An Introduction to Video Compression in C/C++

Fore June

Chapter 1

Quantization and Run-level Encoding

7.1 Introduction

In Chapter 6, we have discussed using DCT to transform image data to the frequency domain. We have seen that DCT alone does not achieve any data compression and does not lose any information. However, the transformation usually clusters the data that allow us to carry out compression in the next stage effectively. Typically, the 'low frequency' components of the DCT coefficients of a block of image position around the DC (0,0) coefficient. As shown in the data output of Section 6.6 of Chapter 6, the nonzero DCT coefficients are clustered around the top-left (DC) coefficient and the distribution is roughly symmetrical in the horizontal and vertical directions. This special characteristics inspire people to reorder the DCT coefficients so that more consecutive zeros are lined up together and thus are easier to encode.

After DCT, the next operation in our compression pipeline involves quantization which will generate even more zeros in a DCT block as small values are approximated by a zeros. After quantization, we shall reorder the data so that they can be encoded effectively using a technique called run-level encoding. The run-level values are then encoded using entropy encoding, which can be Huffman encoding or arithmetic encoding. In this book, we only discuss Huffman encoding, which is a lot faster than arithmetic encoding though the later may yield slightly lower compression ratio. Entropy encoding generates a bit stream, which is ready for transmission or storage. Figure 7-1 summarizes these encoding steps, which extend the steps shown in Figure 6-5.

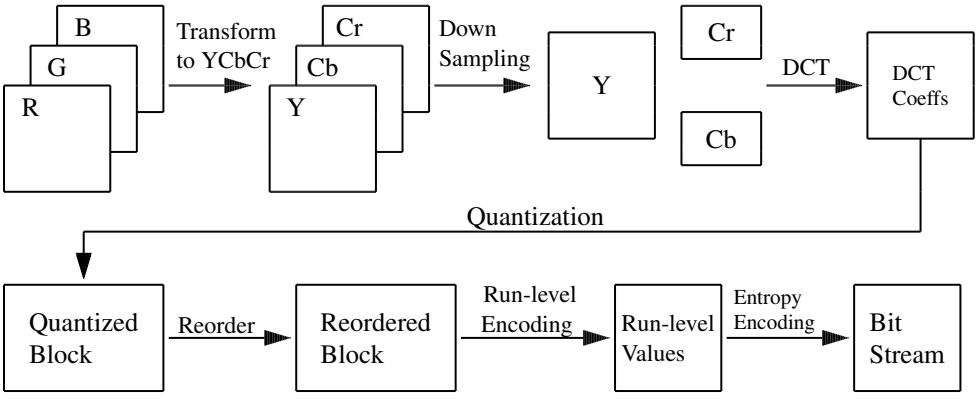

Figure 7-1. Encoding of RGB Image Block

Of all the stages shown in Figure 7-1, only the operations "Down Sampling" and "Quantization" are irreversible. Other stages are reversible; no information is lost in the operation. In particular, no rounding error will be introduced in the operations, "Reorder", "Run-level Encoding", and "Entropy Encoding"; the original data before each operation can be recovered exactly.

Compression occurs in stages "Down Sampling", "Run-level Encoding" and "Entropy Encoding". On the other hand, operations "DCT", "Quantization", and "Reorder" set up the data for these stages to compress them efficiently.

7.2 Quantization

We discussed in Chapter 3 that *quantization* is the procedure of constraining the value of a function at a sampling point to a predetermined finite set of discrete values. A *quantizer* maps a range of values X to a reduced range of values F. Therefore, a quantized signal can be represented by fewer bits than the original signal as the range of quantized values is smaller. To achieve high compression, we do not want to retain the full range of DCT coefficients as we did in Chapter 6 where we have used 16-bit (i.e. data type short) to save a DCT coefficient. In this Chapter, we discuss how to quantize DCT coefficients and represent a coefficient with significantly less bits.

Quantization can be done using a *scalar quantizer* or a *vector quantizer*. A *scaler quantizer* maps one sample of the input signal to one quantized output value. It is a special case of a *vector quantizer*, which maps a group of input samples (a 'vector') to an index of a codebook that contains vectors (groups) of quantized values. A vector quantizer in general yields better results but consumes a lot more computing power.

7.2.1 Scalar Quantization

An example of a simple *scalar quantizer* is an operation that rounds a real number to an integer. Obviously, the operation is a many-to-one mapping and is irreversible. Information is lost in the process; we cannot determine the exact value of the original real number from the rounded integer.

