image. In that case, the image is downsampled.

4. When you zoom in, the image is being upsampled, but only for display purposes. When you zoom out, the image is being downsampled. The stored image file doesn't change, however.

Recall from Chapter 2 how this works in practice. Figure 3.31 shows the Image Size window in Photoshop and GIMP. The Resample Image is checked. Thus, if you change the width and height in inches, the width and height in pixels will change accordingly so that the resolution does not change. Similarly, if you change the width and height in pixels, the width and height in inches will change.

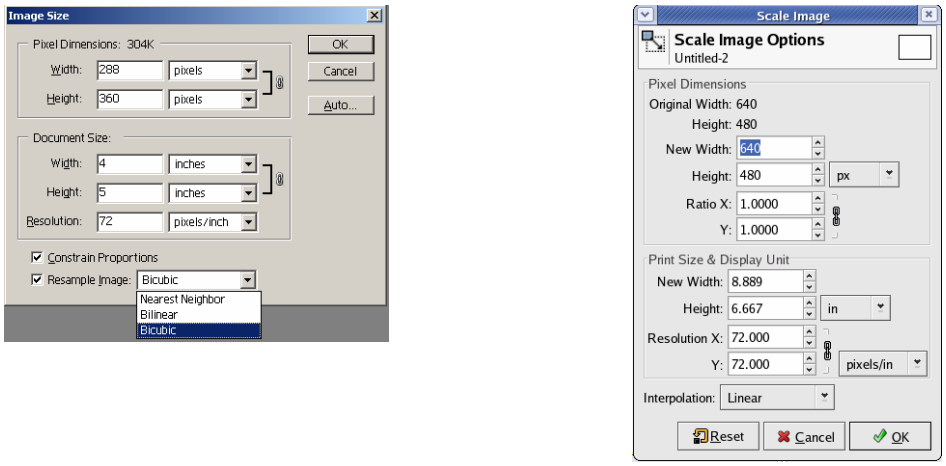


Figure 3.31 Image Size window, Resample Image checked (from Photoshop and GIMP)

The simplest method for upsampling is ***replication*** – a process of inserting pixels and giving them the color value of a neighboring pre-existing pixel. Replication works only if you are enlarging an image by an integer factor. An example of replication on a small block of pixels is shown in Figure 3.32. For each original pixel, a new pixel with the same color value is inserted to the right, below, and diagonally below and to the right.

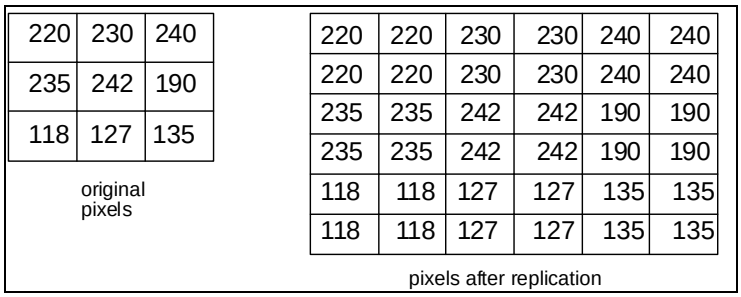


Figure 3.32 Resampling by replication

Correspondingly, the simplest method of downsampling is ***row-column deletion***, the inverse of replication.

Think about how replication and its inverse might affect the quality of an image. Row-column deletion throws away information about the image, so you obviously lose detail. Replication, on the other hand, makes a "guess" about the colors that might have been sampled between existing samples – if the sampling rate had been higher. The

values that are introduced in replication may not be the exact colors that would have been detected if the image had been sampled at a higher resolution to begin with. Thus, even though an upsampled image gains pixels, it doesn't get any sharper. In fact, usually the image loses quality. Since the new pixel values are copied from neighboring pixels, replication causes blockiness in the resampled image. Magnifying a view of an image in an image processing program can be done with simple replication. The image gets bigger, but the blockiness caused by upsampling becomes increasingly evident the more you zoom in. This is illustrated in Figure 3.33. Of course there is no harm done to the file, since the pixel values are upsampled only for display purposes. The values stored in the image file don't change.

Figure 3.33 Image resampled as you zoom in

Figure 3.34 Image resampled to twice and then four times the original number of pixels using replication

You'll notice a similar effect if you upsample a digital image in order to print it out at larger size. Say that you scan in a 4 × 5 inch image at 200 pixels per inch but

decide that you want to print it out at a size of 8 ×10 inches. Increasing the size and keeping the resolution at 200 ppi requires upsampling, since you'll end up with more pixels than you originally captured. But keep in mind that the picture you print out at 8 ×10 inches can't be any clearer or more detailed than the original 4 × 5 inch image. This is illustrated in Figure 3.34.

The point is that the only "true" information you have about an image is the information you get when the image is originally created – by taking a digital photograph or scanning in a picture. Any information you generate after that – by upsampling – is only an approximation or guess about what the original image looked like. Thus, it's best to scan in digital images with sufficient pixel dimensions from the outset. For example, you could capture the 4 × 5 inch image at 400 ppi so that you could then increase its print size to 8 ×10 inches without resampling. If you deselect "resample image" in your image processing program and then change the image's size to 8 ×10 inches, the number of pixels per inch will change so that the total number of pixels does not change. That is, the resolution will automatically be cut in half as the print size is doubled. Then you'll have your 8 ×10 inches at a resolution of 200 ppi, which may be good enough for the print you want to make, and you won't have had to generate any new pixels that weren't captured in the original scan.

Even with your best planning, there will still be situations that call for resampling. Fortunately, there are interpolation methods for resampling that give better results than simple replication or discarding of pixels. ***Interpolation*** is a process of estimating the color of a pixel based on the colors of neighboring pixels. In Figure 3.31, notice the drop-down box beside the Resample Image checkbox. This is where you choose the interpolation method – either nearest neighbor, bilinear, or bicubic. Nearest neighbor is essentially just replication when the scale factor is an integer greater than 1. However, it can be generalized to non-integer scale factors and described in a manner consistent with the other two methods, as we'll do below. We'll describe these algorithms as they would be applied to grayscale images. For RGB color, you could just apply the procedures to each of the color channels.

For this discussion, we describe ***scaling*** as an affine transformation of digital image data that changes the total number of pixels in the image. An ***affine transformation*** is one that preserves colinearity. That is, points that are on the same line remain colinear and lines that are parallel in the original image remain parallel in the transformed image. Clearly, *scaling* is just another word for *resampling*, but we introduce this synonymous term so that we can speak of a scale factor. If the ***scale factor*** *s* is greater than 1, the scaled image will increase in size on the computer display. If it is less than 1, the image will decrease in size. The scale factor doesn't have to be the same in both directions, but we will assume that it is in this discussion for simplicity.

With the scale factor *s* we can define a general procedure for resampling using interpolation, given below as Algorithm 3.1.

```
algorithm resample
/*Input:  A grayscale image f of dimensions w × h.
Scale factor s.
Output:  fs, which is image f  enlarged or shrunk by scale factor s.*/
{
  w′ = w * s
  h′ = h * s
/* create a new scaled image fs with dimensions w′ and h′*/
  for i = 0 to h′–1
    for j = 0 to w′–1 {
    fs(i,j) = interpolate(i, j, f, method)
  }
}

algorithm interpolate(i, j, f, method) {
/*i and j are pixel coordinates in the scaled image fs.*/
  a = i/s
  b = j/s
/*a and b are coordinates in the original image, f .  Note that a and b are not necessarily
integers.  Interpolation entails finding integer-valued coordinates that are neighbors to a
and b.*/
  if method = "nearest neighbor" then
    return f(round(a), round(b))
  else if method = "bilinear" then {
/*  Find the coordinates of the top left pixel in the neighborhood of (a,b) */
    x = floor(a)
    y = floor(b)
/*Each pixel's weight is based on how close it is to (a,b)*/
    value = 0
    for m = 0 to 1  {
      for n = 0 to 1 {
        t = a – (x + m)
        u = b – (y + n)
        value = value + (1 – |t|)*(1-|u|)
      }
    }
    return value
  }
  else if method = "bicubic" then {
/*To simplify this algorithm, we assume that f(x,y) always has sixteen neighbors within
the bounds of the image.  In practice, the algorithm needs to be adjusted to account for
pixels at edges of the image*/
    x = floor(a)
    y = floor(b)
    value = 0
```

```
/*Consider the sixteen neighboring pixels around around (a,b)
    for m = -1 to 2
      for n = -1 to 2 {
/*Determine how far position (a,b) is from each of the sixteen neighbors*/
        neighborX = x+m
        neighborY = y+n
        t = |neighborX – a|
        u = |neighborY – b|
        if t < 1
          x_coeff = 1 – 2*t^2 + t^3
        else
          x_coeff = 4 – 8*t + 5*t^2 – t^3
        if u < 1
          y_coeff = 1 – 2*u^2 + u^3
        else
          y_coeff = 4 – 8*u + 5*u^2 – u^3
        value = value + x_coeff * y_coeff * f(neighborX, neighborY)
      }
    return value
  }
}
```

