Chapter 2

Simple Image File Formats

2.1 Introduction

The purpose of this lecture is to acquaint you with the simplest ideas in image file format design, and to get you ready for this week's assignment - which is to write a program to read, display, and write a file in the PPM *rawbits* format.

Image file storage is obviously an important issue. A TV resolution greyscale image has about 1/3 million pixels – so a full color RGB image will contain $3 \times 1/3 = 1$ million bytes of color information. Now, at 1,800 frames (or images) per minute in a computer animation, we can expect to use up most of a 2 gigabyte disk for each minute of animation we produce! Fortunately, we can do somewhat better than this using various file compression techniques, but disk storage space remains a crucial issue. Related to the space issue is the speed of access issue – that is, the bigger an image file, the longer it takes to read, write and display.

But, for now let us start with looking at the simplest of formats, before moving on to compression schemes and other issues.

2.2 PPM file format

The *PPM*, or *Portable Pixmap*, format was devised to be an intermediate format for use in developing file format conversion systems. Most of you know that there are numerous image file formats, with names like *GIF*, *Targa*, *RLA*, *SGI*, *PICT*, *RLE*, *RLB*, etc. Converting images from one format to another is one

of the common tasks in visualization work, since different software packages and hardware units require different file formats. If there were N different file formats, and we wanted to be able to convert any one of these formats into any of the other formats, we would have to have $N \times (N-1)$ conversion programs – or about N^2. The PPM idea is that we have one format that any other format can be converted into and then write N programs to convert all formats into PPM and then N more programs to convert PPM files into any format. In this way, we need only $2 \times N$ programs – a huge savings if N is a large number (and it is!).

The PPM format is not intended to be an archival format, so it does not need to be too storage efficient. Thus, it is one of the simplest formats. Nevertheless, it will still serve to illustrate features common to many image file formats.

Most file formats are variants of the organization shown in Figure 2.1. The file will typically contain some indication of the file type, a block of header or control information, and the image description data. The header block contains descriptive information necessary to interpret the data in the image data block. The image data block is usually an encoding of the pixmap or bitmap that describes the image. Some formats are fancier, some are extremely complex, but this is the basic layout. Also, most (but not all) formats have some kind of identifier – called the *magic number* – at the start, that identifies the file type. Often the magic number is not a number at all, but is a string of characters. But in any case, that is what it is called.

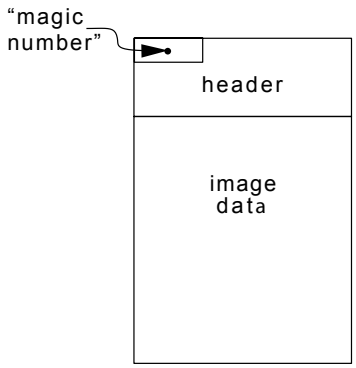

Figure 2.1: Typical Image File Layout

In the PPM format, the magic number is either the ASCII character string `"P1"`, `"P2"`, `"P3"`, `"P4"`, `"P5"`, or `"P6"` depending upon the storage method used. `"P1"` and `"P4"` indicate that the image data is in a bitmap. These files are called PBM (portable bitmap) files. `"P2"` and `"P5"` are used to indicate greyscale images or PGM (portable greymap) files. `"P3"` and `"P6"` are used to indicate full color PPM (portable pixmap) files. The lower numbers – `"P1"`, `"P2"`, `"P3"` – indicate that the image data is stored as ASCII characters; i.e.,

all numbers are stored as character strings. This is a real space waster but has the advantage that you can read the file in a text editor. The higher numbers – `"P4"`, `"P5"`, `"P6"` – indicate that image data is stored in a binary encoding – affectionately known as Portable Pixmap *rawbits* format. In our study of the PPM format, we will look only at `"P6"` type files.