A more general example of scalar quantization is a uniform quantizer where an input value X is divided by a **quantization parameter** (or step size) q and rounded to the nearest integer F_q as shown in Equation (7.1) below:

$$F_q = round(\frac{X}{q}) \tag{7.1}$$

The quantized output level is given by

$$Y = F_q \times q \tag{7.2}$$

The output levels Y are spaced uniformly with step size q. The following example shows a uniform quantizer with various step sizes.

Example 7-1 A uniform quantizer with step sizes, 1, 2, 3, 5, and 8.

X	Y q = 1	Y q = 2	Y q = 3	Y q = 5	Y q = 8
-5	-5	-6	-6	-5	-8
-4	-4	-4	-3	-5	-8
-3	-3	-4	-3	-5	0
-2	-2	-2	-3	0	0
-1	-1	-2	0	0	0
0	0	0	0	0	0
1	1	2	0	0	0
2	2	2	3	0	0
3	3	4	3	5	0
4	4	4	3	5	8
5	5	6	6	5	8
6	6	6	6	5	8
7	7	8	6	5	8
8	8	8	9	10	8
9	9	10	9	10	8
10	10	10	9	10	8
11	11	12	12	10	8
12	12	12	12	10	16

Figure 7-2 shows two examples of scalar quantizer. The linear scalar quantizer shown on the left shows linear mapping between input and output values. The nonlinear quantizer on the right shows a dead zone where small input values are mapped to zero.

More precisely, we can define an N-point scalar quantizer Q as a mapping $Q : R \rightarrow C$ where R is the real line and

$$C \equiv \{y_1, y_2, ..., y_N\} \subset R \tag{7.3}$$

is the output set or codebook with size $|C| = N$. The output values, y_i, are referred to as output levels, output points, or reproduction values. Quite often, we choose the indexing of output values so that

$$y_1 < y_2 < ... < y_N \tag{7.4}$$

The resolution or code rate, r, of a scalar quantizer is defined as $r = log_2 N$, which measures the number of bits required to uniquely specify the quantized value.

Every quantizer can be viewed as making up of two successive operations (mappings), an encoder, E, (or forward quantizer FQ), and a decoder, D (or inverse quantizer IQ). The encoder E is a mapping

$$E : R \rightarrow I \tag{7.5}$$

where $I = \{1, 2, 3, ..., N\}$, and the decoder is the mapping

$$D : I \rightarrow C. \tag{7.6}$$

Therefore, if $Q(x) = y_i$, then $E(x) = i$ and $D(i) = y_i$. Consequently, $Q(x) = D(E(x))$. Note that the decoder can be implemented by a table-lookup process, where the table or codebook contains the output set, which can be stored with very high precision without affecting the

transmission rate R. The decoder is also referred to as inverse quantization and the encoder is sometimes referred to as forward quantization.

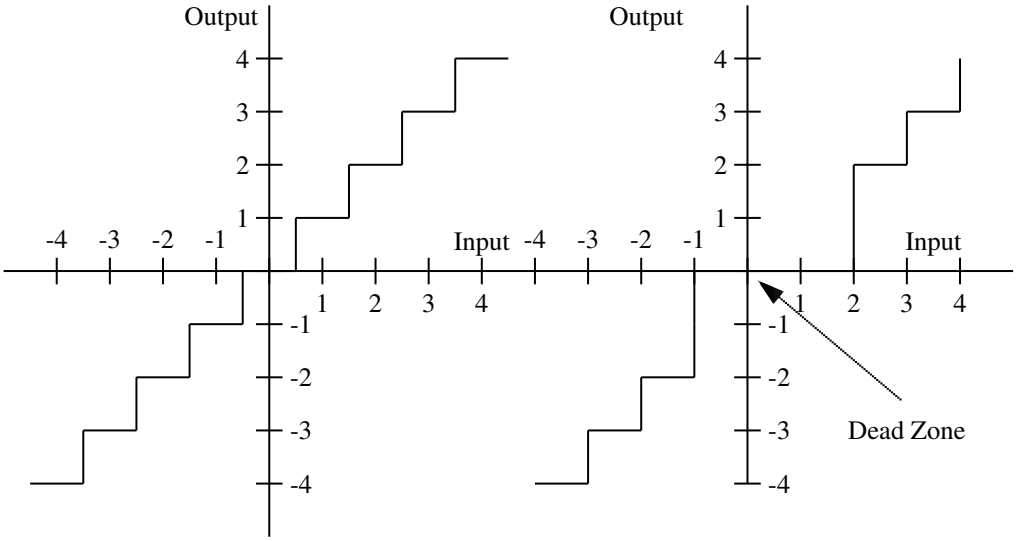

Figure 7-2. Linear and Non-linear Quantizers

7.2.2 Vector Quantization

Vector quantization (VQ) is a generalization of scalar quantization to the quantization of a vector, an ordered set of real numbers. Speech or image samples can be grouped together to form a vector. Thus vector quantization can be regarded as a form of pattern recognition where an input pattern is "approximated" by one of a predetermined set of patterns stored in a codebook.

