# An Introduction to Digital Video Data Compression in Java

Fore June

# Chapter 12 DPCM Video Codec

## 12.1 Introduction

There is a simple technique called frame differencing that we can exploit temporal redundancy in a video sequence to achieve good compression. In the method, we code the difference between one frame and the next. In other words, the predicted frame for the current frame is equal to the previous frame. We code the difference between the two frames. Again, as we discussed before, if the encoding process is lossy, we obtain the predicted frame by reconstructing it from the decoded difference so that both of the encoder and decoder use the same predicted frame. We refer to this method as differential pulse code modulation ( **DPCM** ) coding, a term borrowed from signal processing. Figure 12-1a shows a block diagram of DPCM encoding and Figure 12-1b shows the corresponding decoder.

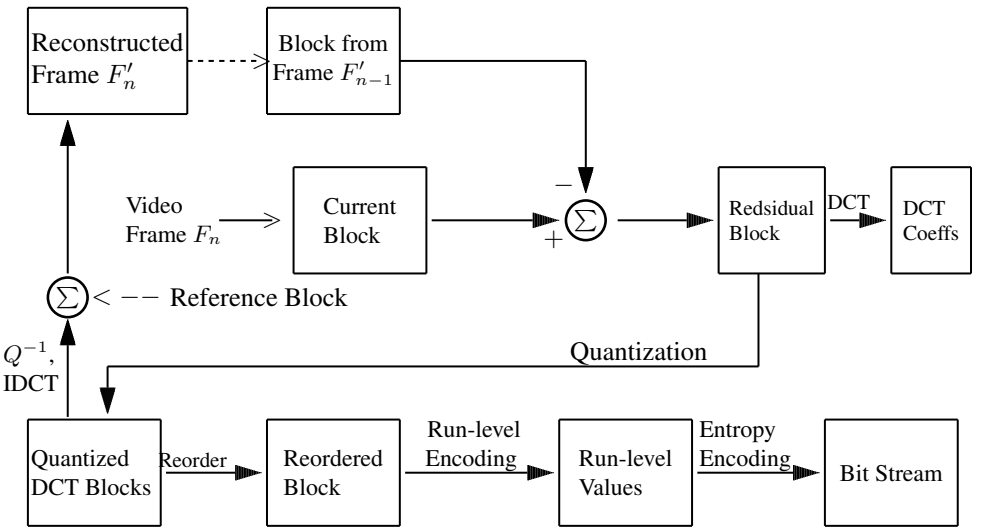


**Figure 12-1a**. DPCM Video Encoder

If there is little motion between successive frames, DPCM yields a difference image that is mostly uniform and can be coded efficiently. However, if there is rapid motion between frames or when a scene changes sharply, DPCM does not give good results.

