arises because it is not know beforehand how long each of the character strings for each code will be. Most LZW programs handle this by taking advantage of the redundant nature of the code table. For example, look at line 29 in Table 27-3, where code 278 is defined to be *ainl*. Rather than storing these four bytes, code 278 could be stored as: *code 269 + l*, where code 269 was previously defined as *ain* in line 17. Likewise, code 269 would be stored as: *code 261 + n*, where code 261 was previously defined as *ai* in line 7. This pattern always holds: every code can be expressed as a previous code plus one new character.

The execution time of the compression algorithm is limited by searching the code table to determine if a match is present. As an analogy, imagine you want to find if a friend's name is listed in the telephone directory. The catch is, the only directory you have is arranged by telephone number, not alphabetical order. This requires you to search page after page trying to find the name you want. This inefficient situation is exactly the same as searching all 4096 codes for a match to a specific character string. The answer: organize the code table so that what you are looking for tells you where to look (like a partially alphabetized telephone directory). In other words, don't assign the 4096 codes to sequential locations in memory. Rather, divide the memory into sections based on what sequences will be stored there. For example, suppose we want to find if the sequence: *code 329 + x,* is in the code table. The code table should be organized so that the "*x*" indicates where to starting looking. There are many schemes for this type of code table management, and they can become quite complicated.

This brings up the last comment on LZW and similar compression schemes: *it is a very competitive field*. While the basics of data compression are relatively simple, the kinds of programs sold as commercial products are extremely sophisticated. Companies make money by selling you programs that perform compression, and jealously protect their trade-secrets through patents and the like. Don't expect to achieve the same level of performance as these programs in a few hours work.

JPEG (Transform Compression)

Many methods of lossy compression have been developed; however, a family of techniques called *transform compression* has proven the most valuable. The best example of transform compression is embodied in the popular JPEG standard of image encoding. JPEG is named after its origin, the *Joint Photographers Experts Group*. We will describe the operation of JPEG to illustrate how lossy compression works.

We have already discussed a simple method of lossy data compression, *coarser sampling and/or quantization* (CS&Q in Table 27-1). This involves reducing the number of bits per sample or entirely discard some of the samples. Both these procedures have the desired effect: the data file becomes smaller at the expense of signal quality. As you might expect, these simple methods do not work very well.