We can define a vector quantizer Q of dimension k and size N as a mapping from a vector in k-dimensional Euclidean space, R^k, into a finite set C that contains N output or reconstructed vectors, called *code vectors* or *codewords*. That is,

$$Q: \ R^k \rightarrow C, \tag{7.7}$$

where

$$\begin{aligned} C &= \{y_1, y_2, ..., y_N\} \\ y_i &\in R^k \\ i &\in I \equiv \{1, 2, ..., N\} \end{aligned} \tag{7.8}$$

The set C is referred to as the *codebook* or the *code* with N distinct elements, each a vector in R^k. The *resolution* or *code rate* r of the vector quantizer is given by:

$$code\ rate\ r = \frac{log_2 N}{k} \tag{7.9}$$

In general, the code rate of a vector quantizer is the number of bits per vector component used to represent the input vector, indicating the accuracy or precision it can achieve with the quantizer. Note that for a given dimension k, the resolution is determined by the size N of the codebook but not by the number of bits used to specify the code vectors stored in the codebook. Even if we specify a code vector to a very high precision, we still can have a very low resolution by using

a small codebook. Typically, a codebook is loaded as a lookup table in the main memory of a computer and the number of bits used in each table entry does not affect the quantizer's resolution or bit rate.

Associated with each of the N vectors in C is a partition of R^k into N regions or *cells*, R_i for $i \in I$:

$$R_i = \{\mathbf{x} \in R^k : Q(\mathbf{x}) = y_i\} \tag{7.10}$$

The ith cell R_i given by (7.10) is called the *inverse image* or *pre-image* of y_i under the mapping Q and can be denoted by:

$$R_i = Q^{-1}(y_i) \tag{7.11}$$

The application of VQ to image compression can be summarized as follows:

1. Partition an image into blocks of pixels.
2. Choose a vector from the codebook that best-approximates the current block.
3. Send the index pointing to the chosen vector to the decoder.
4. At the decoder, reconstruct an approximate copy of the original block using the chosen vector.

Figure 7-3 shows the concept of vector quantization.

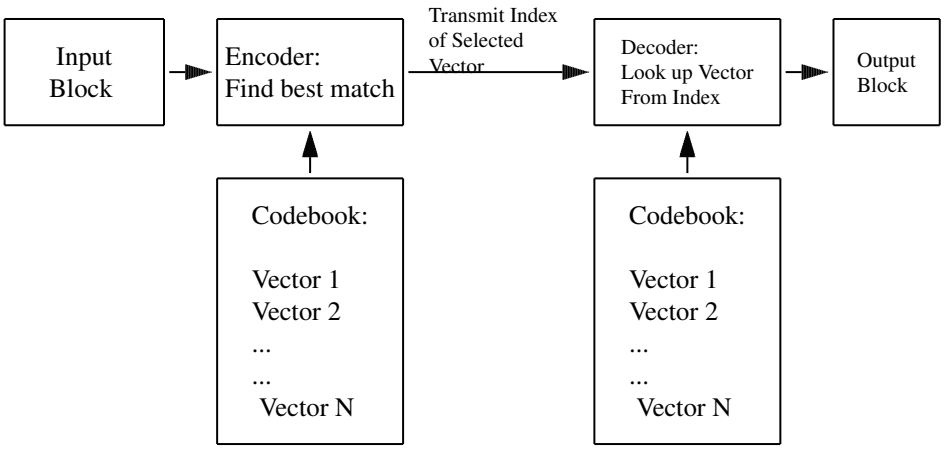

Figure 7-3. Vector Quantization

7.2.3 MPEG-4 Quantization

Video compression standard MPEG-4 allows two methods to quantize DCT coefficients. A parameter called quantizer_scale (quantization_parameter) is used to control how much information is discarded during the quantization process. The parameter can take values from 1 to 31 in the case of 8-bit textures and 1 to $2^{quant\_precision} - 1$ in the case of non 8-bit textures. Each frame may use a different value of quantizer_scale. The two methods are referred to as Method 2 (basic method) and Method 1 (more flexible but more complex). Method 2, which is the default method, specifies the quantization of the DC component, $F[0][0]$ using a fixed quantizer step:

$$\begin{aligned} Forward\ Quantization : F_q[0][0] = & \quad \frac{F[0][0]}{dc\_scalar} \\ Inverse\ Quantization : F'[0][0] = & \quad F_q[0][0] \times dc\_scalar \end{aligned} \tag{7.12}$$

where dc_scalar which has a value of 8 in the short header mode and depends on the quantizer_parameter is determined from the following table:

$quantizer\_parameter(Q_p)$	1 - 4	5 - 8	9 - 24	25 - 31
$dc\_scalar(luminance)$	8	$2Q_p$	$Q_p + 8$	$2Q_p - 16$
$dc\_scalar(chrominance)$	8	$\frac{2Q_p + 13}{2}$	$\frac{Q_p + 13}{2}$	$Q_p - 6$

Table 7-1: MPEG-4 Quantization Parameters.

All other coefficients are rescaled as follows.

$$\begin{aligned} |F| = & \quad Q_p \times (2 \times |F_Q| + 1) && \text{if } Q_p \text{ is odd and } F_Q \neq 0 \\ |F| = & \quad Q_p \times (2 \times |F_Q| + 1) - 1 && \text{if } Q_p \text{ is even and } F_Q \neq 0 \\ |F| = & \quad 0 && \text{if } F_Q = 0 \end{aligned} \tag{7.13}$$

where F_Q is the forward-quantized coefficient and F is the rescaled (inverse-quantized) coefficient.

In Method 1, which is also referred to as alternate quantizer, MPEG-4 uses a weighting factor to exploit properties of the human visual system (HVS). Since human eyes are less sensitive to some frequencies, we can quantize these frequencies with a coarser step-size, which results in a more compactly coded bit-stream and minimizes the image distortion. MPEG-4 recommends different weight matrices for the quantization of various sample blocks. The forward and inverse quantization can be described as follows.

Forward Quantization is described by the following equation (we shall explain the meaning of intra and inter blocks in a later chapter).

$$F_Q(u,v) = \frac{16F(u,v)}{2Q_p(W(u,v) - k \times Q_p)} \tag{7.14}$$

where

$$k = \begin{cases} 0 & \text{for intra coded blocks} \\ sign(F_Q(u,v)) & \text{for inter coded blocks} \end{cases}$$

and

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ +1 & \text{otherwise} \end{cases}$$

Inverse Quantization is described by Eqaution (7.15) shown below.

$$F'(u,v) = \begin{cases} 0 & \text{if } F_Q(u,v) = 0 \\ \dfrac{(2F_Q(u,v)+k) \times W(u,v) \times Q_p}{16} & \text{if } F_Q(u,v) \neq 0 \end{cases} \tag{7.15}$$

Users can define the weighting factors $W(u,v)$ based on their particular applications. MPEG-4 suggests some default weighting factors, which are shown in Table 7-2, where the left table (Table 7-2a) shows the default weighting matrix for intra-coded macroblocks and the right one presents the default weighting matrix for inter-coded macroblocks. Again, we shall explain the difference between intra-coded and inter-coded blocks in a later chapter. The tables assume that a macroblock is of size 8×8. For example, if intra-coded blocks are used, $W(0,0) = 8$, and $W(7,7) = 45$. On the other hand, when inter-coded blocks are used, $W(0,0) = 16$ and $W(7,7) = 33$.

Table 7-2a Intra Block Weights $W(u,v)$

$u\backslash v$	0	1	2	3	4	5	6	7
0	8	17	18	19	21	23	25	27
1	17	18	19	21	23	25	27	28
2	20	21	22	23	24	26	28	30
3	21	22	23	24	26	28	30	32
4	22	23	24	26	28	30	32	35
5	23	24	26	28	30	32	35	38
6	25	26	28	30	32	35	38	41
7	27	28	30	32	35	38	41	45

Table 7-2b Inter Block Weights $W(u,v)$

$u\backslash v$	0	1	2	3	4	5	6	7
0	16	17	18	19	20	21	22	23
1	17	18	19	20	21	22	23	24
2	18	19	20	21	22	23	24	25
3	19	20	21	22	23	24	26	27
4	20	21	22	23	25	26	27	28
5	21	22	23	24	26	27	28	30
6	22	23	24	26	27	28	30	31
7	23	24	25	27	28	30	31	33

Listing 7-1 presents an implementation of a uniform quantizer. The implementation is simple and straightforward; the array *coef[]* holds an 8×8 sample block of DCT coefficients; *Qstep* is the quantization parameter discussed above; its default value is set to 12. The code shows both forward quantization (FQ) and inverse quantization (IQ).