Algorithm 3.1

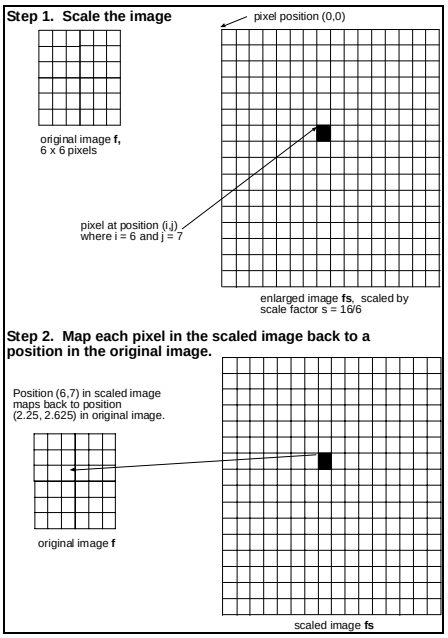


Figure 3.35 The first two steps in resampling

Algorithm 3.1 shows describes three different interpolation methods – nearest neighbor, bilinear, and bicubic interpolation. All three begin with the same two steps, illustrated in Figure 3.35. First, the dimensions of the scaled image are determined by multiplying the original dimensions by the scale factor. A new bitmap of the scaled dimensions is created. In the example, a 6 × 6 pixel image called ***f*** is enlarged to 16 × 16

image called ***fs***, so the scale factor *s* is 16/6. The values of the pixels in the scaled image are to be determined by interpolation. The second step is to map each pixel position in the scaled image back to coordinates within the original image. A pixel at position (*i,j*) in ***fs*** maps back to position (*i/s,j/s*) in ***f***. Clearly, not all pixel positions in ***fs*** map back to integer positions in ***f***. For example, (6,7) maps back to (2.25, 2.625). The idea in all three interpolation algorithms is to find one or more pixels close to position (*i/s,j/s*) in ***f,*** and use their color values to get the color value of ***fs****(i,j)*.

Supplements on interpolation for resampling:

worksheet

Nearest neighbor interpolation simply rounds down to find one "close" pixel whose value is used for ***fs****(i,j)*. In our example, ***f****(2,3)* is used as the color value of ***fs****(6,7)*. If you think about this, you'll realize that when *s* is an integer greater than 1, the nearest neighbor algorithm is effectively equivalent to pixel replication. However, it also works with non-integer scale factors, as shown in our example.

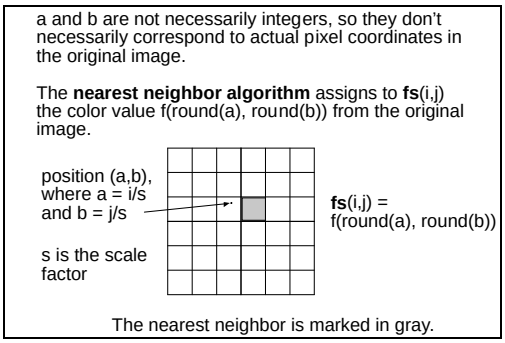


Figure 3.36 Nearest neighbor interpolation

To describe the three interpolation algorithms discussed in Section 3.5, it is sufficient to specify the convolution mask for each. Let's do this for nearest neighbor interpolation. We have an $w \times h$ image called *f* that is being scaled by factor *s*, with the result being written to a new image bitmap *fs* of dimensions $w' = w * s$ and $h' = h * s$. The color value of each pixel ***fs****(i,j)* is obtained by mapping (i, j) back to coordinates in ***f*** with $a = i / s$ and $b = j / s$.

Let $x = floor(a)$ and $y = floor(b)$. Then the neighborhood of (a,b) in ***f*** consists of ***f****(x,y)*, ***f****(x+1,y)*, ***f****(x,y+1)*, and ***f****(x+1,y+1)*. For each pixel ***fs****(i,j)*, define the nearest neighbor convolution mask $\boldsymbol{h}_{nn}(m,n)$ to be a 2×2 matrix of coefficients as follows:

for $0 \le m \le 1$ and $0 \le n \le 1$,

$\boldsymbol{h}_{nn}(m,n) = 1$ if $-0.5 \le (x+m) - a < 0.5$ and $-0.5 \le (y+n) - b < 0.5$

otherwise $\boldsymbol{h}_{nn}(m,n) = 0$

This effectively puts a 1 in the pixel position closest to (a,b) and a 0 everywhere else, so $fs(i, j)$ takes the value of its single closest neighbor, as shown in the example in Figure 3.37. Note that the position of the 1 in the mask depends on the location of (a,b). The mask is applied with its upper left element corresponding to ***f****(x,y)*.

(You may notice that the numbering of the mask positions is not "flipped" the same way as it was in the definition of the linear convolution above. The convolutions for interpolation are different in that they are not applied to every pixel in the original image. The concept is the same, however.)

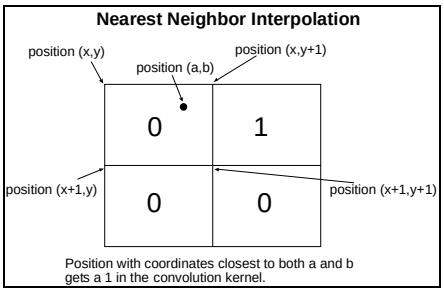


Figure 3.37 Example of a nearest neighbor convolution mask

Bilinear interpolation uses four neighbors and makes ***fs****(i,j)* a weighted sum of their color values. The contribution of each pixel toward the color of ***fs****(i,j)* is a function of how close the pixel's coordinates are to (*a*,*b*). The neighborhood is illustrated in Figure 3.38. The method is called *bilinear* because it uses two linear interpolation functions, one for each dimension. This is illustrated in Figure 3.39. Bilinear interpolation requires more computation time than nearest neighbor, but it results in a smoother image, with fewer jagged edges.

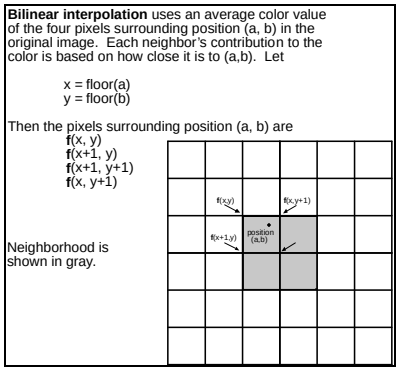


Figure 3.38 Bilinear interpolation

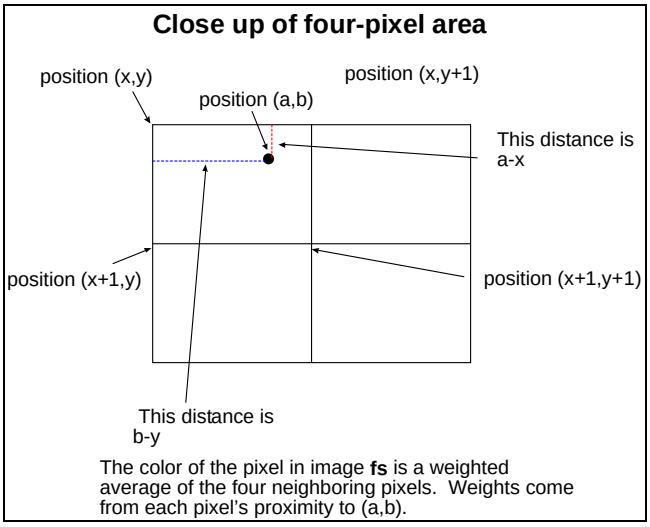


Figure 3.39 Weights used in bilinear interpolation

To specify the convolution mask for bilinear interpolation, let $\boldsymbol{t}(m,n) = a - (x+m)$ and $u(m,n) = b - (y+n)$ for $0 \le m \le 1$ and $0 \le n \le 1$. Then the mask $\boldsymbol{h}_{bl}(m,n)$ is defined as $\boldsymbol{h}_{bl}(m,n) = (1-|t|)(1-|u|)$.