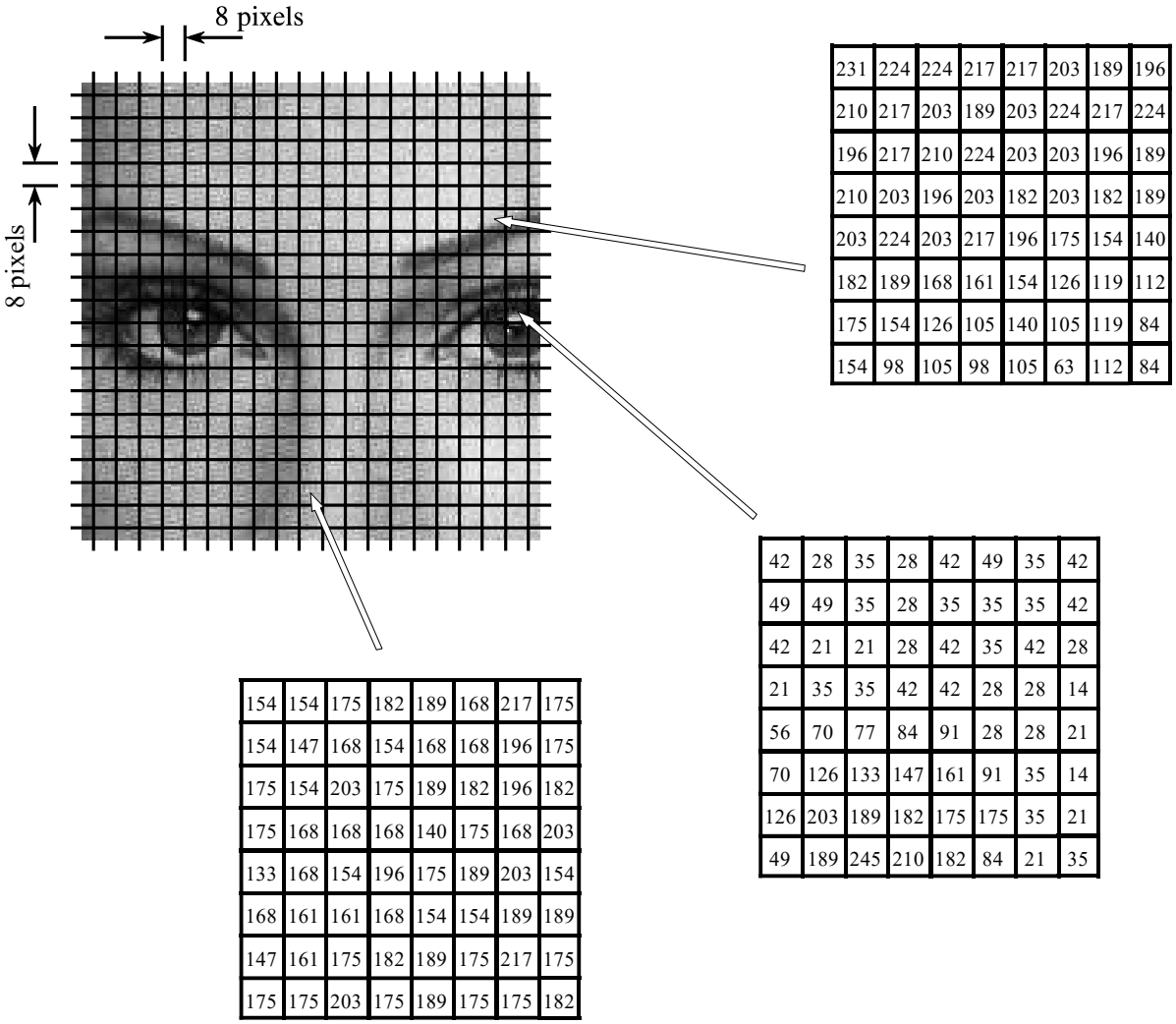


FIGURE 27-9
JPEG image division. JPEG transform compression starts by breaking the image into 8×8 groups, each containing 64 pixels. Three of these 8×8 groups are enlarged in this figure, showing the values of the individual pixels, a single byte value between 0 and 255.

Transform compression is based on a simple premise: when the signal is passed through the Fourier (or other) transform, the resulting data values will no longer be equal in their information carrying roles. In particular, the low frequency components of a signal are more important than the high frequency components. Removing 50% of the bits from the high frequency components might remove, say, only 5% of the encoded information.

As shown in Fig. 27-9, JPEG compression starts by breaking the image into 8×8 pixel groups. The full JPEG algorithm can accept a wide range of bits per pixel, including the use of color information. In this example, each pixel is a single byte, a grayscale value between 0 and 255. These 8×8 pixel groups are treated independently during compression. That is, each group is initially represented by 64 bytes. After transforming and removing data, each group is represented by, say, 2 to 20 bytes. During uncompression, the inverse

transform is taken of the 2 to 20 bytes to create an approximation of the original 8×8 group. These approximated groups are then fitted together to form the uncompressed image. Why use 8×8 pixel groups instead of, for instance, 16×16? The 8×8 grouping was based on the maximum size that integrated circuit technology could handle at the time the standard was developed. In any event, the 8×8 size works well, and it may or may not be changed in the future.

Many different transforms have been investigated for data compression, some of them invented specifically for this purpose. For instance, the *Karhunen-Loeve* transform provides the best possible compression ratio, but is difficult to implement. The *Fourier transform* is easy to use, but does not provide adequate compression. After much competition, the winner is a relative of the Fourier transform, the **Discrete Cosine Transform (DCT)**.