Program Listing 7-1: Implementation of Uniform Quantizer

```
-------------------------------------------------------------------------
void quantize_block ( short coef[8][8] )
{
  for ( int i = 0; i < 8; i++ )
    for ( int j = 0; j < 8; j++ )
      coef[i][j] = ( short ) round ( (double)coef[i][j] / Qstep );
}

//inverse quantize one block
void inverse_quantize_block ( short coef[8][8] )
{
  for ( int i = 0; i < 8; i++ )
    for ( int j = 0; j < 8; j++ )
      coef[i][j] = (short) (  coef[i][j] * Qstep );
}
-------------------------------------------------------------------------
```

7.3 Reordering

After DCT transform and forward quantization, a sample block may have only a few nonzero coefficients and all others are zeros. It is desirable to group the zero coefficients together so that they can be represented effectively. The optimum reordering path (scan order) depends on the distribution of nonzero DCT coefficients. A commonly used scan order is a zigzag path starting from the DC coefficient at the top left corner of an 8×8 sample block as shown in Figure 7-4. After such a reordering, nonzero coefficients tend to cluster together at the beginning of the reordered array, followed by long sequences (runs) of zeros. Data consist of long runs of certain values can be efficiently encoded using a run-level coding technique that we shall discuss in the next section.

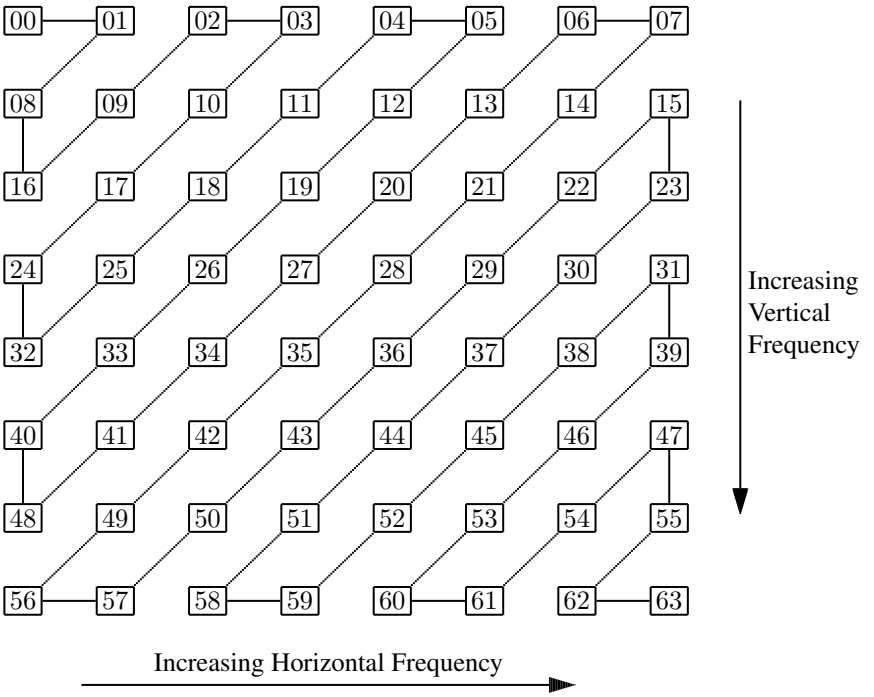

Figure 7-4. Zigzag scan order

Figure 7-4 shows that sample 0 is the first element to be read in a zigzag scan, followed by sample 1, then by sample 8, sample 16, sample 9, and so on. After a zigzag scan, the indices of the original DCT coefficients are reordered as below.

0	1	8	16	9	2	3	10
17	24	32	25	18	11	4	5
12	19	26	33	40	48	41	34
27	20	13	6	7	14	21	28
35	42	49	56	57	50	43	36
29	22	15	23	30	37	44	51
58	59	52	45	38	31	39	46
53	60	61	54	47	55	62	63

Researchers have explored and tried various scanning orders but the zigzag scan remains the most commonly used scan in video compression. However, for some applications such as a field block where the coefficient distribution is often skewed, an alternate scan is more effective. In an alternate scan, the left-hand coefficients are scanned before those on the right side as shown in Figure 7-5.

