An Introduction to Digital Video Data Compression in Java

Fore June

Chapter 11 Video File Formats and Codec Player

11.1 Introduction

In Chapter 10, we have discussed how to play a video using data saved in the raw format of a file. In reality, video data are saved in a predefined format. There have been numerous video formats around and the data of most of them are saved in compressed form. In this Chapter, we shall give a brief discussion on some popular formats and do a case study on the **.avi** file format. Just as raw pixel data are often saved in **.bmp** format, and raw PCM sound samples in **.wav** format, raw video data are often saved in **.avi** format. We shall learn how to extract the raw video data from a .avi file so that we can play the video that we discussed in Chapter 10 or process them in our own way. There exists utilities in the Internet that allows one to convert from other video formats to uncompressed .avi format. With the help of those utilities, we can download a video file saved in any format from the Internet, convert it to uncompressed .avi and experiment the converted uncompressed data with our own encoder, decoder, and video player.

11.2 Video Storage Formats

11.2.1 Requirements of Video File Format

There exists a large number of video file formats in the market not only because competing companies create their own formats, hoping to push out competitors and to make their formats standards, but also because there are legal needs of not overstepping competitors' so called intellectual property. The following sections examine some common characteristics between the popular file formats. We summarize the requirements for a video file format to be successful as follows. A video file format should be able to

1. store video and audio data,
2. provide fast, real-time playback on target viewing platforms,
3. provide efficient scrubbing (fast-forward and rewind while previewing),
4. store metadata (e.g. copyright, authorship, creation dates ...),
5. store additional tracks and multimedia data like thumbnails, subtitle tracks, alternate language audio tracks ...,
6. allow for multiple resolutions,
7. provide file locking mechanisms,
8. allow for video editing,
9. provide integrity checking mechanism, and
10. perform segmentation of audio and video portions into packets for efficient Internet transmission.

11.2.2 Common Internet Video Container File Format

A container file format is hierarchical in structure, and can hold different kinds of media (audio, video, text ..) synchronized in time. The following are some popular container formats, which can save various types of media data:

1. AVI (Audio Video Interleaved) – standard audio / video file format under Windows; not suited for streaming as it does not have any standard way to store packetization data.
2. MOV – Apple's Quick Time format, better than AVI in synching audio and video; also supports Windows, and Linux.
3. ASF (Advanced Streaming Format) – Microsoft's proprietary format (.WMV, .WMA), designed primarily to hold synchronized audio and video.
4. NSV (NullSoft Video) – by NullSoft (a division of AOL) for streaming.
5. RM (RealMedia) – Real's streaming media files; can be extended to hold all types of multimedia; supports Windows, and Linux platforms and many standards, including MPEG-2, MPEG-4, Real, H263.
6. MP4 (MPEG-4) – almost identical to MOV but MPEG-4 players can handle only MPEG-4 related audio, video and multimedia.
7. SWF, SWV (Shockwave Flash) – for Flash movies, typically containing vector-drawn animations with scripting controls; also supports video codecs, JPEG still images, remote loading of SWF files, XML data and raw text.

11.2.3 Simple Raw or Stream Video Format

Simple raw storage or stream formats store the compressed data without extra headers or metadata. These are essentially live audio-video streams saved to disk. Below are some examples:

1. MPEG-1, MPEG-2 – streams are composed of interleaved audio and video data arranged into groups of pictures.
2. MP3 – encode audio; part of MPEG-1.
3. DV (Digital Video) – used by modern digital cameras and video editing software.
4. .263 – video compressed with H.263 codec.
5. .RTP – Real Time Protocol data.

11.2.4 Internet Playlist, Index, and Scripting Format

Index formats have pointers linking to other resources. The following are some of this kind:

1. MOV – has several formats that do not contain actual video or audio data, but merely point to other files.
2. RAM (RealAudio Metafile) – points to the URL of the actual media file (RealMedia .RM or RealAudio .RA).
3. ASX (Active Streaming Index) – indexes files that work in Windows Media system and point to the content held in an ASF media file.
4. SMIL (Synchronized Multimedia Integration Language) – provides instructions to a media player on how to present a multimedia interface and what content to display.