Just as the Fourier transform uses sine and cosine waves to represent a signal, the DCT only uses cosine waves. There are several versions of the DCT, with slight differences in their mathematics. As an example of one version, imagine a 129 point signal, running from sample 0 to sample 128. Now, make this a 256 point signal by duplicating samples 1 through 127 and adding them as samples 255 to 130. That is: $0, 1, 2, \cdots, 127, 128, 127, \cdots, 2, 1$. Taking the Fourier transform of this 256 point signal results in a frequency spectrum of 129 points, spread between 0 and 128. Since the time domain signal was forced to be symmetrical, the spectrum's imaginary part will be composed of all zeros. In other words, we started with a 129 point time domain signal, and ended with a frequency spectrum of 129 points, each the amplitude of a cosine wave. Voila, the DCT!

When the DCT is taken of an 8×8 group, it results in an 8×8 spectrum. In other words, 64 numbers are changed into 64 other numbers. All these values are *real*; there is no complex mathematics here. Just as in Fourier analysis, each value in the spectrum is the amplitude of a **basis function**. Figure 27-10 shows 6 of the 64 basis functions used in an 8×8 DCT, according to where the amplitude sits in the spectrum. The 8×8 DCT basis functions are given by:

EQUATION 27-1
DCT basis functions. The variables x & y are the indexes in the spatial domain, and u & v are the indexes in the frequency spectrum. This is for an 8×8 DCT, making all the indexes run from 0 to 7.

$$b[x,y] = \cos\left[\frac{(2x + 1)u\pi}{16}\right] \cos\left[\frac{(2y + 1)v\pi}{16}\right]$$

The low frequencies reside in the upper-left corner of the spectrum, while the high frequencies are in the lower-right. The DC component is at [0,0], the upper-left most value. The basis function for [0,1] is one-half cycle of a cosine wave in one direction, and a constant value in the other. The basis function for [1,0] is similar, just rotated by 90°.