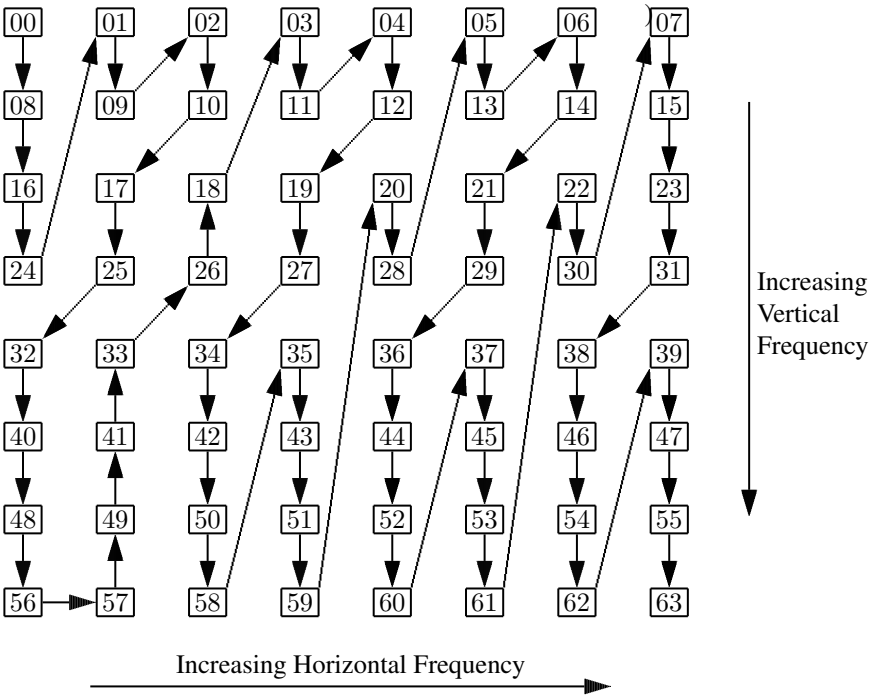

Figure 7-5. Alternate scan order

In our subsequent discussions, we shall only use the zigzag scan. Listing 7-2 shows the implementations of such a reordering and the reverse of it. The code assumes that 64 samples are arranged in an 8×8 sample block. The one dimensional index k is represented as a two-dimensional pair of integers (i, j) with $i = k/8, j = k\%8$. For example, 1 is represented by $(0, 1)$, 8 by $(1, 0)$, 23 by $(2, 7)$, and so on. The function **reorder()** uses zigzag scan to rearrange the 64 sample elements stored in the *Y*[][] array and save the reordered samples in the array *Yr*[][]. The function **reverse_reorder()** does the opposite; it restores the original order from array *Yr*[][] and saves the results in array *Y*[][].

Program Listing 7-2: Reordering

```
%\begin{verbbox}
\begin{verbatim}
int zigzag[] = {
    0,  1,  8, 16,  9,  2,  3, 10,
    17, 24, 32, 25, 18, 11,  4,  5,
    12, 19, 26, 33, 40, 48, 41, 34,
    27, 20, 13,  6,  7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36,
    29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46,
```

```
    53, 60, 61, 54, 47, 55, 62, 63
};

//input : Y,  output : Yr
void reorder ( short Y[8][8], short Yr[8][8] )
{
  int k, i1, j1;
  k = 0;
  for ( int i = 0; i < 8; i++ ){
    for ( int j = 0; j < 8; j++ ){
      i1 =  zigzag[k] / 8;
      j1 = zigzag[k] % 8;
      Yr[i][j] = Y[i1][j1];
      k++;
    }
  }
}

//input : Yr, output : Y
void reverse_reorder ( short Yr[8][8], short Y[8][8] )
{
  int k, i1, j1;
  k = 0;
  for ( int i = 0; i < 8; i++ ){
    for ( int j = 0; j < 8; j++ ){
      i1 = zigzag[k] / 8;
      j1 = zigzag[k] % 8;
      Y[i1][j1] = Yr[i][j];
      k++;
    }
  }
}
-------------------------------------------------------------------------
```

7.4 Run-Level Encoding

After DCT, forward quantization and reordering, we may obtain long sequences of zeros followed by nonzero values. One of the effective methods to encode this kind of data is the three-dimensional (3D) run-level encoding. A 3D run-level codeword is represented by a tuple (*run, level, last*) where *run* is the number of zeros preceding a nonzero coefficient, *level* is the value of the nonzero coefficient, and *last* indicates whether the codeword is the final one with nonzero coefficient in the block. The following shows two examples of 3D run-level encoding.