Bicubic interpolation uses a neighborhood of sixteen pixels to determine the value of ***fs(i,j)***. The neighborhood of (a,b) extends from $x-1$ to $x+2$ in the vertical direction and from $y-1$ to $y+2$ in the horizontal direction, as illustrated in Figure 3.40. The method is called *bicubic* because the weighted average of pixels in the 4 × 4 pixel neighborhood is based on two cubic interpolation functions, one for each dimension. The cubic functions used in our example are sometimes referred to as a "Mexican Hat" convolution mask. Other functions can be used. Bicubic interpolation requires even more computation time and memory than bilinear interpolation, but it creates a smoother image while preserving detail from the original image.

Supplement on interpolation for resampling:

worksheet

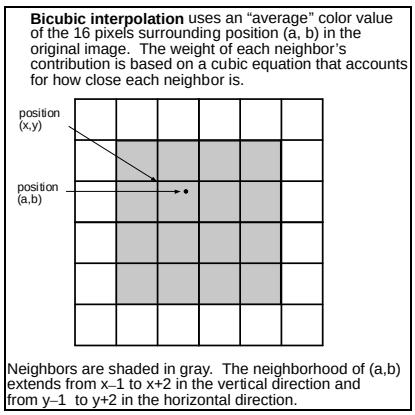


Figure 3.40 Bicubic interpolation

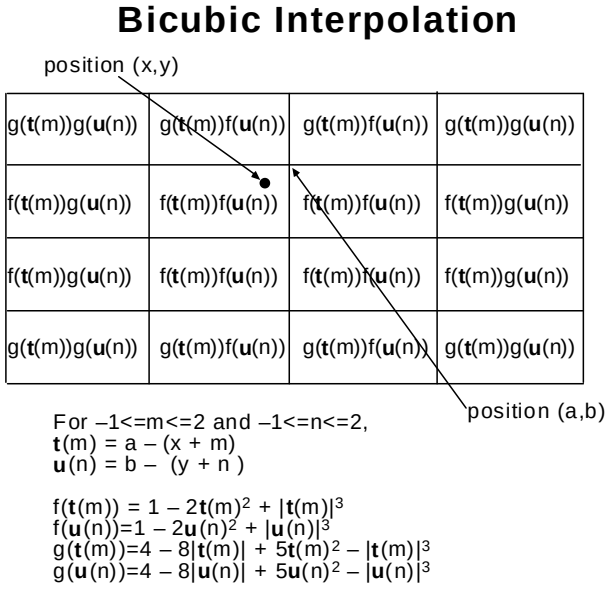


Figure 3.41 Convolution mask for bicubic interpolation

3.10 Digital Image Compression

3.10.1 LZW Compression

LZW compression is a method that is applicable to both text and image compression. LZW stands for Lempel-Ziv-Welch, the creators of the algorithm. Lempel and Ziv published the first version of the algorithm in 1977 (called LZ77), and the procedure was revised and improved by Welch in 1984. The algorithm was first widely used in the 1980s in the *compress* Unix utility. It then went through a number of versions and patents with Sperry, Unisys, and CompuServ Corporations. The method is commonly applied to GIF and TIFF image files. The patents are relevant to GIF and TIFF software developers but do not hamper an individual's right to own or transmit files in GIF or TIFF format. In any case, Unisys's main patents for LZW expired in 2003.

The LZW algorithm is based on the observation that sequences of color in an image file (or sequences of characters in a text file) are often repeated. Thus, the algorithm uses a sliding expandable window to identify successively longer repeated sequences. These are put into a code table as the file is processed for compression. An ingenious feature of the algorithm is that the full code table does not have to be stored with the compressed file; only the part of it containing the original colors in the image is needed, and then the rest – the codes for the sequences of colors – can be regenerated dynamically during the decoding process. Let's see how this works.

With a first pass over the image file, the code table is initialized to contain all the individual colors that exist in the image file. These colors are encoded in consecutive integers. Now, imagine that the pixels in the image file, going left to right and top to bottom, are strung out in one continuous row. After initialization, the sliding expandable window moves across these pixels. The window begins with a width of one pixel. (The height is always one pixel.) If the pixel sequence is already in the code table, the window is successively expanded by one pixel until finally a color sequence not in the table is under the window. Say that this sequence is n pixels long. Then the code for the sequence that is $n-1$ pixels long is output into the compressed file, and the n-pixel-long sequence is put into the code table. This continues until the entire image is compressed. The procedure is illustrated in Figure 3.42.

Supplements on LZW compression:

interactive tutorial

programming exercise

worksheet

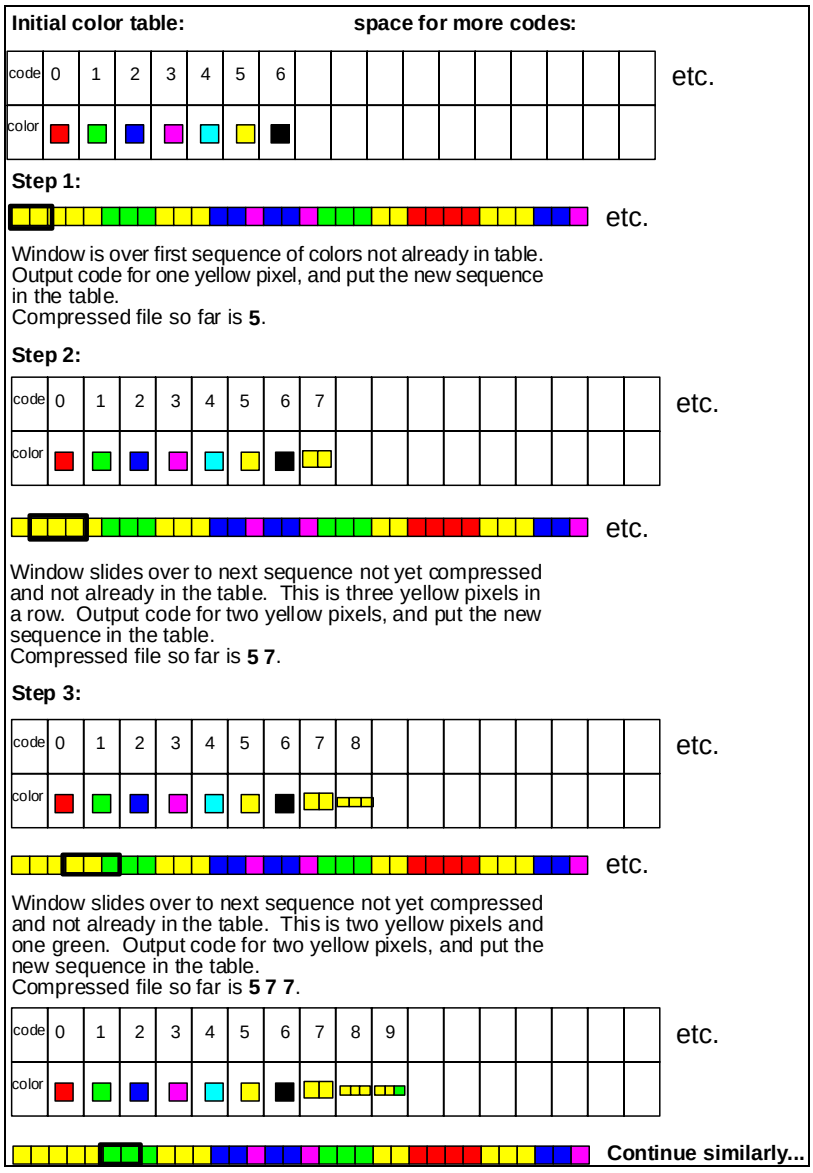


Figure 3.42 A partial trace of LZW compression

An algorithm that will accomplish the procedure pictured in Figure 3.42 is given below.

```
algorithm LZW
/*Input:  A bitmap image.
Output:  A table of the individual colors in the image and a compressed version of the
file.
Note that + is concatenation.*/
{
  initialize table to contain the individual colors in bitmap
  pixelString = first pixel value
  while there are still pixels to process {
    pixel = next pixel value
    stringSoFar = pixelString + pixel
    if stringSoFar is in the table then
      pixelString = stringSoFar
    else  {
      output the code for pixelString
      add stringSoFar  to the table
      pixelString = pixel
    }
  }
  output the code for pixelString
}
```