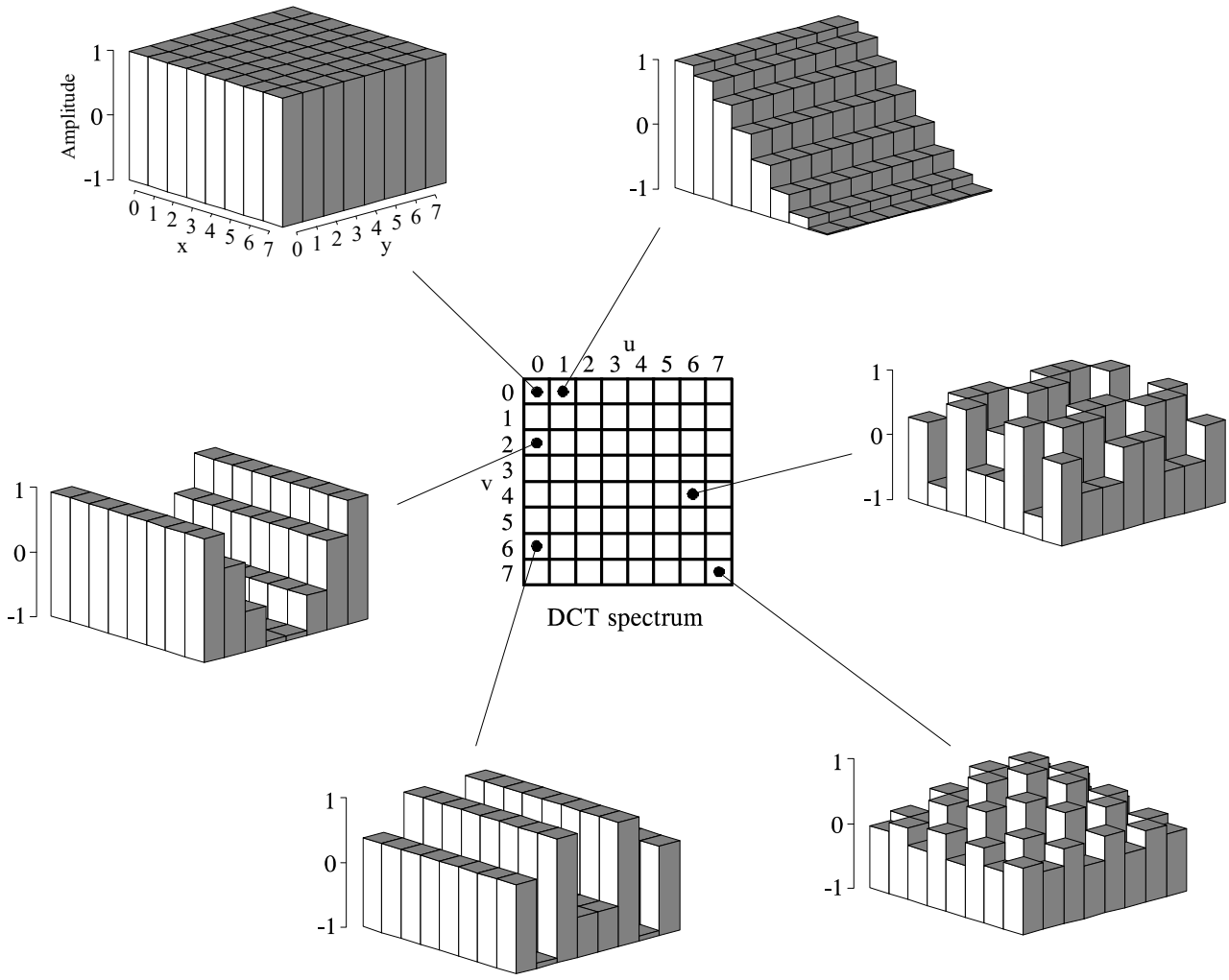


FIGURE 27-10
The DCT basis functions. The DCT spectrum consists of an 8×8 array, with each element in the array being an amplitude of one of the 64 basis functions. Six of these basis functions are shown here, referenced to where the corresponding amplitude resides.

The DCT calculates the spectrum by *correlating* the 8×8 pixel group with each of the basis functions. That is, each spectral value is found by multiplying the appropriate basis function by the 8×8 pixel group, and then summing the products. Two adjustments are then needed to finish the DCT calculation (just as with the Fourier transform). First, divide the 15 spectral values in row 0 and column 0 by *two*. Second, divide all 64 values in the spectrum by 16. The inverse DCT is calculated by assigning each of the amplitudes in the spectrum to the proper basis function, and summing to recreate the spatial domain. No extra steps are required. These are exactly the same concepts as in Fourier analysis, just with different basis functions.

Figure 27-11 illustrates JPEG encoding for the three 8×8 groups identified in Fig. 27-9. The left column, Figs. a, b & c, show the original pixel values. The center column, Figs. d, e & f, show the DCT spectra of these groups.

Original Group

a. Eyebrow

231	224	224	217	217	203	189	196
210	217	203	189	203	224	217	224
196	217	210	224	203	203	196	189
210	203	196	203	182	203	182	189
203	224	203	217	196	175	154	140
182	189	168	161	154	126	119	112
175	154	126	105	140	105	119	84
154	98	105	98	105	63	112	84

b. Eye

42	28	35	28	42	49	35	42
49	49	35	28	35	35	35	42
42	21	21	28	42	35	42	28
21	35	35	42	42	28	28	14
56	70	77	84	91	28	28	21
70	126	133	147	161	91	35	14
126	203	189	182	175	175	35	21
49	189	245	210	182	84	21	35

c. Nose

154	154	175	182	189	168	217	175
154	147	168	154	168	168	196	175
175	154	203	175	189	182	196	182
175	168	168	168	140	175	168	203
133	168	154	196	175	189	203	154
168	161	161	168	154	154	189	189
147	161	175	182	189	175	217	175
175	175	203	175	189	175	175	182