Input Sequence: $1, 0, -2, 3, 0, 0, 0, 4, 5, 0, -1, 6, 0, 0, 0, ..., 0$

Output: $(0, 1, 0), (1, -2, 0), (0, 3, 0), (3, 4, 0), (0, 5, 0), (1, -1, 0), (0, 6, 1)$

Input Sequence: $0, 0, 2, 0, 0, 0, 0, 1, 0, 0, -2, 0, 7, 0, 0, 0, ..., 0$

Output: $(2, 2, 0), (4, 1, 0), (2, -2, 0), (1, 7, 1)$

We have to handle the special case when the whole block contains zeros separately; we code the whole block of zeros by (64, 0, 1):

Input Sequence: $0, 0, 0, ..., 0$

Output: $(64, 0, 1)$

Implementation of run-level encoding can be done by defining a class *run3D* comprising public members *run*, *level*, and *last* as follows. A *run3D* object can hold one run-level codeword.

```
class Run3D {
public:
  unsigned char run;
  short level;
  char last;
};
```

Suppose a macroblock of 8×8 DCT coefficients have been quantized, zigzag-reordered, and saved in an array *Y*[]. Listing 7-3 presents a piece of code that can run-level-encode such a block of 64 coefficients. The function **run_block()** accepts the 8×8 block of coefficients saved in the array *Y*[] as input; the outputs are the run-level codewords returned in the run3D object array *runs*[]. Each run3D object holds the information of the tuple (*run, level, last*) that represents a codeword.

Program Listing 7-3: Run-level Encoding

```
-------------------------------------------------------------------------
/*
    Input: 64 quantized DCT coefficients in Y[][].
    Output: 3D run-level codewords in runs[].
*/
void run_block ( short Y[8][8], Run3D runs[] )
{
    unsigned char run_length = 0, k = 0;
    for ( int i = 0; i < 8; i++ ) {
      for ( int j = 0; j < 8; j++ ) {
        if ( Y[i][j] == 0  ) {
          run_length++;
          continue;
        }
        runs[k].run = run_length;
        runs[k].level = Y[i][j];
        runs[k].last = 0;
        run_length = 0;
        k++;
      }
    }
    if ( k > 0 )
      runs[k-1].last = 1;       //last nonzero element
    else {                      //whole block 0
      runs[0].run = 64;
      runs[0].level = 0;
      runs[0].last = 1;         //this needs to be 1 to terminate
    }
}
-------------------------------------------------------------------------
```

The corresponding code that recovers the block of 64 DCT coefficients from the run-level codewords is presented in Listing 7-4; the code first recovers the zeros and nonzero values from the run-level codewords saved in *runs*[] until it finds the *last* field of the codeword is 1; after detecting the *last* field to be 1, it sets the remaining values of *Y*[] to zero.

Program Listing 7-4: Run-level Decoding

```
-------------------------------------------------------------------------
/*
 *   Input: 3D run-level codewords of a macroblock in runs[].
 *   Output: 64 DCT coefficients in Y[][].
 */
void Run::run_decode ( Run3D runs[], short Y[8][8] )
{
    int i, j, r, k = 0, n = 0;

    while ( n < 64 ) {
      for ( r = 0;  r < runs[k].run; r++ ){
        i = n / 8;
        j = n % 8;
        Y[i][j] = 0;
        n++;
      }
      if ( n < 64 ){
        i = n / 8;
        j = n % 8;
        Y[i][j] = runs[k].level;
        n++;
      }
      if ( runs[k].last != 0 ) break;
      k++;
    }

    //run of 0s to end
    while ( n < 64 ) {
      i = n / 8;
      j = n % 8;
      Y[i][j] = 0;
      n++;
    }
}
-------------------------------------------------------------------------
```

Listing 7-5 presents the program **test_run.cpp**, which is a complete program that demonstrates the operations of quantization, reordering and run-level encoding and the reverse of the operations. It reads DCT coefficients from the file specified by argv[0] which has saved blocks of 8×8 DCT coefficients in short (16-bit) form as discussed in Chapter 6; the file of argv[0] can be obtained from the PPM image file "beach.ppm" after the operations of 4:2:0 YCbCr down sampling and DCT Transformation that we have discussed in Chapter 6. The functions **get64, print_blcock,** and **print_run** are member functions of the class **Run**; **get64** gets 64 sample values from the the specified input file and put them in the two-dimensional array *Y*[][], which will be returned to the calling function; **print_block** prints one 8×8 sample block; **print_run** prints the run-level codewords on one 64-sample block. We have grouped related functions into classes named **Quantizer**, **Reorder**, **Run**, and **Printer**.