11.3 Case Study: AVI Files (.avi)

AVI (Audio Video Interleaved) is a file format defined by Microsoft for use in applications that capture, edit and play back audio-video sequences. Just as raw pixel data are often saved in .bmp format, and raw PCM sound samples in .wav format, raw video data are often saved in .avi format. It is a special case of **RIFF** (Resource Interchange File Format) and is the most commonly used format for storing audio/video data in a PC. An AVI file can be embedded in a web page using a link like:

< A HREF="http://www.*somedomain*.com/movie.avi" > A Movie </A>

In order that your Apache Web server is able to handle avi files, you need to add in the configuration file the following statement.

```
AddType video/avi .avi
```

AVI is considered as an obsolete video/audio file format as it lacks many contemporary and crucial features to support streaming and image processing. However, it has been extended by OpenDML to include some of those features.

11.3.1 RIFF File Format

The AVI file format is based on the RIFF (resource interchange file format) document format. A RIFF file consists of a RIFF header followed by zero or more lists and chunks; it uses a FOURCC (four-character code) to denote a text header. A FOURCC is a 32-bit unsigned integer created by concatenating four ASCII characters. For example, 'abcd' = 0x64636261. The AVI file format uses FOURCC code to identify stream types, data chunks, index entries, and other information. The RIFF file format has the following form.

1. A RIFF header consists of
   'RIFF' *fileSize fileType* (*data*)
   where
   - 'RIFF' is the literal FOURCC code 'RIFF',
   - *fileSize* is a 4-byte value indicating the size of the data in the file including the size of the *fileType* plus the size of the data that follows,
   - *fileType* is a FOURCC that identifies the specific file type,
   - *data* consists of chunks and lists in any order.
2. A **chunk** consists of
   chunkID chunkSize chunkData
   where
   - *chunkID* is a FOURCC that identifies the data contained in the chunk,
   - *chunkSize* is a 4-byte value giving the size of data in chunkData not including padded values,
   - *chunkData* is zero or more bytes of data, padded to nearest WORD boundary.
3. A **list** consists of
   'LIST' *listSize listType listData*
   where

- ‘LIST’ is the literal FOURCC code ‘LIST’,
- *listSize* is a 4-byte value, indicating the size of the list,
- *listType* is a FOURCC code specifying the list type,
- *listData* consists of chunks or lists, in any order.

Table 11-1 shows some sample data from an AVI file. The data are displayed in hexadecimal; the corresponding ASCII characters are printed on the right if they are printable otherwise a dot is printed. Some comments are shown at the far right to indicate what the data represent.

