An Introduction to Digital Video Data Compression in Java

Fore June

Chapter 4 Image and Video Storage Formats

There are a lot of proprietary image and video file formats, each with clear strengths and weaknesses. The file formats are generally not a user-defined option and many of the features are specified by the vendors. This book is about video compression programming and we are not interested in exploring various file formats. However, we do need to know a few formats in order that we can carry out experiments on image or video compression using files downloaded from the Internet. Therefore, we shall discuss a couple simple standard formats and some related tools that we will use later in this book.

4.1 Portable Pixel Map (PPM)

The Portable Pixel Map (PPM) file format is a lowest and simplest common denominator color image format. A PPM file contains very little information about the image besides basic colors and thus it is easy to write programs to process the file, which is the purpose of this format. A PPM file consists of a sequence of one or more PPM images. There are no data, delimiters, or padding before, after, or between images. The PPM format closely relates to two other bitmap formats, the PBM format, which stands for Portable Bitmap (a monochrome bitmap), and PGM format, which stands for Portable Gray Map (a gray scale bitmap). All these formats are not compressed and consequently the files stored in these formats are usually quite large. In addition, the PNM format means any of the three bitmap formats. You may use the unix manual command **man** to learn the details of the PPM format:

```
$man ppm
```

The three bitmap formats can be stored in two possible representations:

1. an ASCII text representation (which is extremely verbose), and
2. a binary representation (which is comparatively smaller).

Each PPM image consists of the following (taken from unix ppm manual):

1. A “magic number” for identifying the file type. A ppm images magic number is the two characters “P6”.
2. Whitespace (blanks, TABs, CRs, LFs).
3. A width, formatted as ASCII characters in decimal.
4. Whitespace.
5. A height, again in ASCII decimal.
6. Whitespace.
7. The maximum color value (*Maxval*), again in ASCII decimal. Must be less than 65536 and more than zero.
8. Newline or other single whitespace character.
9. A raster of *Height* rows, in order from top to bottom. Each row consists of *Width* pixels, in order from left to right. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in pure binary by either 1 or 2 bytes. If the *Maxval* is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first. A row of an image is horizontal. A column is vertical. The pixels in the image are square and contiguous.

10. In the raster, the sample values are "nonlinear." They are proportional to the intensity of the ITU-R Recommendation BT.709 red, green, and blue.

In summary, a PPM file has a header and a body, which may be created using a text editor. The header is very small with the following properties:

1. The first line contains the magic identifier "P3" or "P6".
2. The second line contains the *width* and *height* of the image in ascii code.
3. The last part of the header is the maximum color intensity integer value.
4. Comments are preceded by the symbol #.

Here are some header examples:

Header example 1

```
P6 1024 788 255
```

Header example 2

```
P6
1024 788
# A comment
255
```

Header example 3

```
P3
1024  # the image width
788   # the image height
      # A comment
1023
```

The following is an example of a PPM file in P3 format.

```
P3
# feep.ppm
4 4
15
0  0  0    0  0  0    0  0  0   15  0 15
0  0  0    0 15  7    0  0  0    0  0  0
0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

You can simply use a text editor to create it; for example, copy-and-paste the content into a file named "feep.ppm", which then becomes a PPM file and can be viewed by a browser or the unix utility **xview**. When you execute the unix command,

$ **xview feep.ppm**

you should see a tiny image appear on the upper left corner of your screen along with the following messages,

```
feep.ppm is a 4x4 PPM image with 16 levels
  Building XImage...done
```

4.2 The Convert Utility

Once we obtain an image in PPM format, we can easily convert it to other popular formats such as PNG, JPG, or GIF using the **convert** utility, which is a member of the ImageMagick suite of tools. Conversely, if you obtain an image from other sources in another format, you may also use **convert** to convert it to the PPM format. Besides making conversion between image formats, the utility can also resize an image, blur, crop, despeckle, dither, draw on, flip, join, re-sample, and do much more. It can even create an image from text. We use the unix manual command to see the details of its usage:
    $ **man convert**
We can also run 'convert -help' to get a summary of its command options. The following are some simple examples of its usage.