2.2.1 PPM header block

The header for a PPM file consists of the information shown in Figure 2.2, stored as ASCII characters in consecutive bytes in the file. The image width and height determine the length of a scanline, and the number of scanlines. The maximum color value cannot exceed 255 (8 bits of color information) but may be less, if less than 8 bits of color information per primary are available. In the header, all *white-space* (blanks, carriage returns, newlines, tabs, etc.) is ignored, so the program that writes the file can freely intersperse spaces and line breaks. Exceptions to this are that following an end-of-line character (decimal 10 or hexadecimal 0A) in the PPM header, the character `#` indicates the start of a text comment, and another end-of-line character ends the comment. Also, the maximum color value at the end of the header must be terminated by a single white-space character (typically an end-of-line).

```
P6                  -- magic number
# comment           -- comment lines begin with #
# another comment   -- any number of comment lines
200 300             -- image width & height
255                 -- max color value
```

Figure 2.2: PPM Rawbits Header Block Layout

The PPM P6 data block begins with the first pixel of the top scanline of the image (upper lefthand corner), and pixel data is stored in scanline order from left to right in 3 byte chunks giving the R, G, B values for each pixel, encoded as binary numbers. There is no separator between scanlines, and none is needed as the image width given in the header block exactly determines the number of pixels per scanline. Figure 2.3a shows a red cube on a mid-grey background, and Figure 2.3b gives the first several lines of a hexadecimal *dump* (text display) of the contents of the PPM file describing the image. Each line of this dump has the hexadecimal byte count on the left, followed by 16 bytes of hexadecimal information from the file, and ends with the same 16 bytes of information displayed in ASCII (non-printing characters are displayed using the character `!`). Except for the first line of the dump, which contains the file header information, the ASCII information is meaningless, since the image data in the file is binary encoded. A line in the dump containing only a `*` indicates a sequence of lines all containing exactly the same information as the line above.

a) red cube on a midgrey (0.5 0.5 0.5) background

```
000000  5036 0a33 3030 2032 3030 0a32 3535 0a7f  P6!300 200!255!!
000010  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009870  886d 6d92 5959 8a69 6984 7676 817d 7d80  !mm!YY!ii!vv!}}!
009880  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009bf0  7f80 7e7e 9d41 41b5 0909 b211 11a9 2424  !!~~!AA!!!!!!!$$
009c00  9f3b 3b94 5454 8b68 6886 7272 827b 7b80  !;;!TT!hh!rr!{{!
009c10  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
009f70  7f7f 7f7f 7f82 7979 a72b 2bb9 0000 ba00  !!!!!!yy!++!!!!!
009f80  00ba 0000 b901 01b7 0606 b20f 0faf 1d1d  !!!!!!!!!!!!!!!!
009f90  a532 3297 4d4d 8d62 6286 7272 827a 7a80  !22!MM!bb!rr!zz!
009fa0  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
009fb0  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
00a2f0  7f7f 7f7f 7f7f 7f7f 7f8a 6969 b014 14ba  !!!!!!!!!ii!!!!
00a300  0000 ba00 00ba 0000 ba00 00ba 0000 ba00  !!!!!!!!!!!!!!!!
00a310  00ba 0000 ba00 00b9 0505 b60d 0daf 1d1d  !!!!!!!!!!!!!!!!
00a320  a62f 2f9d 4141 915b 5b88 6d6d 8279 7980  !//!AA![[!mm!yy!
00a330  7e7e 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  ~~!!!!!!!!!!!!!!
00a340  7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f 7f7f  !!!!!!!!!!!!!!!!
*
```

b) several lines in a dump of PPM P6 red-cube image file

Figure 2.3: Example PPM P6 Data

2.3 Homework Nuts and Bolts

2.3.1 OpenGL and PPM

Display of an image using the OpenGL library is done most easily using the procedure `glDrawPixels()`, that takes an image pixmap and displays it in the graphics window. Its calling sequence is

```
glRasterPos2i(0, 0);
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, Pixmap);
```

The call `glRasterPos2i(0, 0)` assures that the image will be drawn in the window starting at the lower left corner (pixel 0 on scanline 0). The `width` and `height` parameters to `glDrawPixels()` specify the width, in pixels, of a scanline, and the height of the image, in number of scanlines. The `GL_RGBA` parameter indicates that the pixels are stored as RGBA quadruples, with color primaries stored in the order red, green, blue, alpha, and the `GL_UNSIGNED_BYTE` parameter indicates that each color primary is stored in a single byte and treated as an unsigned number between zero and 255. Finally, the parameter `Pixmap` is a pointer to an array of integers (`unsigned int *Pixmap`) that is used to store the image's pixmap. Since an `unsigned integer` is 32 bits long, each element of the `Pixmap` array has room for four bytes of information, exactly what is required to store one pixel.