Table 11-1 Sample AVI Data

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15  0123456789012345]
52 49 46 46 DC 6C 57 09 41 56 49 20 4C 49 53 54 |RIFF.lW.AVI LIST|RIFF fileSize
                                                                  fileType LIST
CC 41 00 00 68 64 72 6C 61 76 69 68 38 00 00 00 |.A..hdrlavih8...|listSize
                                                        listType avih structSize
50 C3 00 00 00 B0 04 00 00 00 00 00 10 00 00 00 |P...............|
                                              microSecondPerFrame maxBytesPerSec
A8 02 00 00 00 00 00 00 01 00 00 00 00 84 03 00 |................|totalFrames
                                          initFrames streams suggestedBufferSize
40 01 00 00 F0 00 00 00 00 00 00 00 00 00 00 00 |@...............|width height
00 00 00 00 00 00 00 00 4C 49 53 54 74 40 00 00 |........LISTt@..|
73 74 72 6C 73 74 72 68 38 00 00 00 76 69 64 73 |strlstrh8...vids|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
64 00 00 00 D0 07 00 00 00 00 00 00 A8 02 00 00 |d...............|
00 84 03 00 10 27 00 00 00 00 00 00 00 00 00 00 |.....’..........|
40 01 F0 00 73 74 72 66 28 00 00 00 28 00 00 00 |@...strf(...(...|
40 01 00 00 F0 00 00 00 01 00 18 00 00 00 00 00 |@...............|
00 84 03 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00 00 00 00 69 6E 64 78 F8 3F 00 00 04 00 00 00 |....indx.?......|
01 00 00 00 30 30 64 62 00 00 00 00 00 00 00 00 |....00db........|
00 00 00 00 0C 44 00 00 00 00 00 00 00 40 00 00 |.....D.......@..|
.
.
4C 49 53 54 38 F9 56 09 6D 6F 76 69 69 78 30 30 |LIST8.V.moviix00|LISTlistSize
                                                             listType indexBlock
F8 3F 00 00 02 00 00 01 A8 02 00 00 30 30 64 62 |.?..........00db|
                                                      ....00db(uncompress frame)
```

As we can see from the sample avi data of Table 11-1, a two-character code is used to define the type of information in the chunk:

Two-character code	**Description**
db	Uncompressed video frame
dc	Compressed video frame
pc	Palette change
wb	Audio data

For example, if stream 0 contains audio, the data chunks for that stream would have the FOURCC ‘00wb’. If stream 1 contains video, the data chunks for that stream would have the FOURCC ‘01db’ or ‘01dc’.

11.3.2 AVI RIFF Format

As shown in Table 11-1, the FOURCC ‘AVI ’ in a RIFF header identifies the file to be an AVI file. An AVI file has two mandatory LIST chunks, defining the format of the streams and the stream data, respectively. An AVI file might also include an index chunk, indicating the address of the data chunks of the file; it has the following form (Table 11-2):

Table 11-2 AVI RIFF Format

```
RIFF ('AVI '
        LIST ('hdrl' ... )
        LIST ('movi' ... )
        ['idx1' () ]
       )
```

The ‘hdrl’ list defines the format of the data and is the first mandatory LIST chunk. The ‘movi’ list contains the data for the AVI sequence and is the second required LIST chunk. An optional index (‘idx1’) chunk can follow the ‘movi’ list. The index contains a list of the data chunks and their location in the file. If we expand ‘hdrl’ and ‘movi’ in Table 11-2, we shall get a form as shown below:

```
RIFF ('AVI '
      LIST ('hdrl'//header length
            'avih'()
            LIST ('strl'//stream length
                  'strh'()
                  'strf'()
                  [ 'strd'() ]
                  [ 'strn'() ]
                  ...
                 )
             ...
           )
      LIST ('movi'
            {SubChunk | LIST ('rec '
                                  SubChunk1
                                  SubChunk2
                                  ...
                                 )
               ...
            }
            ...
           )
      ['idx1' () ]
     )
```

The ‘hdrl’ list begins with the main AVI header, which is contained in an ‘avih’ chunk. The main header contains global information for the entire AVI file, such as the number of streams within the file and the width and height of the AVI sequence. This main header structure is shown below:

```
typedef struct _avimainheader {
    FOURCC fcc;      //'avih'
    DWORD  cb;       //size of structure, not including 1st 8 bytes
    DWORD  dwMicroSecPerFrame;
    DWORD  dwMaxBytesPerSec;
    DWORD  dwPaddingGranularity;
    DWORD  dwFlags;
    DWORD  dwTotalFrames;
    DWORD  dwInitialFrames;
    DWORD  dwStreams;
    DWORD  dwSuggestedBufferSize;
    DWORD  dwWidth;
    DWORD  dwHeight;
    DWORD  dwReserved[4];
} AVIMAINHEADER;
```

One or more ‘strl’ lists follow the main header. A ‘strl’ list is required for each data stream. Each ‘strl’ list contains information about one stream in the file, and must contain a stream header chunk (‘strh’) and a stream format chunk (‘strf’). In addition, a ‘strl’ list might contain a stream-header data chunk (‘strd’) and a stream name chunk (‘strn’). The stream header chunk (‘strh’) consists of an AVISTREAMHEADER structure as shown in Table 11-3.

Table 11-3 AVI Main Stream Header

