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

Fore June

# Chapter 1

# *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 of 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 image’s 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" with the header "P3" aligned to the leftmost margin; it then becomes a PPM file and can be viewed by a browser or the unix utility **display** by ImageMagick. When you

execute the command,

$ **display feep.ppm**

you should see a tiny image appear on the upper left corner of your screen.

## 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 128x32 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". Figure 4-1 shows the image thus created.


**Figure 4-1** Image Created by **convert**

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

To process any PPM and related graphics file, you may use the the **netpbm** library *http://netpbm.sourceforge.net*, which can be downloaded from the Internet. However, for the purpose of this book, we just need something very simple to read or write a PPM file. In this section, we present a simple C program that shows how to read or write a PPM file.

The C/C++ program shown in **Listing 4-1** briefly demonstrates the reading and writing of PPM files; the file names and some parameters are hard-coded; the class **CImage** with public members *red, green*, and *blue* is used to save the color data of one pixel. In a C++ program, a public **class** is the same as a C **struct**.

**Program Listing 4-1** Read and Write PPM Files

```
-------------------------------------------------------------------------
/* ppmdemo.cpp
 * Demostrate read and write of PPM files.
```

```
 * Compile: g++ -o ppmdemo ppmdemo.cpp
 * Execute: ./ppmdemo
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//A public class is the same as a 'struct'
class CImage {
public:
  unsigned char red;
  unsigned char green;
  unsigned char blue;
};

// Create PPM header from image width and height. "P6" format used.
// PPM header returned in integer array ppmh[].
void make_ppm_header ( int ppmh[], int width, int height )
{
  //standard header data, 'P' = 0x50, '6' = 0x36, '\n' = 0x0A
  int ca[] = {0x50, 0x36, 0x0A,                    //"P6"
      //image width=260, height = 288
      0x33, 0x36, 0x30, 0x20, 0x32, 0x38,
              //color levels / pixel = 256
              0x38, 0x0A, 0x32, 0x35, 0x35, 0x0A };

  //only have to change width and height
  char temp[10], k;

  sprintf(temp, "%3d", width );             //width in ascii code
  k = 0;
  for ( int i = 3; i <= 5; ++i )            //replace width
    ca[i] = temp[k++];

  sprintf(temp, "%3d", height );            //height in ascii code
  k = 0;
  for ( int i = 7; i <=9; ++i )             //replace height
    ca[i] = temp[k++];

  for ( int i = 0; i < 15; ++i )            //form header
    ppmh[i] = ca[i];
}

void save_ppmdata (FILE *fp, CImage *ip, int width, int height)
{
  int size = width * height;
  for ( int i = 0; i < size; ++i ){
    putc ( ip[i].red, fp );
    putc ( ip[i].green, fp );
    putc ( ip[i].blue, fp );
  }
}

void ppm_read_comments ( FILE *fp )
{
```

```
  int c;
  while ( (c = getc ( fp ) )  == '#' ) {
    while ( getc( fp ) != '\n' )
        ;
  }
  ungetc ( c, fp );
}

class ppm_error
{
  public:
    ppm_error() {
      printf("\nIncorrect PPM format!\n");
      exit ( 1 );
    }
};

int main()
{
  int ppmh[20];                              //PPM header
  int width = 32, height = 32;               //image width and height
  make_ppm_header ( ppmh, width, height );
  //PPM file for testing read
  FILE *input = fopen("testread.ppm", "rb");
  //PPM file for testing write
  FILE *output = fopen ("testwrite.ppm", "wb");

  //write demo
  for ( int i = 0; i < 15; ++i )             //save PPM header
    putc ( ppmh[i], output );

  CImage image[width][height];
  for ( int i = 0; i < height; ++i ) {       //create a red rectangle
    for ( int j = 0; j < width; ++j ) {
        image[i][j].red = 255;               //red component
        image[i][j].green = 0;               //green component
        image[i][j].blue = 0;                //blue component
    }
  }
  save_ppmdata ( output, (CImage*) image, width, height );
  printf("\nPPM file testwrite.ppm created!\n");
  fclose ( output );

  //read demo
  ppm_read_comments ( input );               //read comments
  char temp[100];
  fscanf ( input, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( input );
  fscanf ( input, "%d", &width );
  ppm_read_comments ( input );
  fscanf ( input, "%d", &height );
  ppm_read_comments ( input );
```

```
  int colorlevels;
  fscanf ( input, "%d", &colorlevels );
  printf("\n%s PPM file: ", temp );
  printf(" \n\twidth=%d\theight=%d\tcolorlevles=%d\n",
              width,height,colorlevels+1 );
  ppm_read_comments ( input );
  while (( c = getc ( input )) == '\n'); //get rid of extra returns
  ungetc ( c ,input );

  //save the data in another file
  CImage ibuf[width][height];
  fread ( ibuf,  3, width * height, input );
  output = fopen("test.ppm", "wb"); //to save PPM data in "test.ppm"
  make_ppm_header ( ppmh, width, height );
  for ( int i = 0; i < 15; ++i )    //save PPM header
    putc ( ppmh[i], output );
  save_ppmdata (output, (CImage*) ibuf, width, height);//save data
  printf("\nPPM file test.ppm created!\n");

  fclose ( input );  fclose ( output );
  return 0;
}
-------------------------------------------------------------------------
```

When you execute the program **ppmdemo**, you should see messages similar to the following displayed.

```
PPM file testwrite.ppm created!

P6 PPM file:
        width=200       height=300      colorlevles=256

PPM file test.ppm created!
```

The program first creates a PPM file named "testwrite.ppm" whose data form a red square. If you view the file with the command **display testwrite.ppm**, you should see a small red square image. The program then reads in the data from the PPM file "testread.ppm" in the *data* directory and prints out its width, height and color levels. Finally, it writes the information to another file named "test.ppm". Again, you can view the image using the command **display test.ppm**.

## 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 the data 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 $352 \times 288$ and a frame rate of $30000/1001 (\approx 29.97)$ fps with color encoding using YCbCr 4:2:0. Note that a CIF-image ( $352 \times 288$ ) consists of $22 \times 18$ macroblocks, each of which is a $16 \times 16$ 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 | $352 \times 288$ | 1216512 |
| QCIF | $176 \times 144$ | 304128 |
| SQCIF | $128 \times 96$ | 147456 |
| 4CIF | $704 \times 576$ | 4866048 |
| 16CIF | $1408 \times 1152$ | 14598144 |

The CIF formats do not use square pixels. Rather, they specify a pixel to have a native aspect ratio 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 give 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.