Algorithm 3.2 LZW compression algorithm

```
algorithm LZW_decompress
/*Input:  Compressed bitmap image and table of individual colors in image.
Output:  Decompressed image.*/
{
 stringSoFar = NULL
 while there are still codes to process in the code string {
   code = next code in the code string
   colors = the colors corresponding to code in the table
   if colors == NULL      /*Case where code is not in the table*/
 /*stringSoFar[0] is the first color in stringSoFar*/
      colors = stringSoFar + stringSoFar[0]
   output colors
   if stringSoFar != NULL
     put stringSoFar + colors[0] in the table
   stringSoFar = colors
 }
}
```

Algorithm 3.3 LZW decompression algorithm

The decoding process requires only a table initialized with the colors in the image. From this information, the remaining codes are recaptured as the decoding progresses. The decompression algorithm is given as Algorithm 3.3.

3.10.2 Huffman Encoding

Huffman encoding is another lossless compression algorithm that is used on bitmap image files. It differs from LZW in that it is a ***variable-length encoding*** scheme; that is, not all color codes use the same number of bits. The algorithm is devised such that colors that appear more frequently in the image are encoded with fewer bits. Thus, it is a form of entropy encoding, like the Shannon-Fano algorithm described in Chapter 1. The Huffman encoding algorithm requires two passes: (1) determining the codes for the colors and (2) compressing the image file by replacing each color with its code.

Supplements on Huffman encoding:

interactive tutorial

programming exercise

worksheet

In the first pass through the image file, the number of instances of each color is determined. This information is stored in a frequency table. A tree data structure is built from the frequency table in the following manner: A node is created for each of the colors in the image, with the frequency of that color's appearance stored in the node. These nodes will be the leaves of the code tree. Let's use the variable *freq* to hold the frequency in each node. Now the two nodes with the smallest value for *freq* are joined such that they are the children of a common parent node, and the parent node's *freq* value is set to the sum of the *freq* values in the children nodes. This node-combining process repeats until you arrive at the creation of a root node. The algorithm for this process is given in Algorithm 3.4, and a short trace is given in the figures below.

Consider this simple example. Say that your image file has only 729 pixels in it, with the following colors and the corresponding frequencies:

white 70
black 50
red 130
green 234
blue 245

The initial nodes are as pictured in Figure 3.43. The arrangement of the nodes is not important, but moving nodes around as the tree is constructed can make the tree easier to understand.

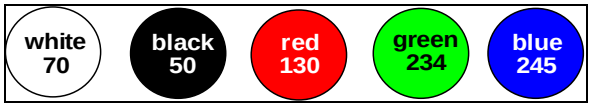


Figure 3.43 Leaf nodes of Huffman tree

```
algorithm Huffman_encoding
/*Input:  Bitmap image.
Output:  Compressed image and code table.*/
{
  /*Let color_freq[] be the frequency table listing each color that appears in the image
and how many times it appears.  Without loss of generality, assume that all colors from 0
to n-1 appear in the image.*/
  initialize color_freq
/*Assume each node in the Huffman tree contains variable c for color and variable freq
for the number of times color c appears in the image.*/
  for i = 0 to n-1 {
/*Let nd.c denote the c field of node nd*/
    create a node nd such that nd.c = i and nd.freq = color_freq[i]
  }
  while at least two nodes without a parent node remain {
    node1 = the node that has the smallest freq among nodes remaining that have no
parent node
    node2 = the node that has the second smallest freq among nodes remaining that have
no parent node
/*Assume some protocol for handling cases where two of sums are the same*/
    nd_new = a new node
    nd_new.freq = node1.freq + node2.freq
   make nd_new the parent of node1 and node2
  }
/*Assign codes to each color by labeling branches of the Huffman tree*/
    label each left branch of the tree with a 0 and each right branch with a 1
  for each leaf node {
    travel down the tree from the root to each leaf node, gathering the code for the color
associated with the leaf nodes
    put the color and the code in a code table
  }
  using the code table, compress the image file
}
```

Algorithm 3.4

Combining nodes into a Huffman tree proceeds as pictured in Figure 3.44. Note that at any point in this process, any two nodes that do not already have a parent node can be combined – as long as they are the nodes with the least value. That is, sometimes you might combine two leaf nodes, sometimes two interior nodes, or sometimes a leaf node with an interior node. Also note that if two minimum-value nodes have the same value, the choice of which one to use is arbitrary. This implies that there may be more than one legal Huffman tree – and thus more than one possible code table – derivable from a set of leaf nodes (i.e., for a given image).

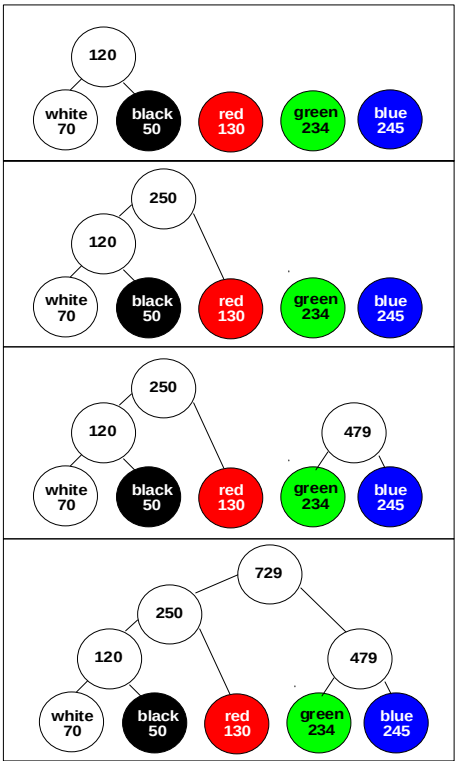


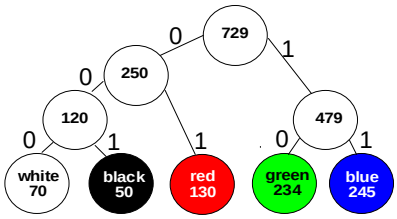
Figure 3.44 Combining nodes to create a Huffman tree

Once the tree has been created, the branches are labeled with 0s on the left and 1s on the right. Then for each leaf node, you traverse the tree from the root to the leaf, gathering the code on the way down the tree. This is pictured in Figure 3.45.

After the codes have been created, the image file can be compressed using these codes. Say that the first ten pixels in the image are wwwkkwwbgr (with black abbreviated as k, white as w, red and r, green as g, and blue as b).
The compressed version of this string of pixels, in binary, is

0000000000010010000001110 01

Label branches with 0s on the left and 1s on the right.



For each leaf node, traverse tree from root to leaf node gather code for the color associated with the leaf node.

white	000
black	001
red	01
green	10
blue	11

Note that not all codes are the same number of bits, and no code is a prefix of any other code.

Figure 3.45 Creating codes from a Huffman tree

If you think about how the Huffman encoding algorithm works, you'll understand why it is designed the way it is. By combining least-valued nodes first and creating the tree from the bottom up, the algorithm ensures that the colors that appear least frequently in the image have the longest codes. Also, because the codes are created from the tree data structure, no code can be a prefix of another code. This is a necessary characteristic of the codes because the codes are not necessarily the same length. Imagine that you had the string 11101011100000001001 to decode, and there was both a 1 (as black, say) and a 11 (the code for white) in the code table. How would you know if you should decode the first two 1s as two black pixels or as a single white pixel? This is the problem you would encounter if one code was a prefix for another.