```
$convert feep.ppm  feep.png
$convert house.jpg house.ppm
$convert house.jpg -resize 60% house.png
$convert -size 128x128 xc:transparent -font \
    Bookman-DemiItalic -pointsize 28 -channel RGBA \
    -gaussian 0x4 -fill lightgreen -stroke green \
    -draw "text 0,20 'Freedom'" freedom.png
```

The last command creates a PNG (Portable Network Graphics) file named "freedom.png" from the text "Freedom". If you want to convert a PDF file to PPM, you may use the utility **pdftoppm**. You may run **"pdftoppm –help"** to find out the details of its usage.

4.3 Read and Write PPM Files

In this section, we present a simple java program that shows how to read and write a PPM file. We will use scalar numeric data types to read or write data to a file. In java, a scalar data type is stored in a variable that is passed by value, meaning that a copy of the data variable is made during the invocation of a method. The scalar numeric data types of java are **byte, char, short, int, long, float,** and **double**. A byte is an 8-bit signed quantity. A char is a 16 bit unsigned quantity. A short, an int, and a long are all signed data types and are 16-bit, 32-bit and 64-bit respectively. The floating point scalar data types of java are **float** and **double**. A float is a signed 32-bit IEEE-1985 floating point quantity while a double is a 64-bit IEEE floating point quantity.

Note that java does not have an unsigned byte data type. So in many cases, we use data type **byte** to read and write the color components of an image as each component only has less than 256 different values. However, when we process the data, we need to convert them to integers, keeping in mind that a negative **byte** value actually means an unsigned value larger than 127; for example, -1 represents 255.

The complete java program **Ppmdemo.java** that demonstrates the reading and writing of PPM files is shown in **Listing 4-1**; the file names and some parameters are hard-coded. We have used the Java Advanced Imaging API (JAI) to simplify some of our coding. The Java Advanced Imaging API provides a set of object-oriented interfaces that supports a simple, high-level programming model which allows images to be manipulated easily in Java applications and applets. As shown in the program, JAI is used to read in a PPM file and render the image on screen; it reads in the RGB components of the image using the method **getPixels**() and save them in the integer array *samples*[]. After rendering the image, the program opens the file "testwrite.ppm". It then outputs the PPM header to the file, converts the integer array *samples*[] to the byte array *bytes*[]. sends the byte array to the file using the method **write**().

After compiling the program with the command "$javac Ppmdemo.java", you may test it with a PPM file using a command similar to the following, which reads the image data from the PPM file "beach0.ppm" and renders the image on screen:

```
$ java Ppmdemo ../data/beach0.ppm
```

Figure 4-1 shows the rendering of such an image using the program. The resulted data are saved in "testwrite.ppm" and you may examine the image using the command "$xview testwrite.ppm". If you want to compare the data of the original file "beach0.ppm" with that of the newly created file, you may use the **diff** command:

```
$ diff testwrite.ppm ../data/beach0.ppm
```

You may find that there's no difference between the files "testwrite.ppm" and "beach0.ppm".

Program Listing 4-1

```
/*
  Ppmdemo.java
  PPM files may either have ASCII or raw (binary) data.
  The decoder automatically determines the data format
  and reads the data accordingly. By default the encoder
  stores the image data in raw format whenever possible.
*/
import java.io.*;
import java.awt.Frame;
import java.awt.image.*;
import javax.media.jai.JAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import javax.media.jai.widget.ScrollingImagePanel;
import com.sun.media.jai.codec.PNMEncodeParam;

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

    /*
     * Create an input stream from the specified file name
     * to be used with the file decoding operator.
     */
     FileSeekableStream stream = null;
     try {
       stream = new FileSeekableStream(args[0]);
     } catch (IOException e) {
       e.printStackTrace();  System.exit(0);
     }

    //First we demonstrate the reading of pnm data

    /* Create an operator to decode the image file. */
    RenderedOp image1 = JAI.create("stream", stream);

    /* Get the width and height of image. */
    int width = image1.getWidth();
    int height = image1.getHeight();

    //allocate array to hold RGB data
    int [] samples = new int[3*width*height];

    Raster ras = image1.getData();
    //save pixel RGB data in samples[]
    ras.getPixels( 0, 0, width, height, samples );

    System.out.printf("Image width=%d, height=%d\n", width, height );

    /* Attach image1 to a scrolling panel to be displayed. */
    ScrollingImagePanel panel=new ScrollingImagePanel(image1,width,height);

    /* Create a frame to contain the panel. */
    Frame window = new Frame("Displaying PPM Data");
    window.add(panel);
    window.pack();    window.show();
    Thread.sleep( 2000 ); //sleep for two seconds
    window.dispose();     //close the frame

    //now, we demonstrate the writing of ppm data
    String filename = "testwrite.ppm";
    try {
      File f = new File ( filename );
      OutputStream out = new FileOutputStream( f );