```
typedef struct _avistreamheader {
     FOURCC fcc;
     DWORD  cb;
     FOURCC fccType;//'vids'-video, 'auds'-audio, 'txts'-subtitle
     FOURCC fccHandler;
     DWORD  dwFlags;
     WORD   wPriority;
     WORD   wLanguage;
     DWORD  dwInitialFrames;
     DWORD  dwScale;
     DWORD  dwRate;
     DWORD  dwStart;
     DWORD  dwLength;
     DWORD  dwSuggestedBufferSize;
     DWORD  dwQuality;
     DWORD  dwSampleSize;
     struct {
         short int left;
         short int top;
         short int right;
         short int bottom;
     }  rcFrame;
} AVISTREAMHEADER;
```

One can also express Digital Video (DV) data in the AVI file format. The following example shows the AIFF RIFF form for an AVI file with one DV data stream, expanded with completed header chunks.

Example AVI File With One DV Stream

```
00000000 RIFF (0FAE35D4) 'AVI '
0000000C     LIST (00000106) 'hdrl'
00000018         avih (00000038)
                   dwMicroSecPerFrame    : 33367
                   dwMaxBytesPerSec      : 3728000
                   dwPaddingGranularity  : 0
                   dwFlags            : 0x810 HASINDEX|TRUSTCKTYPE
                   dwTotalFrames         : 2192
                   dwInitialFrames       : 0
                   dwStreams             : 1
                   dwSuggestedBufferSize : 120000
                   dwWidth               : 720
                   dwHeight              : 480
                   dwReserved            : 0x0
00000058         LIST (0000006C) 'strl'
00000064             strh (00000038)
                         fccType               : 'iavs'
                         fccHandler            : 'dvsd'
                         dwFlags               : 0x0
                         wPriority             : 0
                         wLanguage             : 0x0 undefined
                         dwInitialFrames       : 0
                         dwScale         : 100 (29.970 Frames/Sec)
                         dwRate                : 2997
                         dwStart               : 0
                         dwLength              : 2192
                         dwSuggestedBufferSize : 120000
                         dwQuality             : 0
                         dwSampleSize          : 0
                         rcFrame               : 0,0,720,480
000000A4             strf (00000020)
                         dwDVAAuxSrc     : 0x........
                         dwDVAAuxCtl     : 0x........
                         dwDVAAuxSrc1    : 0x........
                         dwDVAAuxCtl1    : 0x........
                         dwDVVAuxSrc     : 0x........
                         dwDVVAuxCtl     : 0x........
                         dwDVReserved[2] : 0,0
000000CC     LIST (0FADAC00) 'movi'
0FADACD4     idx1 (00008900)
```

11.4 Utility Program for Reading AVI Files

In order to extract the data from an AVI file, we need a program that can understand the AVI format. Rather than reinventing the wheel and developing such a program, which is not directly related to video compression, we shall make use of some existing open-source libraries and code to help us do the job.

We shall utilize the **ImageJ** library to help us read uncompressed .avi files. ImageJ is a public domain java image processing and analysis program. You may download it from the site,