To understand decoding, picture what you would have in the compressed file. Either a code table must be saved, the Huffman tree (in which the codes are implicit) must be saved, or the frequency table must be saved so that the Huffman tree can be used for the decoding. The easiest way to imagine the decoding process is to assume that the Huffman tree is available to the decoder. Now picture how the decoder would decode the encoded bitmap. Bits are read sequentially from the encoded file, guiding the traversal down the Huffman tree until a leaf node is encountered. A 0 bit means take a left branch. A 1 bit means take a right branch. When the decoder arrives at a leaf node, it finds the color associated with the bit sequence just consumed. Then the code for this sequence is output, and the search continues for the next sequence, with the decoder moving back to the root of the Huffman tree.

To determine the compression rate for a Huffman encoded file, we need to make some assumptions about how the codes are stored in the file. Think about the implications of storing the codes in a code table. The first column of each row would contain one of the original colors from the image file – let's say a 24-bit RGB color. Associated with each color is its code. But since the codes are of variable length, the decoder would have to know how many bits there are in this code. Thus there would first have to be a value indicating the length of this code. An alternative to storing the code table is to store the frequency table and have the decoder regenerate the Huffman tree. Then there would be the encoded image itself in the compressed file. With this in mind, let's compute the compression rate for our example.

Assuming 24 bit color, the original image size is computed as follows:

729 pixels * 24 bits/pixel = 17,496 bits in the original image

Now for the compressed image. We can get the size of the compressed image by considering that we have 70 white bits encoded with three bits each, 50 black pixels encoded with three bits each, 130 red pixels encoded with two bits each, 234 green pixels encoded with two bits each, and 245 blue pixels encoded with two bits each. That's (70+50) * 3 + (130+234+245) * 2 = 1,578 bits in the compressed image. Thus the compression rate is 17,496:1,578, better than 11:1.

You may notice that we didn't really need 24 bits per color if we had only five colors. We could have used just eight bits. Even if we assume only eight bits per color in the original file, we get a compression rate of 3.7:1. This is just a small example (a 27 × 27 pixel image), but it gives you the basic concepts.

Huffman encoding is useful as a step in JPEG compression, as we will see in the next section.

3.10.3 JPEG Compression

Supplements on JPEG compression:

interactive tutorial

worksheet

JPEG is an acronym for Joint Photographic Experts Group. In common usage, ***JPEG compression*** refers to a compression algorithm suitable for reducing the size of photographic images or continuous-tone artwork – pictures where the scene transitions move smoothly from one color to another, as opposed to cartoon-like images where there are sharp divisions between colors. If you see a file with the *.jpg* or *.jpeg* suffix, you can assume it has been compressed with the JPEG method, but it is also possible to apply JPEG compression to images saved in TIFF, PICT, and EPS files. Although JPEG is a lossy compression method, it is designed so that the information that is lost is not very important to how the picture looks – that is, the algorithm removes closely-spaced changes in color that are not easily perceived by the human eye. This is made possible by transforming the image data from the spatial domain to the frequency domain. When the image is represented in terms of its frequency components, it is possible to "throw away" the high frequency components, which correspond to barely-perceptible details in color-change.

Another advantage of JPEG compression is that image processing programs allow you to choose the JPEG compression rate, so you can specify how important the image size is as opposed to the image's fidelity to the original subject. JPEG compression on a 24-bit color image yields an excellent compression rate. With a rate of about 10:1 or 20:1, you'll notice hardly any difference from the original to the compressed image. Even compression rates up to 50:1 can give acceptable results for some purposes. The main disadvantage to JPEG compression is that it takes longer for the encoding and decoding than other algorithms require, but usually the compression/decompression time is well justified when compared to the savings in storage space and download time required for the image file. Without JPEG compression, most people would not want to be bothered with the time it would take to download pictures on web pages.

To be precise, the term JPEG does not refer to a standardized file format, but only more generally to a method of compressing image files that was created by the Joint Photographic Experts Group in 1990. A standardized JPEG file format with the name ***JFIF*** (JPEG File Interchange Format) was introduced about a year later by C-Cube Microsystems. An alternative file format designed by C-Cube, called TIFF/JPEG, offers the ability to store more information about the image file, but the simpler JFIF format has become the *de facto* standard.

The algorithm we describe below is the basic method that has been used for JPEG compression since the 1980s, adapted from a scheme proposed by Chen and Pratt. The key step in this algorithm is the transformation of image data from the spatial to the frequency domain by means of the discrete cosine transform (DCT). The main steps are listed in Algorithm 3.5 below. Let's consider the motivation and then examine some of the details of each of these steps.

```
algorithm jpeg
/*Input:  A bitmap image in RGB mode.
Output:  The same image, compressed.*/
{
    Divide image into 8 × 8 pixel blocks
    Convert image to a luminance/chrominance model such as YCbCr (optional)
    Shift pixel values by subtracting 128
    Use discrete cosine transform to transform the pixel data from the spatial domain to
    the frequency domain
    Quantize frequency values
    Store DC value (upper left corner) as the difference between current DC value and
    DC from previous block
    Arrange the block in a zigzag order
    Do run-length encoding
    Do entropy encoding (e.g., Huffman)
}
```

Algorithm 3.5

Step 1. Divide the image into 8 × 8 pixel blocks and convert RGB to a luminance/chrominance color model.

Motivation: The image is divided into 8 × 8 pixel blocks to make it computationally more manageable for the next steps. Converting the color mode to a luminance/chrominance model makes it possible to remove some of the chrominance information, to which the human eye is less sensitive, without significant loss of quality in the image.

Details: For efficiency reasons, JPEG compression operates on 8 × 8 pixel blocks on the image file. If the file's length and width are not multiples of 8, the bitmap can be padded and the extra pixels removed later.

> **Aside**: If you know something about computational complexity, you can understand how dividing the image into 8 × 8 blocks makes execution of the DCT less computationally expensive. Let's assume that an $O(n^3)$ DCT algorithm is used, where n is the dimension of the pixel block horizontally and vertically. If you apply this algorithm on a 16 × 16 block, the algorithm takes on the order of $16^3 = 4096$ steps. Alternatively, you can apply the algorithm on four 8 × 8 blocks. This requires on the order of $4 * 8^3 = 2048$ steps, which is less than 16^3. Applying the DCT on four 8 × 8 blocks covers the same total pixel area as applying it to 16 × 16 block, but in fewer steps. It's computationally more manageable to use smaller blocks.

JPEG compression can be performed on 24-bit color or 8-bit grayscale images. If the original pixel data is in RGB color mode, then there is an 8 × 8 pixel block for each of the color channels. The blocks for these color channels are processed separately with the steps described below, but the three blocks are grouped so that the image can be reconstructed upon decoding.

Converting the image file from RGB to a model like YCbCr makes it possible to achieve an even greater compression rate by means of chrominance subsampling. As discussed in Chapter 2, the YCbCr color model represents color in terms of one luminance component, Y, and two chrominance components, Cb and Cr. The human eye is more sensitive to changes in light (i.e., luminance) than in color (i.e., chrominance). Thus, we need less detailed information with regard to chrominance, since we won't

notice very subtle differences anyway. Transforming directly from RGB to one of the luminance/chrominance models is a straightforward linear operation, as discussed in Chapter 2.

Simply doing this transformation doesn't reduce the number of bits per pixel. However, it does separate the pixel data so that some of it can be discarded. ***Chrominance subsampling*** (also called ***chrominance downsampling***) is a process of throwing away some of the bits used to represent pixels – in particular, some of the color information. For example, with YCbCr color mode, we might choose to save only one Cb value and one Cr value but four Y values for every four pixel values. Three commonly-used subsampling rates are pictured in Figure 3.46. The conventional notation for luminance/chrominance subsampling is in the form $a : b : c$. Common subsampling rates are 4:1:1, 4:2:0, and 4:2:2. To understand what these numbers represent, count the number of samples taken for Y and Cb (or Cr, since they are the same) in each pair of four-pixel-wide rows. a is the number of Y samples in both rows. b is the number of Cb samples in the first row (and also the number of Cr samples). c is the number of Cb (and Cr samples) in the second row. Note that we have not specified how the Cb and Cr values are derived. Just one of the values in a sub-block could be used, or the values could be averaged. (In fact, MPEG-1 and MPEG-2 video compression use different methods for determining the single chrominance values corresponding to four luminance values in 4:2:0 downsampling.)