      //First write PPM header.
      byte [] P6 = { 'P', '6', '\n' };
      out.write ( P6 );
      String s = Integer.toString ( width );
      for ( int i = 0; i < s.length(); ++i )   //write the width
        out.write ( s.charAt(i) );
      out.write ( ' ' );
      s = Integer.toString ( height );
      for ( int i = 0; i < s.length(); ++i )   //write the height
        out.write ( s.charAt(i) );
      out.write ( '\n' );
      //write color levels
      byte [] colorLevels = { '2', '5', '5', '\n' };
      out.write ( colorLevels );

      int size = 3 * width * height;
      byte [] bytes = new byte[size];
      //save the image data
      for ( int i = 0; i < size; i += 1 ){
         bytes[i] = (byte) samples[i];
      }
      out.write ( bytes );
      out.close();
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }
  }
}
```

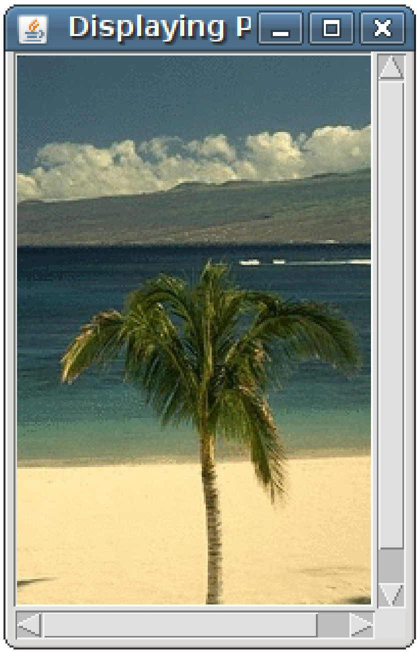

Figure 4-1 Displaying a PPM Image

4.4 Common Intermediate Format (CIF)

There exists a wide variety of ‘standard’ video formats which would lay a heavy burden on a developer to study and understand them for encoding or decoding data saved in their formats. In practice, it is common for a party to use a utility program to capture or convert to a set of standard ‘intermediate formats’ before compressing or transmitting the data. The **Common Intermediate Format** (**CIF**), first proposed in the H.261 standard, is designed for the purpose of standardizing the horizontal and vertical resolutions in pixels of YCbCr video data. CIF allows easy conversions to standard television systems of PAL (Phase Alternating Line) and NTSC (the National Television System Committee). CIF is also known as **FCIF** (Full Common Intermediate Format); it defines a video sequence with a luminance resolution of 352×288 and a frame rate of $30000/1001 (\approx 29.97)$ fps with color encoding using YCbCr 4:2:0. Note that a CIF-image (352×288) consists of 22×18 macroblocks, each of which is a 16×16 pixel block that we shall discuss in Chapter 5. **QCIF**, meaning “Quarter CIF” defines a resolution with frame width and height halved as compared to that of CIF. Similarly, **SQCIF** (Sub Quarter CIF), **4CIF** ($4 \times$ CIF) and **16CIF** define various resolutions with CIF as the basis. Table 4-1 below summarizes these formats.

Table 4-1 Common Intermediate Format

Format	Luminance Resolution (horizontal $\times$ vertical)	Bits / Frame (4:2:0, 8 bits/Sample)
CIF	352×288	1216512
QCIF	176×144	304128
SQCIF	128×96	147456
4CIF	704×576	4866048
16CIF	1408×1152	14598144

The CIF formats do not use square pixels. Rather, they specify a pixel to have a native aspect rate of approximately 1.222:1 because on older television systems, a pixel aspect ratio of 1.2:1 was the standard for 525-line systems. As computer systems use square-pixel, a CIF raster has to be rescaled horizontally by about 109% in order to avoid a “stretched” appearance.

The choice of a particular CIF format depends on the application and available resources like storage and transmission capacity. For example, video conferencing requires real-time transmission of data and its applications commonly use CIF and QCIF that gives fairly good resolution but do not give an overwhelming amount of data. As standard-definition-television has higher transmission bandwidth and DVD-videos are recorded off-line, 4CIF is an appropriate format. For mobile multimedia applications, QCIF or SQCIF are appropriate as the display resolution and transmission bandwidth are limited. Column 3 of Table 4-1 shows the number of bits required to represent one uncompressed frame for each CIF format, where YCbCr 4:2:0 format and 8 bits per luma and chroma sample are used.