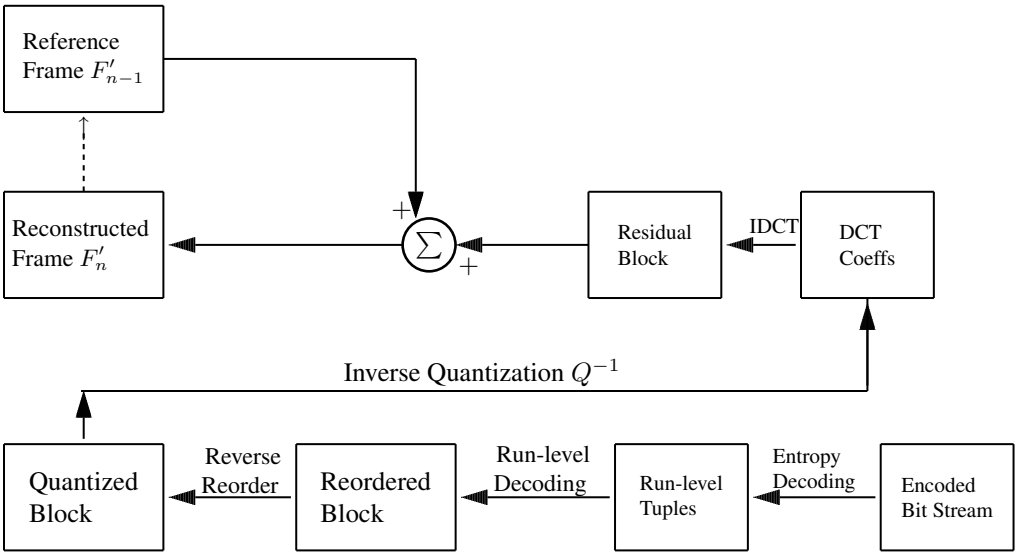


**Figure 12-1b**. DPCM Video Decoder

## 12.2 DPCM Encoding and Decoding

As shown in Figure 12-1, the DPCM encoder processes the n-th video frame $F_n$ to produce a compressed bitstream, and the decoder decompresses the encoded bitstream to the n-th reconstructed video frame $F'_n$, which is usually not identical to the original source video frame $F_n$. Many of the decoding functions are actually contained within the encoder as the encoder needs to reconstruct $F'_n$ to be used as the predictor for the next source frame. The following steps describe the encoding and decoding processes of such a codec.

**Encoder:**

1. **Forward Encoding**:

   **1.** Read a 24-bit RGB image frame from an uncompressed avi file.

   **2.** Decompose the RGB frame into $16 \times 16$ macroblocks.

   **3.** Transform and down-sample each $16 \times 16$ RGB macroblock to six $8 \times 8$ YCbCr sample blocks using YCbCr 4:2:0 format.

   **4.** Calculate the differences between the current sample blocks and the corresponding sample blocks from the previously encoded-and-reconstructed frame.

   **5.** Apply Discrete Cosine Transform ( DCT ) to each 8x8 difference sample block to obtain an $8 \times 8$ block of integer DCT coefficients.

   **6.** Forward-quantize the DCT block; reconstruct the block as described in **reconstruction process**.

   **7.** Reorder each quantized $8 \times 8$ DCT block in a zigzag manner.

**8.** Run-level encode each quantized reordered DCT block to obtain 3D ( run, level, last ) tuples.

**9.** Use pre-calculated Huffman codewords along with sign bits to encode the 3D tuples.

**10.** Save the output bitstream of the Huffman coder in a file.

2. **Reconstruction Process**:

**1.** Inverse-quantize each quantized DCT block. ( Note that quantization is a lossy process and therefore, the recovered block is not identical to the one before quantization. )

**2.** Apply Inverse DCT ( IDCT ) to resulted DCT blocks to obtain $8 \times 8$ YCbCr difference sample blocks.

**3.** Add the difference sample blocks to the corresponding reference sample blocks that were previously encoded and reconstructed.

**4.** Save the reconstructed sample blocks which will be used as the reference blocks when we encode the next frame.

**Decoder:**

1. Construct a Huffman tree from pre-calculated Huffman codewords.

2. Read a bit stream from the encoded file and traverse the Huffman tree to recover 3D run-level tuples to obtain $8 \times 8$ DCT blocks.

3. Reverse-reorder and inverse-quantize each DCT block.

4. Apply Inverse DCT ( IDCT ) to resulted DCT blocks of Step 3 to obtain $8 \times 8$ YCbCr difference sample blocks.

5. Add the difference sample blocks to the corresponding reference sample blocks that were previously encoded and reconstructed.

6. Save the decoded sample blocks which will be used as the reference blocks when we decode the next frame.

7. Use six $8 \times 8$ YCbCr sample blocks to obtain a $16 \times 16$ RGB macroblock.

8. Combine the RGB macroblocks to form an image frame.

The above descriptions and Figure 12-1 clearly present the requirement of a decoding path in the encoding process. This is necessary to ensure that the encoder and decoder use the same reference sample blocks to calculate residuals.