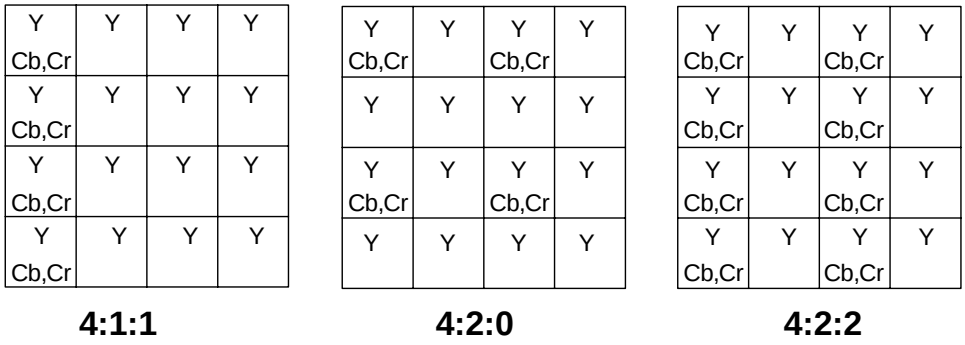


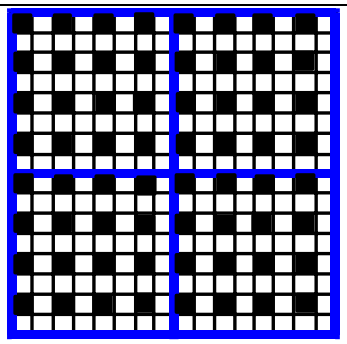
Figure 3.46 Chrominance subsampling

With RGB color mode, we can imagine that for each 8 × 8 pixel section of the image, there are actually three 8 × 8 blocks to be processed, one for each of the R, G, and B color channels. With YCbCr color mode, we have to picture this a little differently. We begin by dividing the image into 16 ×16 pixel macroblocks. Then, with 4:2:0 chroma downsampling, we get four 8 × 8 blocks of Y data for every one 8 × 8 block of Cb and one 8 × 8 block of Cr data. This is pictured in Figure 3.47. Each of the blocks undergoes the remaining steps of the algorithm, and the resulting compressed data is reconstructed upon decoding.

You can see that using a luminance/chrominance color model and then chrominance subsampling reduces the image file size even before the rest of the compression steps are performed. Consider the reduction in file size if 4:2:0 downsampling is used by counting how many bytes area of pixels with a width of four pixels and a height of two pixels – eight pixels total. Assuming that each component requires one byte, an unsubsampled image requires 8 * 3 = 24 bytes. Subsampling at a

rate of 4:2:0 requires 8 + (2 * 2) = 12 bytes (eight for the Y component but only two for each of the Cb and Cr). This is a 2:1 savings.

Chrominance subsampling is not a required step in JPEG compression, and some JPEG compressors allow you to turn this option off if you think that subsampling will compromise the desired sharpness of your image. Usually, however, this isn't necessary.



With 4:2:0 YCbCr chrominance subsampling, a 16 x 16 macroblock yields:
--four 8 x 8 blocks of Y values
--one 8 x 8 block of Cb values
--one 8 x 8 block of Cr values
Each black location represents one Cb and one Cr sample taken for a block of four pixels. Y samples are taken at all locations.

Figure 3.47 Chrominance subsampling

We will trace through an example of JPEG compression performed on an 8 × 8 block of a grayscale image. In grayscale images, the RGB color channels all have the same value, so it's necessary to store only one byte value per pixel. Thus, our example will need to show the processing of only one 8 × 8 block for an 8 × 8 pixel area. But keep in mind that if 4:2:0 chroma subsampled YCbCr color is used, for every 16 × 16 pixel area, there are four 8 × 8 blocks of Y data and one each of Cb and Cr data. If RGB color is used, there are three 8 × 8 blocks – one each for R, G, and B – for every 8 × 8 pixel area.

Step 2. Shift values by –128 and transform from the spatial to the frequency domain.

Motivation: On an intuitive level, you can think of shifting the values by –128 as looking at the image function as a waveform that cycles through positive and negative values. This step is in preparation for representing the function in terms of its frequency components. Transforming from the spatial to the frequency domain makes it possible to remove high frequency components. High frequency components are present if color values go up and down quickly in a small space. These small changes are barely perceptible in most people's vision, so removing them does not compromise image quality significantly.

Details: Let's review briefly what was covered in Chapter 2 concerning the spatial versus the frequency domain: When an $M \times N$ digital image is represented in the spatial domain, it can be stored in a two-dimensional array where each element in the array is the color value – or amplitude – of the image at that point. Thus, a two-

dimensional digital image bitmap defines a surface, each pixel value telling how high the surface is at a given point. In this sense, we can view the image data as a two-dimensional waveform. Figure 3.48 shows an 8 × 8 grayscale pixel area, and Figure 3.49 gives the corresponding surface in the spatial domain. Viewing the image data as a waveform leads us to an alternative representation of the data that is equivalent to the spatial domain. When digital image data is transformed from the spatial to the frequency domain, each value in the $M \times N$ array indicates "how much" of each frequency component exists in the waveform. That is, the elements in the array are coefficients by which we multiply the cosine basis functions such that these functions can be summed to yield the surface of the image.

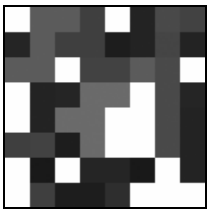


Figure 3.48 8 × 8 pixel area, enlarged

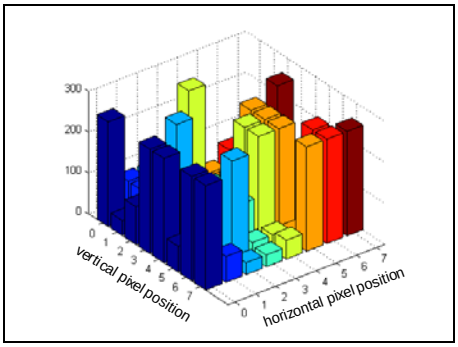


Figure 3.49 Figure 3.48 graphed in the spatial domain

If the spatial and frequency domains give equivalent representations of a digital image, why do we need to transform from one to the other? The answer is that sometimes it is convenient to rearrange data so that we can access certain parts more easily. In this case, it is useful to separate out the high frequency components of the image, because this is the part that the human eye is least sensitive to. High frequency components in an image correspond to places where colors change in a small space. This kind of detail can be almost imperceptible, so eliminating it has little effect on the perceived difference between the original and the compressed image.

The DCT is a mathematical procedure that transforms image data from the spatial to the frequency domain. Chapter 2 describes this procedure in detail. It is a lossless procedure, aside from the small unavoidable error introduced through floating point arithmetic. The DCT performs the transform in one direction, and the inverse DCT can restore the original data without loss of information.

We will trace through an example to see the effects of JPEG compression. The example is based on an 8 × 8 pixel area taken from the picture shown in Figure 3.50.

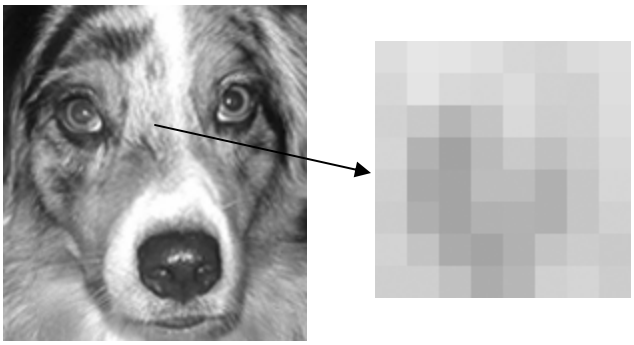


Figure 3.50 Grayscale image and enlarged 8 × 8 pixel area taken from the image

222	231	229	224	216	213	220	224
216	229	217	215	221	210	209	223
211	202	283	198	218	207	209	221
214	180	164	188	203	193	205	217
209	171	166	190	190	178	199	215
206	177	166	179	180	178	199	210
212	197	173	166	179	198	206	203
208	208	195	174	184	210	214	206

Table 3.3 Grayscale values for 8 × 8 pixel area shown in Figure 3.50

94	103	101	96	88	85	92	96
88	101	89	87	93	82	81	95
83	74	55	70	90	79	81	93
86	52	36	60	75	65	77	89
81	43	38	62	62	50	71	87
78	49	38	51	52	50	71	82
84	69	45	38	51	70	78	75
80	80	67	46	56	82	86	78

Table 3.4 Pixel values for image in Figure 3.50 shifted by –128

The pixel values are given in Table 3.3, and the values shifted by –128 are given in Table 3.4.