DCT Spectrum

d. Eyebrow spectrum

174	19	0	3	1	0	-3	1
52	-13	-3	-4	-4	-4	5	-8
-18	-4	8	3	3	2	0	9
5	12	-4	0	0	-5	-1	0
1	2	-2	-1	4	4	2	0
-1	2	1	3	0	0	1	1
-2	5	-5	-5	3	2	-1	-1
3	5	-7	0	0	0	-4	0

e. Eye spectrum

70	24	-28	-4	-2	-10	-1	0
-53	-35	43	13	7	13	1	3
23	9	-10	-8	-7	-6	5	-3
6	2	-2	8	2	-1	0	-1
-10	-2	-1	-12	2	1	-1	4
3	0	0	11	-4	-1	5	6
-3	-5	-5	-4	3	2	-3	5
3	0	4	5	1	2	1	0

f. Nose spectrum

174	-11	-2	-3	-3	6	-3	4
-2	-3	1	2	0	3	1	2
3	0	-4	0	0	0	-1	9
-4	-6	-2	1	-1	4	-10	-3
1	2	-2	0	0	-2	0	-5
3	-1	3	-2	2	1	1	0
3	5	2	-2	3	0	4	3
4	-3	-13	3	-4	3	-5	3

Quantization Error

g. Using 10 bits

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

h. Using 8 bits

0	-3	-1	-1	1	0	0	-1
1	0	-1	-1	0	0	0	-1
-1	-2	1	0	-2	0	-2	-2
-1	-2	-1	2	0	2	0	1
0	-2	1	0	0	1	0	0
0	-4	-1	0	1	0	0	0
0	-2	0	1	-1	-1	1	-1
-1	-3	1	1	1	-3	-2	-1

i. Using 5 bits

-13	-7	1	4	0	0	10	-2
-22	6	-13	5	-5	2	-2	-13
-9	-15	0	-17	-8	8	12	25
-9	16	1	9	1	-5	-5	13
-20	-3	-13	-16	-19	-1	-4	-22
-11	6	-8	16	-9	-3	-7	6
-14	10	-9	4	-15	3	3	-4
-13	19	12	9	18	5	-5	10

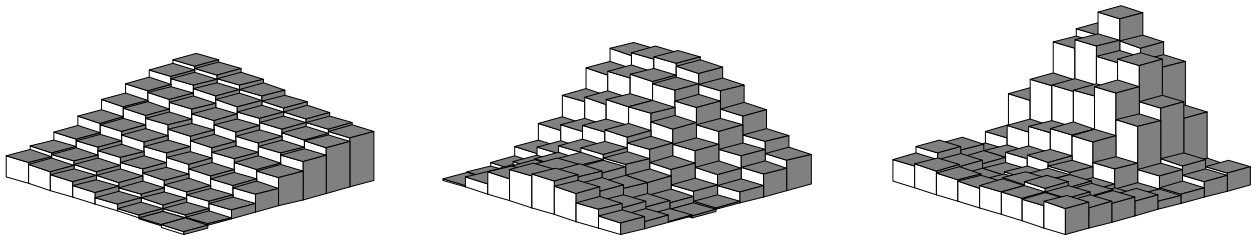
FIGURE 27-11
Example of JPEG encoding. The left column shows three 8×8 pixel groups, the same ones shown in Fig. 27-9. The center column shows the DCT spectra of these three groups. The third column shows the error in the uncompressed pixel values resulting from using a finite number of bits to represent the spectrum.

The right column, Figs. g, h & i, shows the effect of reducing the number of bits used to represent each component in the frequency spectrum. For instance, (g) is formed by truncating each of the samples in (d) to ten bits, taking the inverse DCT, and then subtracting the reconstructed image from the original. Likewise, (h) and (i) are formed by truncating each sample in the spectrum to eight and five bits, respectively. As expected, the error in the reconstruction

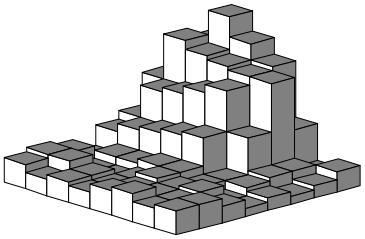
increases as fewer bits are used to represent the data. As an example of this bit truncation, the spectra shown in the center column are represented with 8 bits per spectral value, arranged as 0 to 255 for the DC component, and -127 to 127 for the other values.