Please make special note of the following complications:

1. By default, OpenGL wants the image raster stored from the bottom scanline of the image to the top scanline, whereas PPM stores the image from the top scanline to the bottom.
2. Each 32 bit `int` in the pixmap stores 1 pixel value in the order R, G, B, α. Each of R, G, B, and α take 1 byte. α is an opacity value that you can set to 255 (or `0xff` in hexadecimal), which indicates that the color is fully opaque.
3. PPM stores pixel primaries in R, G, B order, and there is no provision for an α value.

2.3.2 Logical shifts, and, or

Reading a PPM file and displaying it, requires you to pack the red, green and blue information for a single pixel into one 32 bit word. When you have to write the file back out, it is necessary to unpack the red, green and blue components.

There are several ways to do this. One way is to make each color an array of four unsigned characters. Another is to make each color a struct, with four elements, each an unsigned character.

Another way to do this is by making each color an unsigned integer (which is a 32 bit quantity) and shift each of the eight-bit color values into place. However, to do this properly, you need to know the architecture of your machine. If you are using a machine with an Intel-like architecture (like all Windows systems), then the underlying storage is in "Little Endian" order. What this means is that in a four-byte `int`, the least significant part of the binary number is stored in the leftmost byte, and the most significant in the rightmost byte. However, as the `int` is extracted from memory, to be treated as a 32-bit integer number, it is rearranged so that the most significant byte is on the left and the least on the right. Therefore, if you want your color channels to be arranged in the order (R, G, B, α) when they are used on the display, then you need to order them in the order (α, B, G, R) in memory. If you are using a machine with "Big Endian" storage order, like one of the older PowerPC Macintosh computers, then you would use the (R, G, B, α) order. The example code below assumes a "Little Endian" architecture:

```
int red, green, blue;
unsigned int pixel;

/* packing */
pixel = 0xff << 24 | blue << 16 | green << 8 | red;

/* unpacking */
alpha = (pixel >> 24);
blue = (pixel >> 16) & 0xff;
green = (pixel >> 8) & 0xff;
red = pixel & 0xff;
```

The operator `<<`, when used in an arithmetic expression, causes the bits in the memory cell specified to the left of the `<<` symbol to be shifted to the left by the number of positions specified to the right of the `<<` symbol. 0's are shifted into the rightmost end of the cell to fill vacated positions. Likewise, when used in an arithmetic expression, `>>` causes a shift to the right by the specified number of bits, with 0's shifted into the leftmost end of the cell. The operator `|` specifies a *logical or* operation between the value specified to its left and the value specified to its right. Similarly, the operator `&` specifies a *logical and* operation between its left and right operands. Both the logical *or* and *and* operations operate bit by bit between corresponding bit positions in their two operands. A logical *or* of two bits yields a 1 if either of the operand bits is a 1, otherwise it yields a 0. In other words, if either one or the other or both operands are 1 the result is 1. A logical *and* of two bits yields a 0 if either of the operand bits is a 0, otherwise

it returns a 1. In other words, the result is 1 only if the left operand and the right operand are 1. These operations are diagrammed in Figure 2.4.

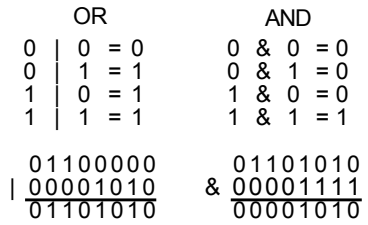

Figure 2.4: Logical **or** and **and** Operations

There are many practical uses of these logical operations, but for our purposes, the most important is that they allow us to do selective operations on groups of bits (*fields*) within a byte or word. Logical *or* allows superposition of fields within a word, and logical *and* allows *masking off* of fields, leaving only the values in the bit positions that have been logically *anded* with 1 bits. Figure 2.5 gives a few examples to show how these logical operations work. The examples use only 8 bits for simplicity, but the same principles hold for a 32 bit quantity.