Table 3.5 gives the DCT values corresponding to the pixel values from Table 3.4.

585.7500	-24.5397	59.5959	21.0853	25.7500	-2.2393	-8.9907	1.8239
78.1982	12.4534	-32.6034	-19.4953	10.7193	-10.5910	-5.1086	-0.5523
57.1373	24.829	-7.5355	-13.3367	-45.0612	-10.0027	4.9142	-2.4993
-11.8655	6.9798	3.8993	-14.4061	8.5967	12.9151	-0.3122	-0.1844
5.2500	-1.7212	-1.0824	-3.2106	1.2500	9.3595	2.6131	1.1199
-5.9658	-4.0865	7.6451	13.0616	-1.1927	1.1782	-1.0733	-0.5631
-1.2074	-5.7729	-2.0858	-1.9347	1.6173	2.6671	-0.4645	0.6144
0.6362	-1.4059	-0.7191	1.6339	-0.1438	0.2755	-0.0268	-0.2255

Table 3.5 DCT of an 8 x 8 pixel area

Notice that some of the values are negative. We have said that it is possible to express a digital image as a sum of discretized sinusoidal functions, but sometimes a term in the sum must be negative. You can picture adding a negative amount of a frequency component as adding the inverted waveform.

Step 3. Quantize the frequency values.

Motivation: Quantization involves dividing each frequency coefficient by an integer and rounding off. The coefficients for high frequency components are typically small, so they often round down to 0 – which means in effect that they are "thrown away."

Details: Not every value in the matrix needs to be divided by the same integer. The amount of error introduced by the rounding is proportional to the size of the integer by which a frequency coefficient is divided. It is preferable to divide the high frequency coefficients by larger integers, since the human eye is less sensitive to high frequency components in the image. Because there is not a constant integer by which all frequency coefficients are divided, a quantization table must be stored with the compressed image. We use the divisors given in Table 3.6.

8	6	6	7	6	5	8	7
7	7	9	9	8	10	12	20
13	12	11	11	12	25	18	19
15	20	29	26	31	30	29	26
28	28	32	36	46	39	32	34
44	35	28	28	40	55	41	44
48	49	52	52	52	31	39	57
61	56	50	60	46	51	52	50

Table 3.6 Quantization table

73	-4	10	3	4	0	-1	0
11	2	-4	-2	1	-1	0	0
4	2	-1	-1	-4	0	0	0
-1	0	0	-1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 3.7 Quantized DCT values

Typically in JPEG compression, there will be a lot of zeros at the end of the matrix.

Rounding off after dividing by an integer has effectively thrown away many high frequency components. Then the strings of zeros make a good compression rate possible when run-length encoding is applied. Rounding during quantization makes JPEG a lossy compression method, but – depending on the compression rate chosen – the information that is lost usually does not unduly compromise the quality of the image.

4. Apply DPCM to the block.

Motivation: ***DPCM*** is the abbreviation for ***differential pulse code modulation***. In this context, DCPM is simply storing the difference between the the first value in the previous 8×8 block and the first value in the current block. Since the difference is generally smaller than the actual value, this step adds to the compression.

Details: DPCM is a compression technique that works by recording the difference between consecutive data values rather than the actual values. DPCM is effective in cases where consecutive values don't change very much because fewer bits are needed to record the change as opposed to the value itself. DPCM can be applied in more complex ways in other compression algorithms for digital sound and image (e.g.,

see Chapter 5), but in JPEG compression the application is simple: The upper leftmost value in an 8 × 8 block – called the DC component – is stored as the difference from the DC component in the previous block. The abbreviation DC is borrowed from electrical engineering, where it refers to *direct current*. The DC component is proportional to the average amplitude for all values in an 8 × 8 block. Since the DC value usually doesn't change much from one block to the neighboring block, it takes fewer bits to store the difference rather than the actual value.

We are considering only one block in our example, so we'll omit the DPCM step.

5. Arrange the values in a zigzag order and do run-length encoding.

Motivation: The zigzag reordering sorts the values so that they go from low frequency to high frequency components. The high frequency coefficients are grouped together at the end. If many of them round to zero after quantization, run-length encoding is even more effective.

Details: In Figure 3.51, we show the basis functions corresponding to each position in the 8 × 8 block. If you examine this picture, you'll notice that the frequencies increase from left to right in the horizontal direction and from top to bottom in the vertical direction. Thus, if we want to order the coefficients in order of increasing frequency, a good way to do this is the zigzag order pictured in Figure 3.52. The order of quantized values is now 73, -4, 11, 4, 2, 10, 3, -4, 2, -1, 0, 0, -1, -2, 4, 0, 1, -1, 0, 0, 0, 0, 0, 0, -1, -4, -1, -1, and 36 zeros. Reordering the coefficients in this way will increase the likelihood that you'll have strings of zeros, and this will increase the compression rate for the digital image.

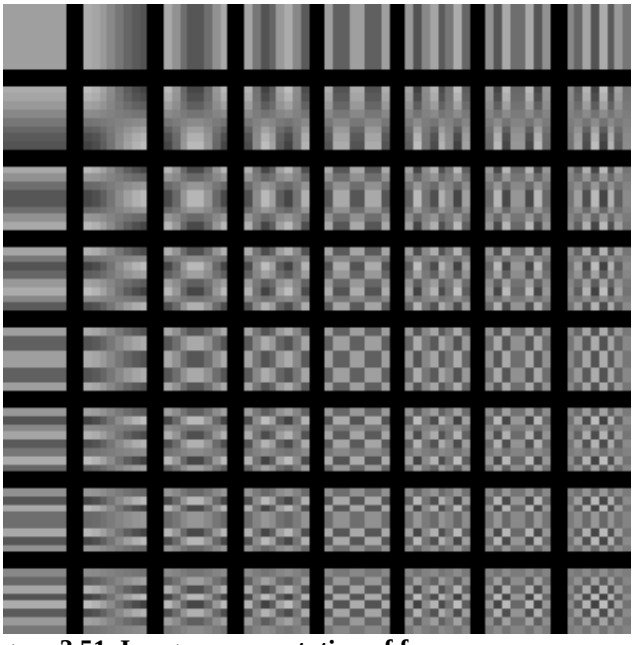


Figure 3.51 Image representation of frequency components

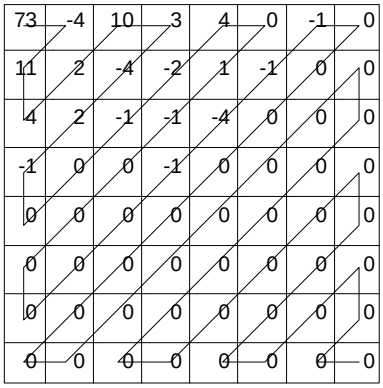


Figure 3.52 Quantized DCT values rearranged from low to high frequency components

Chapter 1 showed a simple way to do run-length encoding. An alternative method can be done using pairs of the form (skip, value) where *skip* is the number of zeros in a row and *value* is the first non-zero value after the string of zeros. A (0,0) indicates that there are no more non-zero values in the block. The run-length encoding of our example would be as follows:
(0,73), (0,-4), (0,11),(0,4),(0,2),(0,10), (0,3), (0,-4),(0,2),(0,-1),(2,-1),(0,-2),(0,4),(1,1), (0,-1),(6,-1),(0,-4),(0,-1),(0,-1),(0,0)

6. Do entropy encoding.

Motivation: Additional compression can be achieved with some kind of entropy encoding.

Details: ***Entropy encoding*** is a compression strategy whereby the length of the code for a symbol is proportional to the probability that the symbol will appear in the file. Both Huffman encoding and arithmetic encoding take this approach. See Chapter 1 and the section on Huffman encoding in this chapter for a detailed description.