```
http://rsb.info.nih.gov/ij/
```

which also contains detailed documentation of the package. You may also obtain it from the web site of this book. The package is commonly referred to as **ij** and comes as a jar file named **ij.jar**.

You may run ImageJ, either as an online applet or as a downloadable application, on any computer with java1.5 or later installed. It can display, edit, analyze, process, save and print 8-bit, 16-bit and 32-bit images. It can read many image formats including TIFF, GIF, JPEG, BMP, DICOM, FITS and 'raw'. It supports 'stacksx' (and hyperstacks), a series of images that share a single window.

Actually, we only have to use one of the programs, **AVI_Reader.java** in the package. AVI_Reader is written by Michael Schmid, base on a plugin by Daniel Marsh and Wayne Rasband. It only has very limited support for processing .avi files but the functions are rich enough for our purpose, which is mainly to read the data of an uncompressed .avi file. As of this writing, AVI_Reader only supports the following formats:

1. uncompressed 8 bit with palette (=LUT)
2. uncompressed 8 and 16 bit grayscale
3. uncompressed 24 and 32 bit RGB (alpha channel ignored)
4. uncompressed 32 bit AYUV (alpha channel ignored)
5. various YUV 4:2:2 compressed formats
6. png or jpeg-encoded individual frames.

It also has the following limitations:

1. Most MJPG (motion-JPEG) formats are not read correctly.
2. Does not read avi formats with more than one frame per chunk.
3. Palette changes during the video are not supported.
4. Out-of-sequence frames (sequence given by index) not supported.
5. Different frame sizes in one file (rcFrame) are not supported.
6. Conversion of (A)YUV formats to grayscale is non-standard: all 255 levels are kept as in the input (i.e. the full dynamic range of data from a frame grabber is preserved).

The above information of AVI_Reader can be found at the site *http://rsbweb.nih.gov/ij/plugins/avi-reader.html*. The relations of AVI_Reader class to other java classes are shown below:

```
java.lang.Object
  l-- ij.ImageStack
        l-- ij.VirtualStack
              l--ij.plugin.AVI_Reader
```

AVI_Reader extends Virtual_Reader and implements PlugIn of ij. It has only one constructor, "AVI_Reader()". The following are some of the commonly used methods of AVI_Reader:

```
public int getWidth()

  Returns the image width of a frame
----------------------------------------------------------
public int getHeight()

  Returns the image height of a frame
----------------------------------------------------------
public int getSize()

  Returns the number of frames in the video
```

```
    ---------------------------------------------------------
    public ImageStack makeStack(java.lang.String path,
                                int firstFrameNumber,
                                int lastFrameNumber,
                                boolean isVirtual,
                                boolean convertToGray,
                                boolean flipVertical)

      Create an ImageStack from an avi file with given path.

      Parameters:
        path - Directoy+filename of the avi file
        firstFrameNumber - Number of first frame to read (starts from 1)
        lastFrameNumber - Number of last frame to read or 0 for reading all,
                            -1 for all but last...
        isVirtual - Whether to return a virtual stack
        convertToGray - Whether to convert color images to grayscale
      Returns:
        Returns the stack; null on failure. The stack returned may be
           non-null, but have a length of zero if no suitable frames were
           found
```

Listing 11-1 shows a simple example of using AVI_Reader to play an uncompressed .avi file and to print out some parameters of the .avi file. It uses the the method **getPixels**() to read the RGB pixel values from the file and put them into an integer array:

Program Listing 11-1: PlayAVI Plays Uncompressed AVI Video Using AVI_Reader

```
/*
  PlayAVI.java
  Demonstrates the use of AVI_Reader of ImageJ to
  play an uncompressed .avi file.
*/
import java.io.*;
import java.awt.Frame;
import java.awt.image.*;
import java.awt.image.ColorModel;
import java.awt.color.ColorSpace;
import java.awt.*;
import javax.media.jai.widget.ScrollingImagePanel;
import javax.media.jai.*;
import ij.plugin.AVI_Reader;

public class PlayAVI {

  public static void main(String[] args) throws InterruptedException {
    if (args.length != 1) {
      System.out.println("Usage: java "  +
                                    "AVI_input_image_filename");
      System.exit(-1);
    }

    AVI_Reader avi = new AVI_Reader();
    //read images from AVI file to virtual stack
    avi.makeStack ( args[0], 0, 0, true, false, false  );

    int vsize = avi.getSize();     //number of images in the virtual stack
    int width = avi.getWidth();    //image width
    int height = avi.getHeight();  //image height
    System.out.printf("total frames=%d, width=%d, height=%d \n",
                                              vsize, width, height );