The second method of compressing the frequency domain is to discard some of the 64 spectral values. As shown by the spectra in Fig. 27-11, nearly all of the signal is contained in the low frequency components. This means the highest frequency components can be eliminated, while only degrading the signal a small amount. Figure 27-12 shows an example of the image distortion that occurs when various numbers of the high frequency components are deleted. The 8×8 group used in this example is the *eye* image of Fig. 27-10. Figure (d) shows the correct reconstruction using all 64 spectral values. The remaining figures show the reconstruction using the indicated number of lowest frequency coefficients. As illustrated in (c), even removing three-fourths of the highest frequency components produces little error in the reconstruction. Even better, the error that does occur looks very much like random noise.

JPEG is good example of how several data compression schemes can be combined for greater effectiveness. The entire JPEG procedure is outlined in the following steps. First, the image is broken into the 8×8 groups. Second, the DCT is taken of each group. Third, each 8×8 spectrum is compressed by the above methods: reducing the number of bits and eliminating some of the components. This takes place in a single step, controlled by a **quantization table**. Two examples of quantization tables are shown in Fig. 27-13. Each value in the spectrum is divided by the matching value in the quantization table, and the result rounded to the nearest integer. For instance, the upper-left value of the quantization table is *one*,



a. 3 coefficients b. 6 coefficients c. 15 coefficients



d. 64 coefficients (correct image)

FIGURE 27-12
Example of JPEG reconstruction. The 8×8 pixel group used in this example is the *eye* in Fig. 27-9. As shown, less than 1/4 of the 64 values are needed to achieve a good approximation to the correct image.

a. Low compression

1	1	1	1	1	2	2	4
1	1	1	1	1	2	2	4
1	1	1	1	2	2	2	4
1	1	1	1	2	2	4	8
1	1	2	2	2	2	4	8
2	2	2	2	2	4	8	8
2	2	2	4	4	8	8	16
4	4	4	4	8	8	16	16

b. High compression

1	2	4	8	16	32	64	128
2	4	4	8	16	32	64	128
4	4	8	16	32	64	128	128
8	8	16	32	64	128	128	256
16	16	32	64	128	128	256	256
32	32	64	128	128	256	256	256
64	64	128	128	256	256	256	256
128	128	128	256	256	256	256	256

FIGURE 27-13
JPEG quantization tables. These are two example quantization tables that might be used during compression. Each value in the DCT spectrum is divided by the corresponding value in the quantization table, and the result rounded to the nearest integer.

resulting in the DC value being left unchanged. In comparison, the lower-right entry in (a) is 16, meaning that the original range of -127 to 127 is reduced to only -7 to 7. In other words, the value has been reduced in precision from eight bits to four bits. In a more extreme case, the lower-right entry in (b) is 256, completely eliminating the spectral value.

In the fourth step of JPEG encoding, the modified spectrum is converted from an 8×8 array into a linear sequence. The serpentine pattern shown in Figure 27-14 is used for this step, placing all of the high frequency components together at the end of the linear sequence. This groups the *zeros* from the eliminated components into long runs. The fifth step compresses these runs of zeros by run-length encoding. In the sixth step, the sequence is encoded by either Huffman or arithmetic encoding to form the final compressed file.

The amount of compression, and the resulting loss of image quality, can be selected when the JPEG compression program is run. Figure 27-15 shows the type of image distortion resulting from high compression ratios. With the 45:1 compression ratio shown, each of the 8×8 groups is represented by only about 12 bits. Close inspection of this image shows that six of the lowest frequency basis functions are represented to some degree.