After these steps have been performed, the compressed file is put into a standardized format that can be recognized by the decompressor. A header will contain global information such as the type of file, the width and height, one or more quantization tables, Huffman code tables, and an indication of any pixel-padding necessary to create properly-sized coding units. The compressed image is divided into minimum coding units (MCUs) as described above. When YCbCr color model is used, an MCU consists of a 16 × 16 macroblock of pixels that is divided into four 8 × 8 blocks of Y fields and one 8 × 8 for each of the Cb and Cr fields.

An alternative JPEG compression method called ***JPEG2000***, noted for its high compression rate and good quality, without some of the blocky artifacts of standard JPEG. This method represents digital image data as wavelets as an alternative to the DCT. Some digital imaging application programs accommodate JPEG2000, at least as a plug-in or "goodie," but it cannot become the new standard until web browsers and digital cameras support it more widely.

3.11 Vocabulary

affine transformation
alpha channel
analog gain
chrominance subsampling (chrominance downsampling)

color quantization
clipped
convolution
convolution mask
differential pulse code modulation (DPCM)
dithering
    error diffusion dithering (Floyd-Steinberg algorithm)
    noise dithering (random dithering)
    pattern dithering (ordered dithering or Bayer method)
drawing program
entropy encoding
filter
    filtering in the frequency domain
    filtering in the spatial domain
gamma value
Gaussian blur
histogram (histogram function)
Huffman encoding
image noise
indexed color
    popularity algorithm
    octree algorithm
    median-cut algorithm
    uniform partitioning algorithm
interpolation
    nearest neighbor interpolation
    bilinear interpolation
    bicubic interpolation
JFIF
JPEG compression
JPEG 2000 compression
low-pass filter
LZW compression
mean
median
metafile
mode
paint program
palette
pixel point processing
progressive download
raster graphics editor
raw image file
resampling
    downsampling (e.g., row-column deletion)
    upsampling (e.g., replication)

scale factor
scaling
spatial filtering
standard deviation
thresholding
transform
unsharp mask
variable-length encoding
web-safe colors

3.12 Exercises and Programs

1. LZW compression interactive tutorial, worksheet, and programming exercise, online
2. Huffman encoding interactive tutorial, worksheet, and programming exercise, online
3. JPEG compression interactive tutorial and worksheet, online
4. Octree algorithm for indexed color interactive tutorial, worksheet, and programming exercise, online
5. Dithering interactive tutorial, worksheet, and programming exercise, online
6. Histogram interactive tutorial and worksheet, online
7. Curves interactive tutorial and worksheet, online
8. Convolutions interactive tutorial and worksheet, online

3.13 Applications

1. Examine the specifications of your digital camera (or the one you would like to have). Is it an SLR camera? What file format does it save images in? Does it have a RAW format? Does the camera allow you to do white balancing? Does it have a software image sharpening feature? If so, is there any indication of the algorithm it uses? Does have a histogram feature?
2. Examine the specifications of your scanner (or the one you would like to have). How many physical sensors does it have? What is its advertised maximum resolution? Are any of the pixels generated through interpolation?
3. Take a photograph with a digital camera. Transport the image to your computer and open it with an image processing program. Let's say that the image is saved by your camera as a JPEG file. Use your image processing program to save the image in different file types. First, save it as a TIFF file. Are you given the option of compressing the file? Describe this. Now save it as a BMP. Are you given the option of compressing the file? Describe this. Now save it as a GIF file? What options must you deal with when you save the file as GIF? How about PNG?
4. Find a poem or short story that you would like to illustrate with a digital image (or find some other motivation for taking an interesting digital photograph). Plan to create a web-based and a printable version of this photograph. Plan your project before you begin, thinking about the aspect ratio you want for the photograph, the pixel dimensions for the original photograph, the file type you want to work with, the point at which you should crop each image pixel dimensions and resolution, the final and so forth. Experiment with the features of your image processing program to make refinements on the image and produce special effects. Keep notes on your work, and when you're done,

describe your process and the reasons for your decisions. Include a list of questions about features that you would like to use but couldn't figure out.

Examine the features of your digital image processing program, vector graphic (i.e., draw), and/or paint programs and try the exercises below with features that are available.

5. What image file types are supported by your image processing software? What file types are supported by your vector graphics program?

6. Take a photograph with a digital camera. Transport the image to your computer and open it with an image processing program. What file type is it, as created by the camera? Let's say it's JPEG (common for many digital cameras). Change the image to indexed color. Then see if your image processing program will show you the color palette. (There may be a menu selection for color palette or color table.) Reduce the color palette to just eight colors if you can. Do this without dithering. Then select the dithering option. Describe the difference in the original, the reduced-color without dithering, and the reduced-color with dithering versions. Are different dithering algorithms offered? If so, what are they?

7. Look at the filters that are offered in your image processing program, relating the numbers in the convolution mask to the effect they create. If there is a "custom filter" feature, try designing your own custom filter. Explain the numbers you put in the convolution mask, and describe what effect they create.

8. a. If you have access to Adobe Illustrator, create a very simple drawing – for example, a single line. Save the image as an AI file. Then try to open the file with a text editor to see if you can read the file as text. Try to decipher what you see.

b. Create a simple EPS file with whatever software you have available that supports this file type. See if you can read if as a text files. Try to find an example of a CGM file that can be read as text.

9. With an image processing or paint program, create an image that is just a blue circle on a white background. Make sure the edge of the circle is anti-aliased. Then make the background transparent. Save the image as a GIF file. Then insert the GIF file into another image that has a different-colored background. What is the undesirable effect that results from the anti-aliasing? What lesson is to be learned from this?

Additional exercises or applications may be found at the book or author's websites.

3.14 References

3.14.1 Print Publications

Adobe Photoshop CS2 Classroom in a Book. Berkeley, CA: Adobe Press, 2005.

Bayer, B. E., "An Optimum Method for Two-Level Rendition of Continuous-Tone Pictures," *IEEE Conference Record of the International Conference on Communications* (1973), pp. 26-11 – 26-15.

Chen, W.-H., and W. K. Pratt. "Scene Adaptive Coder." *IEEE Transactions on Communications*, COM-32: 255-232, March 1984.

Floyd, R. and Steinberg, L. "An Adaptive Algorithm for Spatial Gray Scale." *Society for Information Display 1975 Symposium Digest of Technical Papers* (1975), p. 36.

Gallagher, R. G. "Variations on a Theme by Huffman." *IEEE Transactions on Information Theory*," 24(6): 668-674, Nov. 1978.

Gervautz, M. and W. Purgathofer. "A Simple Method for Color Quantization: Octree Quantization." *Graphics Gems*, Academic Press, 1990.

Gibson, J. D., et al. *Digital Compression for Multimedia: Principles and Standards*. San Francisco: Morgan Kaufmann Publishers, 1998.

Gonzalez, Rafael C., and Richard E. Woods. *Digital Image Processing*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 2002.

Heckbert, P. "Color Image Quantization for Frame Buffer Display." *SIGGRAPH 82*, pp. 297-307.

Huffman, D. A. "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the Institute for Radio Engineers* [now IEEE], 40(9): 1098-1101, 1952.

Ifeachor, Emmanuel C., and Barrie W. Jervis. *Digital Signal Processing*. 2nd ed. Addison-Wesley Publishing, 2001.

McClelland, Deke. *Photoshop CS Bible: Professional Edition*. Indianapolis, IN: Wiley Publishing, 2004.

Pennebaker, W. B., and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. New York: Van Nostrand Reinhold, 1993.

Sayood, Khalid. *Introduction to Data Compression*, 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

Welch, T. A. "A Technique for High Performance Data Compression. *IEEE Computer*, 17(6): 8-19, June 1984.

Ziv, J., and A. Lempel. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*, 23(3): 337-343, 1977.

Ziv, J. and A. Lempel. "Compression of Individual Sequences Via Variable-Rate Coding." *IEEE Transactions of Information Theory*, 24(5): 530-536, September 1978.

3.14.2 Websites

The JPEG Committee. http://www.jpeg.org/

JPEG 2000. http://www.jpeg.org/jpeg2000/

The JPEG Standard. http://www.w3.org/Graphics/JPEG/itu-t81.pdf

See Chapter 2 for additional references on digital imaging.