## 12.3 Implementation of DPCM Codec

To implement the DPCM codec, we just need to make minor modifications to the codec presented in the previous chapter ( Chapter 11 ). Actually, the main difference between the current codec and the previous one is that the current codec encodes the residuals ( differences ) between two adjacent frames as opposed to encoding the frame directly in the previous case. Note that we calculate the residuals in the YCbCr space rather than in the RGB space. A simple way to accommodate this feature is to reconstruct the YCbCr macroblock from the quantized difference DCT blocks and save it using a vector; the saved macroblock will be used as the reference block when we process the next frame. Initially, the reference block sample values are set to zero.

### Java Vector

The java Vector class implements a growable array of objects. It is in the java.util package and thus to utilize the class, one has to add the statement "import java.util.*" at the beginning of the program. Unlike arrays, vectors expand automatically when new data is added to them. The Java 2 Collections API introduced a similar data structure called ArrayList. ArrayLists are unsynchronized and therefore works faster than Vectors, but they are less secure in a multithreaded environment. The Vector class was changed in Java 2 to add the additional methods supported by ArrayList.

Vectors can hold only Objects but not primitive types like int and char. If you want to put a primitive type in a Vector, you have to embed it inside an object. For example, to save an integer value, you may use the Integer class or define your own class that contains an int. If you use the Integer wrapper, you will not be able to change the integer value, so it may be more useful to define your own class.

The Vector class was updated in Java 2 to implement the List interface. Some old methods have changed. For example, old methods **addElement**(), **elementAt**(), and **setElementAt**() have been changed to **add**(), **get**() and **set**() respectively. We will use the new methods in our programs.

### Reference Vector

We use java Vector to construct the class **RefVector** that rebuilds the reference frames. The constructor of **RefVector** creates a vector (*ycc_refv* ) that is big enough to hold all the YCbCr macroblocks of one frame; it also sets all sample values in each block to zero. The member function **calculate_diff**() calculates the residuals of a YCbCr macroblock in the current frame as compared to a corresponding one in the previous frame. This member function is used only by the **Encoder** class. The member function **reconstruct_macro**() reconstructs the YCbCr macroblock from the quantized DCT coefficients. As discussed above both the encoding and decoding processes need to reconstruct the reference frames. Thus this function is used by both the Encoder and Decoder classes. Note that inverse DCT and inverse quantization take place at this function. Therefore, the Decoder class does not need to perform these operations any more. Listing 12-1 presents the code of this class:

**Program Listing 12-1** Reference Vector

```
class RefVector {
```

```
private CircularQueue buf;  //shared buffer
private YCbCr_MACRO m;
public Vector<YCbCr_MACRO>  ycc_refv; //vector holding YCbCr macroblocks

//constructor
public RefVector ( CircularQueue q )
{
  buf = q;
  //initialize the YCbCr Macro reference vectors
  //calculate the number of macroblocks in one frame
  int n = ( buf.width/16 ) * ( buf.height / 16 );
  //set reference frame values to 0
  ycc_refv = new Vector<YCbCr_MACRO>();
  for ( int i = 0; i < n; i++ ){
    m = new YCbCr_MACRO();
    for ( int ii = 0; ii < 256; ii++ ){
      m.Y[ii] = 0;
      if ( ii < 64 )
        m.Cb[ii] = m.Cr[ii] = 0;
    }
    ycc_refv.add( m );   //push_back, capacity automatically expanded
  }
}

//Calculates residuals of macroblock pointed by mp.
//Returns results through diff.
public void calculate_diff (  YCbCr_MACRO mp, YCbCr_MACRO diff, int nm )
{
  for ( int i = 0; i < 256; ++i ) {
    diff.Y[i] = mp.Y[i] -  ycc_refv.get(nm).Y[i];
  }
  for ( int i = 0; i < 64; ++i ) {
    diff.Cb[i] =  mp.Cb[i] -  ycc_refv.get(nm).Cb[i];
    diff.Cr[i] =  mp.Cr[i] -  ycc_refv.get(nm).Cr[i];
  }
}

//reconstructs YCbCr macroblock from quantized DCT coefficients
//saves reconstructed block in ycc_refv[nm]
public void reconstruct_macro ( short dctcoefs[][][], int nm )
{
  int [][] X = new int[8][8], Y = new int[8][8];
  YCbCr_MACRO diff_macro = new YCbCr_MACRO();
  short py;
  int i, j, k = 0;
  Quantizer quantizer = new Quantizer();
  DctVideo dct_idct = new DctVideo();

  for ( int b = 0; b < 4; ++b ){        //Y blocks
    quantizer.inverse_quantize_block ( dctcoefs[b] );
    for ( i = 0; i < 8; i++ )
      for ( j = 0; j < 8; j++ )
        Y[i][j] = dctcoefs[b][i][j];
    dct_idct.idct ( Y, X );
    k = 0;
    if ( b < 2 )
      k = 8 * b;                //points to beginning of block
    else
      k = 128 + 8 * ( b - 2 );  //points to beginning of block
    for ( i = 0; i < 8; i++ ) { //one sample-block
```

```
        if ( i > 0 ) k += 16;     //advance by 1 row of macroblock
        for ( j = 0; j < 8; j++ ){
          diff_macro.Y[k+j] =  X[i][j];
        }
      }
    }  //for b

    //Cb
    quantizer.inverse_quantize_block ( dctcoefs[4] );
    for (  i = 0; i < 8; i++ )
        for ( j = 0; j < 8; j++ )
          Y[i][j] =  dctcoefs[4][i][j];
    dct_idct.idct( Y, X );
    k = 0;
    for ( i = 0; i < 8; i++ ) {
      for ( j = 0; j < 8; j++ ) {
        diff_macro.Cb[k] = X[i][j];
        k++;
      }
    }
    //Cr
    quantizer.inverse_quantize_block ( dctcoefs[5] );
    for (  i = 0; i < 8; i++ )
        for ( j = 0; j < 8; j++ )
          Y[i][j] =  dctcoefs[5][i][j];
    dct_idct.idct( Y, X );
    k = 0;
    for ( i = 0; i < 8; i++ ) {
      for ( j = 0; j < 8; j++ ) {
        diff_macro.Cr[k] = X[i][j];
        k++;
      }
    }

    YCbCr_MACRO aMacro;
    aMacro = ycc_refv.get(nm);
    for ( i = 0; i < 256; i++ ){
      aMacro.Y[i] += diff_macro.Y[i];
      if ( i < 64  ) {
        aMacro.Cb[i] += diff_macro.Cb[i];
        aMacro.Cr[i] += diff_macro.Cr[i];
      }
    }
    ycc_refv.set( nm, aMacro );
  }

  //return the requested YCbCr macroblock
  public YCbCr_MACRO get ( int nm )
  {
    return ycc_refv.get(nm);
  }
}
```

---

The encoding process has a reconstruction path. The function **reconstruct_macro()** shown above reconstructs the YCbCr macroblock from the DCT coefficients of the residuals of a current macroblock; it first inverse-quantizes the coefficients, applies IDCT to them to obtain the residuals, and then adds the residuals to the reference sample values to

reconstruct the macroblock. The newly reconstructed macroblock replaces the one in the vector *ycc_refv*; this macroblock will be used as the reference block when we encode the next frame.

Note that we do **not** need to transform the YCbCr samples back to RGB values because in the DPCM codec, we always operate in the YCbCr space when we calculate the residuals.

**Encoding:**

We only need to make minor modifications to the function **encode_one_frame**() that encodes one frame of the video. We use **calculate_diff**() of **refVector** to obtain the residuals of the current macroblock from that of the reference frame. We apply DCT transform to the residuals; other encoding processes are the same as those discussed in Chapter 11. We then use the function **reconstruct_macro**() to reconstruct the macroblock and save it in the vector *ycc_refv* of **refVector**, which will be used as the reference block ( or predictor ) when we process the corresponding macroblock of the next frame.