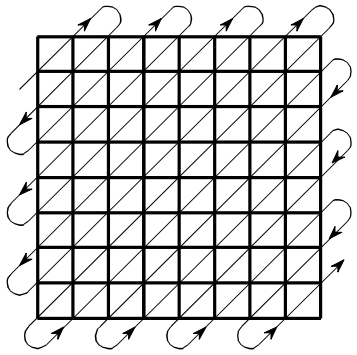


FIGURE 27-14
JPEG serial conversion. A serpentine pattern used to convert the 8×8 DCT spectrum into a linear sequence of 64 values. This places all of the high frequency components together, where the large number of zeros can be efficiently compressed with run-length encoding.

a. Original image

b. With 10:1 compression

c. With 45:1 compression

FIGURE 27-15
Example of JPEG distortion. Figure (a) shows the original image, while (b) and (c) shows restored images using compression ratios of 10:1 and 45:1, respectively. The high compression ratio used in (c) results in each 8×8 pixel group being represented by less than 12 bits.

Why is the DCT better than the Fourier transform for image compression? The main reason is that the DCT has one-half cycle basis functions, i.e., S[0,1] and S[1,0]. As shown in Fig. 27-10, these gently slope from one side of the array to the other. In comparison, the lowest frequencies in the Fourier transform form *one complete cycle*. Images nearly always contain regions where the brightness is gradually changing over a region. Using a basis function that matches this basic pattern allows for better compression.

MPEG

MPEG is a compression standard for digital video sequences, such as used in computer video and digital television networks. In addition, MPEG also provides for the compression of the sound track associated with the video. The name comes from its originating organization, the *Moving Pictures Experts Group*. If you think JPEG is complicated, MPEG is a nightmare! MPEG is something you buy, not try to write yourself. The future of this technology is

to encode the compression and uncompression algorithms directly into integrated circuits. The potential of MPEG is vast. Think of thousands of video channels being carried on a single optical fiber running into your home. This is a key technology of the 21st century.

In addition to reducing the data rate, MPEG has several important features. The movie can be played *forward* or in *reverse*, and at either *normal* or *fast* speed. The encoded information is *random access*, that is, any individual frame in the sequence can be easily displayed as a still picture. This goes along with making the movie *editable*, meaning that short segments from the movie can be encoded only with reference to themselves, not the entire sequence. MPEG is designed to be robust to errors. The last thing you want is for a single bit error to cause a disruption of the movie.

The approach used by MPEG can be divided into two types of compression: *within-the-frame* and *between-frame*. Within-the-frame compression means that individual frames making up the video sequence are encoded as if they were ordinary still images. This compression is preformed using the JPEG standard, with just a few variations. In MPEG terminology, a frame that has been encoded in this way is called an intra-coded or **I-picture**.

Most of the pixels in a video sequence change very little from one frame to the next. Unless the camera is moving, most of the image is composed of a background that remains constant over dozens of frames. MPEG takes advantage of this with a sophisticated form of *delta encoding* to compress the redundant information *between frames*. After compressing one of the frames as an I-picture, MPEG encodes successive frames as predictive-coded or **P-pictures**. That is, only the pixels that have changed since the I-picture are included in the P-picture.

While these two compression schemes form the backbone of MPEG, the actual implementation is immensely more sophisticated than described here. For example, a P-picture can be referenced to an I-picture that has been *shifted*, accounting for motion of objects in the image sequence. There are also bidirectional predictive-coded or **B-pictures**. These are referenced to both a previous and a future I-picture. This handles regions in the image that gradually change over many of frames. The individual frames can also be stored out-of-order in the compressed data to facilitate the proper sequencing of the I, P, and B-pictures. The addition of color and sound makes this all the more complicated.

The main distortion associated with MPEG occurs when large sections of the image change quickly. In effect, a burst of information is needed to keep up with the rapidly changing scenes. If the data rate is fixed, the viewer notices "blocky" patterns when changing from one scene to the next. This can be minimized in networks that transmit multiple video channels simultaneously, such as cable television. The sudden burst of information needed to support a rapidly changing scene in one video channel, is averaged with the modest requirements of the relatively static scenes in the other channels.