    //create a TiledImage using a ColorModel and a SampleModel
    ColorSpace colorspace = ColorSpace.getInstance ( ColorSpace.CS_sRGB );
    int[] bits = { 8, 8, 8 };
    ColorModel  colormodel = new ComponentColorModel( colorspace, bits,
        false, false, Transparency.OPAQUE, DataBuffer.TYPE_BYTE );
    TiledImage outImage;
    SampleModel  samplemodel = new BandedSampleModel(DataBuffer.TYPE_BYTE,
                                                     width, height, 3);
    outImage = new TiledImage(0,0,width,height,0,0,samplemodel,colormodel);
    ScrollingImagePanel panel = new ScrollingImagePanel(outImage,width,height);
    Frame window = new Frame("AVI_Reader Demo");
    window.add(panel);
    window.pack();
    window.show();

    //render the AVI images
    for ( int s = 1; s < vsize; ++s ) {
      Object obj = avi.getPixels ( s );
      if ( obj instanceof int[] ) {
        int[] pixels = (int[]) obj;
        int k = 0;
        for (int y = 0; y < height; y++) {
          for (int x = 0; x < width; x++) {
            int c, n = 16;
            //consider colors R, G, B
            for ( int i = 0; i < 3; ++i ) {
              c = (int)( 0x000000ff & (pixels[k]>>n) );
              outImage.setSample(x, y, i, c );
              n -= 8;
            }
            k++;
          }
        }
        panel.set ( outImage );
        Thread.sleep ( 60 );
      } else {
        System.out.println("Only supports integer pixels\n");
        System.exit ( -1 );
      }
    }
    Thread.sleep( 4000 ); //sleep for four seconds
    window.dispose();     //close the frame
  }
}
```

You can simply compile the program with the command “javac PlayAVI.java”. However, you must point your class path to the correct location of the ImageJ library. The following are typical commands to set the class path, compile and run the program,

```
export CLASSPATH=$CLASSPATH:/path_to_library/ImageJ/ij.jar
javac PlayAVI.java
java PlayAVI ../data/jvideo.avi
```

where *path_to_library* is the path to the directory that saves the ImageJ library, and "jvideo. avi" is a sample uncompressed .avi file which can be downloaded from the web site of this book.

11.5 A Simple Video Codec for Intra Frames

In previous chapters, we have discussed the coding and decoding of intra-frames of a video as well as playing videos using a solution for the producer-consumer problem. In this section, we combine all the code and put together a simple video codec (coder-decoder) that does intra-frame coding and decoding. At this point, our codec does not consider inter-frame coding that utilizes Motion Estimation (ME) and Motion Compensation (MC). Also, the pre-calculated Huffman code is very brief and is far from optimized.

Like what we did before, we use integer arithmetic in the implementation to speed up the computing process; we utilize the ImageJ and Java Imaging packages to simplify the program and make it more robust. The codec can compress an uncompressed AVI file and play it back. We assume that the video data of the avi file are saved using the 24-bit RGB colour model. Most of the code presented here have been discussed in previous sections or chapters.

11.5.1 Compressed File Header

We first define our own header of the compressed file. The header contains the basic information of the compressed data and consists of 24 bytes as listed in Table 11-4.

Table 11-4

Bytes	Information
0 - 9	contains "FORJUNEV" as I.D. of file
10 - 11	frame rate (frames per second)
12 - 15	number of frames
16 - 17	width of an image frame
18 - 19	height of an image frame
20	bits per pixel
21	quantization method
22	extension, 0 for uncompressed data
23	dummy

The default extension of such a compressed file is **.fjv**. The dummy byte in the header is used for byte-alignment so that the header makes up 24 bytes rather than 23 bytes; it may be used for other purposes in the future. This header can be implemented by defining a class like the following:

```
class Vheader {
  public byte  id[] = new byte[10]; //I.D. of file, ``FORJUNEV"
  public short fps;      //frame per second
  public int   nframes;  //number of frames
  public short width;    //width of video frame
  public short height;   //height of video frame
  public byte  bpp;      //bits per pixel
  public byte  qmethod;  //quantization method
  public byte  ext;      //extension
  public byte  dummy;    //for byte alignment,make header size=24

  //Constructor
  Vheader ()
  {
    byte src [] = {'F', 'O', 'R', 'J', 'U', 'N', 'E', 'V', 0, 0};
    System.arraycopy(src, 0, id, 0, src.length);//set header I.D.
  }

  //Constructor
  Vheader (short w, short h, short frameRate, int numVideoFrames)
  {
    this();     //call the other constructor
    //System.out.printf( "%s\n", id );
    fps = frameRate;      //frame per second
    nframes = numVideoFrames;  //number of frames
    width = w;                 //image width
    height = h;                //image height
    bpp = 8;                   //number of bits per pixel
    qmethod = 1;               //quantization method
    ext = 1;                   //file contains compressed data
    dummy = 0;
  }
  ......
}
```

At present, we set the quantization method variable *qmethod* to 1. The extension variable *ext* is set to 0 if the **.fjv** data are uncompressed and 1 if they are compressed. In summary, at this point the encoding and decoding processes consist of the following steps:

1. **Encoding**:

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

   2. Decompose the RGB frame into 16×16 macroblocks.

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

   4. Apply Discrete Cosine Transform (DCT) to each 8×8 sample block to obtain an 8×8 block of integer DCT coefficients.

   5. Forward-quantize the DCT block.

   6. Reorder each quantized 8×8 DCT block in a zigzag manner.

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

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

9. Save the output bit stream of the Huffman coder in a file.

2. **Decoding**:

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 8×8 DCT blocks.

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

4. Apply Inverse DCT (IDCT) to resulted DCT blocks to obtain 8x8 YCbCr sample blocks.

5. Use six 8×8 YCbCr sample blocks to obtain a 16×16 RGB macroblocks.

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

We have discussed all of the above steps and their implementations in detail in previous chapters. We just need to make very minor modifications to accomplish the task. The two files **Encoder.java**, and **Decoder.java** discussed in Chapter 6 need some modifications. The file **CircularQueue.java** also needs modification because we now use **AVI_Reader** to put samples into the circular queue. On the other hand, the following files can be used without any any modifications:

Table 11-5

Files	**Chapter**
RgbYcc.java	5
common.java	5
DctVideo.java	6
Quantizer.java	7
Reorder.java	7
Run3D.java	7
Run.java	7
BitIO.java	8
Hcodec.java	8
Dtables.java	8

Of course, to use the the classes of the files, we need to point the class path to the directories that contain those classes. Suppose our current working directory is **11/** and the ImageJ package is in **11/avi/**. The following is a typical command that may be used to set the class path to point to the necessary classes discussed above:

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

The files that we need to modify or develop in this section (in the direcotry **11/**) include the following:

AviFrameProducer.java	Decoder.java	Encoder.java	Vheader.java
CircularQueue.java	Display.java	Vcodec.java	

We are going to discuss these new files briefly below. Again, the whole package along with a couple of sample AVI files are provided in the book's web site at:

http://www.forejune.com/jvcompress/.

1. **Vheader.java**

   The class **Vheader** reads and writes the header of a **.fjv** file defined in Table 11-4. Its data members and constructors are shown above in section 11.5.1.

2. **Display.java**

   The class **Display** simply displays a frame in the window. It uses the ColorModel, ColorSpace and SampleModel discussed in Chapter 10 to create a TiledImage which is attached to a ScrollingImagePanel. It gets the image data from the head of the shared circular buffer and send the data to the panel using the method **set()**:

   Program Listing 11-2: Display Class Renders the Image of a Frame

```
public class Display {
 private CircularQueue buf;
 private int height;
 private int width;
 private TiledImage outImage;
 private ScrollingImagePanel panel;
 private Frame window;

 //constructor
 public Display ( CircularQueue q, int w, int h )
 {
   buf = q;
   width = w;
   height = h;