**Program Listing 12-2** Modified encode_one_frame() of Encoder

```
private void encode_one_frame ()
{
  //encode the frame at the head of buf
  int h = buf.getHead();
  int row, col, i, j, k, r;
  int nm = 0;                    //for indexing macroblock

  RGB_MACRO rgb_macro = new RGB_MACRO();
  YCbCr_MACRO ycbcr_macro = new YCbCr_MACRO();//macroblock for YCbCr samples
  //difference between current and reference YCbCr macroblocks
  YCbCr_MACRO ycc_diff = new YCbCr_MACRO();

  RgbYcc rgbycc = new RgbYcc ();
  short dctcoefs[][][] = new short[6][8][8];
  short Yr[][] = new short[8][8];
  short Y[][] = new short[8][8];

  Quantizer quantizer = new Quantizer();
  Reorder reorder = new Reorder();
  Run3D runs[] = new Run3D[64];
  for ( i = 0; i < 64; i++ )
    runs[i] = new Run3D();
  Run run = new Run();

  for ( row = 0; row < buf.height; row += 16 ){ //scan all rows of image
    for ( col = 0; col < buf.width; col += 16 ){//scan all columns of image
        k = (row * buf.width + col) * 3;        //x3 for RGB
        r = 0;
        for ( i = 0; i < 16; i++ ) {
          for ( j = 0; j < 16; j++ ) {
            rgb_macro.rgb[r].B = 0x000000ff & (int) buf.buffer[h][k];
            rgb_macro.rgb[r].G = 0x000000ff & (int) buf.buffer[h][k+1];
            rgb_macro.rgb[r].R = 0x000000ff & (int) buf.buffer[h][k+2];
            ++r;
            k += 3;
          }
          k += ( buf.width - 16 )*3;//points to next row within macroblock
```

```
        }

        //convert from RGB to YCbCr
        rgbycc.macroblock2ycbcr( rgb_macro, ycbcr_macro );
        //obtain difference between current and reference
        refVector.calculate_diff (  ycbcr_macro, ycc_diff, nm );
        get_dctcoefs ( ycc_diff, dctcoefs );
        for ( int bn = 0; bn < 6; bn++ ) {
          //quantize one dct sample block
          quantizer.quantize_block ( dctcoefs[bn] );
          reorder.reorder ( dctcoefs[bn], Yr ); //reorder sample block
          run.run_block(Yr, runs); //encode DCT coefs with run-level code
          /*
            Encode the 3D runs with precalculated Huffman codes using the
            provided Huffman table htable. Save the encoded bit stream in
            file pointed by bitout.
          */
          hcodec.huff_encode ( runs, bitout );   //encode and save
        } //for bn
        //reconstruct YCbCr macroblock from quantized DCT coefficients;
        //  save block in ycc_refv[nm] of refVector
        refVector.reconstruct_macro ( dctcoefs, nm );
        nm++;                                     //next macroblock
      } //for col
    } //for row
}
```

**Decoding**:

The decoding process makes use of the function **reconstruct_macro**() of the **RefVector** class to decode the bitstream. We make minor modifications to the decoding code discussed in Chapter 11. The main change is in the **get_yccblocks**() function. It uses **get_dct_block()** to get a residual DCT sample block by reading from the input bitstream, and carrying out the Huffman decoding, run-level decoding and reverse-reordering. However, **get_dct_block**() does not perform the task of inverse-quantization, which has been moved to the function **reconstruct_macro**(). The function **get_yccblocks**() gathers six quantized sample DCT blocks into the array *dctcoefs* with the help of **copy_dct_block**() and calls **reconstruct_macro**() to reconstruct the corresponding YCbCr macro block which will be held by the Vector *ycc_refv* of **RefVector**. Vector *ycc_refv* contains all reference YCbCr macroblocks:

**Program Listing 12-3** Modified get_yccblocks of Decoder

```
private int get_yccblocks( YCbCr_MACRO ycbcr_macro, int nm )
{
  int r, row, col, i, j, k, n, b, c;
  byte  abyte;
  int [][] Y = new int[8][8], X = new int[8][8];
  short dctcoefs[][][] = new short[6][8][8];

  //read data from file and put them in four 8x8 Y sample blocks
  for ( b = 0; b < 4; b++ ) {
    if ( !get_dct_block ( Y ) )
      return 0;
```

```
    copy_dct_block ( Y, dctcoefs, b );
  } //for b

  //now do that for 8x8 Cb block
  if ( !get_dct_block( Y ) )     //read in one DCT block
    return 0;
  copy_dct_block ( Y, dctcoefs, 4 );

  //now do that for 8x8 Cr block
  if ( !get_dct_block( Y ) )     //read in one DCT block
    return 0;
  copy_dct_block ( Y, dctcoefs, 5 );

  //Reconstruct current frame from DCT coefficients
  refVector.reconstruct_macro ( dctcoefs, nm );
  YCbCr_MACRO aMacro;
  aMacro = refVector.get(nm);
  for ( i = 0; i < 256; i++ ){
    ycbcr_macro.Y[i] = aMacro.Y[i];
    if ( i < 64  ) {
      ycbcr_macro.Cb[i] = aMacro.Cb[i];
      ycbcr_macro.Cr[i] = aMacro.Cr[i];
    }
  }

  n = 6 * 64;
  return n;               //number of bytes read
}
```

---

The function **decode_one_frame**() makes use of **get_yccblocks**() and **ycbcr2macroblock**() to recover the RGB samples. All other classes used are the same as before. Thus the class files that reside in the directory of this Chapter ( **12/** ) are: **Decoder.java, Encoder.java**, and **RefVector.java**. Again, you need to point CLASSPATH to the directories that contain classes developed in previous chapters and are used by the Encoder and Decoder classes. The following command will do the job:

*export CLASSPATH=$CLASSPATH:../5/:../6/:../7/:../8:../10/:../11/:../11/avi/ImageJ/ij.jar*

After setting the correct class paths, we can run Vcodec in the current directory **12/**. For example, the following command,

*java Vcodec ../data/jvideo.avi*

generates the compressed file "jvideo.fjv". ( The Vcodec class resides in directory **11/** but we have pointed the class path to that directory. ) We may check the file sizes with the "ls" command:

```
$ ls -l ../data/jvideo.*
-rw-r--r-- 1 user user  5762656 ../data/jvideo.avi
-rw-r--r-- 1 user user   446687 ../data/jvideo.fjv
```

To decode and play the video of "jvideo.fjv", we can run Vcodec with the "-d" switch:

```
java Vcodec -d ../data/jvideo.fjv
```

In the next chapter, we will discuss the implementation of a codec that includes more formal motion estimation and compensation.

Other books by the same author

# Windows Fan, Linux Fan

by *Fore June*

*Windws Fan, Linux Fan* describes a true story about a spiritual battle between a Linux fan and a Windows fan. You can learn from the successful fan to become a successful Internet Service Provider ( ISP ) and create your own wealth.

Second Edition, 2002.
ISBN: 0-595-26355-0 Price: $6.86

# An Introduction to Video Compression in C/C++

by *Fore June*

The book describes the the principles of digital video data compression techniques and its implementations in C/C++. Topics covered include RBG-YCbCr conversion, macroblocks, DCT and IDCT, integer arithmetic, quantization, reorder, run-level encoding, entropy encoding, motion estimation, motion compensation and hybrid coding.

January 2010
ISBN: 9781451522273

# An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

by *Fore June*

November 2011
ISBN-13: 978-1466488359