Program Listing 7-5: Quantization, Reordering, and Run-level Encoding

```
-------------------------------------------------------------------------
/*
  test_run.cpp
  A demo program that illustrates the concepts of quantization, zigzag
  reordering and run-level encoding.  It reads DCT coefficients from the
  file specified by argv[0] which has saved 8x8 blocks of DCT
  coefficients in short ( 16-bit ) form.
*/
#include <stdio.h>
#include <string.h>
#include "run3D.h"
#include "run.h"
#include "reorder.h"
#include "quantizer.h"
#include "../util/printer.h"

using namespace std;

//function only used in this file
static int read_dct_header ( FILE *fp )
{
  char header[] =  { 'D', 'C', 'T', '4', ':', '2', ':', '0' };
  int len = strlen(header);
  char buf[len];

  fread ( buf, 1, len, fp );
  int width;
  int height;
  fread ( &width, 1, 2, fp );     //not used here
  fread ( &height, 1, 2, fp );     //not used here

  for ( int i = 0; i < len; ++i )
      if ( buf[i] != header[i] )
        return -1;      //wrong header

  return 1;
}

int main( int argc, char *argv[] )
{
    if (argc < 2) {
      printf("\nUsage: %s input_dct_filename", argv[0] );
      printf("\n  e.g. %s ../data/beach.dct\n", argv[0] );
      return 1;
    }

    //read the DCT data back from argv[1]
    FILE *fp = fopen ( argv[1], "rb" );
    if ( fp == NULL ) {
      printf("\nError opening file\n");
      return 1;
    }
```

```
    Run3D  runs[64];
    short  Y[8][8];
    short  Yr[8][8];
    Quantizer quantizer;
    Reorder reorder;
    Run run;
    Printer printer;

    if ( read_dct_header ( fp ) == -1 ){
        printf("\nNot dct File!\n");
        return 1;
    }

    while ( run.get64 ( Y, fp ) > 0 ) {       //read a block of 64 samples
        printf("\nA block of DCT coefficients:");
        printer.print_block ( Y );
        quantizer.quantize_block ( Y );

        printf("\nDCT block after quantization (Qstep=%d):",quantizer.Qstep);
        printer.print_block ( Y );
        reorder.reorder ( Y, Yr );

        printf("\nDCT block After zigzag reorder:");
        printer.print_block ( Yr );
        run.run_block ( Yr, runs );

        printf("\n3D run-level codewords of the reordered quantized
                                                    DCT coeficients:");
        printer.print_run ( runs );
        printf("\nHit any key to reverse the processes:");
        getchar();

        //reversing the process
        short new_Y[8][8];
        run.run_decode ( runs, new_Y );
        printf("\nDCT block after decode run:");
        printer.print_block ( new_Y );

        reorder.reverse_reorder ( new_Y, Y );
        printf("\nDCT block after reversing reorder:");
        printer.print_block ( Y );

        quantizer.inverse_quantize_block ( Y );
        printf("\nDCT block after inverse quantization:");
        printer.print_block ( Y );

        printf("\nDo you want another block? ( y/n) ");
        char c;
        scanf("%c", &c );
        if ( c == 'n' ) break;
    }
    fclose( fp );
    return 0;
}
```

Listing 7-6 shows sample outputs of the program **test_run.cpp**. You can see from the data that reordering and run-level encoding are reversible while quantization is not.

Listing 7-6: Sample outputs of test_run.cpp

```
--------------------------------------------------------------------------------
A block of DCT coefficients:
   629,     -5,     -1,     -1,      1,     -1,     -2,     -1,
    -8,     -5,     -1,      0,      0,     -1,     -2,     -2,
     2,     -3,     -1,     -1,     -4,     -4,     -3,     -2,
    -3,     -4,     -3,     -1,      1,     -2,     -4,     -3,
    -1,     -6,      0,     -1,     -1,      1,      0,      0,
    -4,     -5,     -3,      0,      2,     -1,     -1,      0,
    -1,     -2,     -2,     -1,     -1,     -1,     -1,     -2,
     1,     -1,      0,      0,      0,     -2,     -2,     -1,
DCT block after quantization ( Qstep = 12 ):
    52,      0,      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,      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,
DCT block After zigzag reorder:
    52,      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,      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,
3D run-level codewords of the reordered quantized DCT coeficients:
( 0,  52, 0)    ( 1,  -1, 1)

Hit any key to reverse the processes:

DCT block after decode run:
    52,      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,      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,
DCT block after reversing reorder:
    52,      0,      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,      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,
DCT block after inverse quantization:
   624,      0,      0,      0,      0,      0,      0,      0,
   -12,      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,      0,      0,      0,
     0,      0,      0,      0,      0,      0,      0,      0,
     0,      0,      0,      0,      0,      0,      0,      0,
Do you want another block? ( y/n)
--------------------------------------------------------------------------------
```