   //create a TiledImage using a ColorModel and a SampleModel
   ColorSpace colorspace=ColorSpace.getInstance(ColorSpace.CS_sRGB);
   int[] bits = { 8, 8, 8 };
   ColorModel  colormodel = new ComponentColorModel(colorspace,
        bits,false,false,Transparency.OPAQUE, DataBuffer.TYPE_BYTE);
   SampleModel samplemodel=new BandedSampleModel
                        (DataBuffer.TYPE_BYTE, width, height, 3);
   outImage=new TiledImage(0,0,width,height,0,0,samplemodel,
                                                    colormodel);
   panel = new ScrollingImagePanel(outImage,width,height);
   window = new Frame("Player");
   window.add(panel);
   window.pack();
   window.show();
 }

 public void display_frame()
 {
    //render the  images
    int h = buf.getHead();
    int k = 0;
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        int c;
        //consider colors R, G, B
        for ( int i = 0; i < 3; ++i ) {
          c = (int)( buf.buffer[h][k++] );
          outImage.setSample(x, y, i, c );
        }
      }
    }
    panel.set ( outImage );
  }

 public void close_window()
 {
    window.dispose();     //close the frame
 }
}
```

3. **AviFrameProducer.java**:

   The class **AviFrameProducer** makes use of **AVI_Reader** to read frames from an avi file and put the frames in the shared circular buffer. It acts as a producer which produces uncompressed frames for consumers to consume:

   Program Listing 11-3: AviFrameProducer is a Producer of Image Frames

```
import ij.plugin.AVI_Reader;

//producer
class AviFrameProducer extends Thread {

  private AVI_Reader avi;
  private CircularQueue buf;

  //constructor
  public AviFrameProducer(AVI_Reader aviReader, CircularQueue q)
  {
    avi = aviReader;
    buf = q;
  }

  public void run()
  {
    //get number of images in the virtual stack
    int vsize = avi.getSize();
    int nFrame = 1;
    while ( !buf.quit ) {
      buf.waitIfBufferFull();
      //produces data
      buf.putSamples ( avi, nFrame );  //put samples in buffer
      buf.tailInc();
      ++nFrame;
      if ( nFrame > vsize )
        buf.quit = true;
    }
  }
}
}
```

4. **Vcodec.java**:

   The class **Vcodec** consists of the entry point **main** () function. The **main**() asks for an AVI file as input to encode. If a switch "-d" is provided, it tries to decode the input file whose data are saved in the **.fjv** format; it plays the video data using the **Display** class. It utilizes **Encoder** to encode uncompressed video data and **Decoder** to decode compressed data. In the encoding process, **AviFrameProducer** acts as a producer and puts image data in the shared circular buffer; the **Encoder** acts as a consumer which gets data from the circular buffer and encodes them. It also uses **Display** to render the frames; if you do not want to render the frames while encoding, simply comment out the statement "display.display_frame();" in the run() method of the file "Encoder.java".

   In the decoding process, the **Decoder** is a producer and puts data in the shared circular buffer; we have not written a player class which acts as a consumer to fetch data from the buffer and plays them on the screen; instead the **Decoder** renders the frame itself after decoding it.

   You may compile all the java programs in the directory **11/** using the command, "javac *.java" after setting the CLASSPATH correctly. The following are two examples of using **Vcodec** to compress and decompress video data. The following command compresses the avi file "t.avi" in the directory:

   ```
   java Vcodec t.avi
   ```

   It encodes the video data and saves the compressed data in the file "t.fjv". The following command decodes the compressed data and renders the images on the screen:

   ```
   java Vcodec -d t.fjv
   ```

The programs presented here are for intra-frame coding. No temporal correlations have been considered. However, the code presented here forms the basis for implementations of more advanced compression techniques. In the next two chapters, we discuss the implementations of inter-frame coding and codecs that exploit temporal redundancies. A lot of the code presented here will be reused in those implementations.