`unsigned char a = 6;`	a:	00000110
`unsigned char b = 10;`	b:	00001010
`unsigned char c = 106;`	c:	01101010
`unsigned char byte;`	byte:	xxxxxxxx

a) starting values in variables `a`, `b`, `c`, and `byte`

`byte = a << 4;`	byte:	01100000
`byte = (a << 4) | b;`	byte:	01101010
`byte = c & 0x0f;`	byte:	00001010
`byte = (c >> 4) & 0x0f;`	byte:	00000110

b) examples of *shifts*, *or*, *and*

Figure 2.5: Logical *Shift*, *Or* and *And* Operations in C

2.3.3 Dynamic memory allocation

You will not know how big to make the pixmap storage array until you have read the PPM Header information to get the image width and height. Once you have that, in `C++` simply do

```
unsigned int *Pixmap;
```

```
Pixmap = new unsigned int[width * height];
```

or in traditional C simply do

```
unsigned int *Pixmap;
Pixmap = (unsigned int *)malloc(width * height * sizeof(int));
```

to allocate exactly enough space to store the image when it is read in.

Note that Pixmap will just be a big 1D array, not a 2D array nicely arranged by scanline. Thus, your program will have to figure out where each scanline starts and ends using image width.

2.3.4 Reading from the image file

Unless you are quite experienced with C++ I/O facilities for handling files, I recommend that you use the file input/output routines from standard C, rather than using the stream I/O facilities of C++. To make use of them you will have to

```
#include <cstdio>
```

Once you have opened a binary file for reading via

```
FILE * infile;
infile = fopen(infilename, "rb");
```

you can read individual bytes from the file via

```
int ch;
ch = fgetc(infile);
```

Once you have opened a binary file for writing via

```
FILE *outfile;
outfile = fopen(outfilename, "wb");
```

you can write individual bytes to the file via:

```
int ch;
fputc(ch, outfile);
```

Note, in the code above I have used the second argument `"rb"` in `fopen()` and `"wb"` in `fclose()`. The C programming language standard does not recognize the `b` character. However, if you are programming in Visual Studio, and you want to either read data from a binary file or write data to a binary file, you must include the `b` character to indicate that the file is binary. When Visual Studio reads or writes an ASCII formatted file, it treats certain 8-bit configurations as control characters, which affect file reading or writing. Putting the `b` in the `fopen()` or `fclose()` tells Visual Studio to accept all 8-bit configurations and to not check for control characters. You do not have to worry about this in a standard C implementation, using any of the other C compilers, like the GNU system.

2.3.5 C command-line interface

The C command-line interface allows you to determine what was typed on the command line to run your program. It works as follows – declare your `main()` procedure like this:

```
int main (int argc, char *argv[]){
  .
  .
  .
}
```

Unix parses the command line and places strings from it into the array `argv`. It also sets `argc` to be the number of strings parsed. If the command line were:

```
ppmview in.ppm out.ppm
```

then the `argv` and `argc` data structures would be built as shown in Figure 2.6. Thus, the file name of the input file would be given by `argv[1]`, and the output file by `argv[2]`. `argv[0]` contains the name of the program itself.

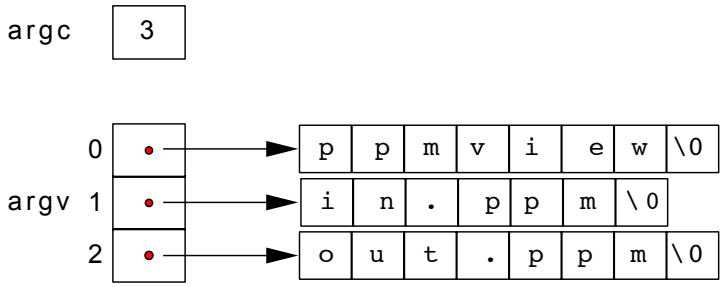

Figure 2.6: argc and argv data structures in C