

Chapter 2

Image File Formats

2.1 Introduction

In this chapter we shall consider the bitmap (.BMP) and audio video (.AVI) binary files and their structure to provide a foundation for the understanding of pixel colour representation and the drawing of the information. We shall not consider every variation (and there are many) of these files but only those, which will be developed through later examples in the area of image processing. The file formats are an essential precursor to the study of image processing methods and this knowledge will also support the work on texturing.

2.2 The Bitmap File

Graphics files are stored in a variety of formats and a full description is given in (Miano 1999). We shall consider the bitmap file format as it is simple to use, easily obtained from scanners and digital cameras, useful in the understanding of audio visual output [AVI] from web cameras and forms a foundation for basic image processing and graphics systems.

Bitmap files are stored with the name extension .BMP and occasionally we also see bitmap files stored in a device-independent bitmap form with the name extension .DIB. Device-independent bitmap files are simply a list of pixels with values for the red, green and blue components and omit the header information associated with size and other descriptors. The bitmap image format originated in early versions of the Microsoft Windows and has stayed ever since through packages such as Paintbrush/Paint. While the Windows operating system supports images with 1, 4, 8, 16, 24 and 32 bits per pixel we shall largely focus on the use of monochrome and 24-bit colour in this text.

2.2.1 *Bitmap-file Structure*

The bitmap file structure is very simple and consists of a bitmap-file header, a bitmap-information header, a colour table, and an array of bytes that define the bitmap image. The file has the following form:

File Header Information
Image Header Information
Colour Table (if present)
Pixel values

BITMAPFILEHEADER

The bitmap file header contains information about the type, size, and layout of a bitmap file and permits a check as to the type of file the user is reading. The first two bytes of the file contain the ASCII character “B” followed by an “M” to identify the file type.

The next four bytes of the header contain the file size with the least significant bit first.

The next four bytes are unused and set to zero.

The final four bytes are an offset to the start of the image pixel data from the header and measured in bytes. Formally the structure is of the form:

```
BITMAPFILEHEADER {  
uint      2 bytes   file type  
dword     4 bytes   file size in bytes  
uint      2 bytes   reserved  
uint      2 bytes   reserved  
dword     4 bytes   offset to data in bytes  
} BITMAPFILEHEADER;
```

BITMAPINFOHEADER

The bitmap-information header specifies the dimensions, compression type, and colour format for the bitmap.

The first four bytes are the header size, usually 40 bytes, followed by the width and height of the image measured in pixels. The next two bytes contain 1 which is the number of planes. The next two bytes store the number of bits used to represent the colour intensities of a pixel, which in this text is usually 24 (referred to as true colour) as we frequently use such images. Twenty-four bit colour has over the years become more prevalent as memory has become cheaper and processor speeds have increased. The next four bytes store the compression (0 for 24 bit RGB) followed by the Image size (may be 0 if not compressed). The next eight bytes store the *X* and *Y* resolution (pixels/meter). The final entries in the bitmap information section are the

number of colour map entries and the number of significant colours. Formally this is written as:

```
BITMAPINFOHEADER {
dword   4bytes   needed for BITMAPINFOHEADER structsize
long    4 bytes  bitmap width in pixels
long    4 bytes  bitmap height in pixel
word     2 bytes  1
word     2 bytes  bits/pixel (1 = monochrome)
dword   4 bytes  compression 0, 8, 4
dword   4 bytes  image size in bytes (may be 0 for monochrome)
long    4 bytes  pixels/metre
long    4 bytes  pixels/metre
dword   4 bytes  number of colour indexes used by bitmap in colour table
dword   4 bytes  number of colour indexes considered important
} BITMAPINFOHEADER;
```

COLOUR TABLE

The colour table is not present for bitmaps with 24 bit files because each pixel is represented by the 8-bit blue-green-red (BGR) values in the actual bitmap data area.

IMAGE DATA

The bitmap data immediately following the colour table consist of BYTE values representing consecutive rows (scan lines) of the bitmap image in left-to-right order. A scan line must be zero-padded to end on a 32-bit boundary or rounded up to a multiple of four bytes. The scan lines in the bitmap are stored from bottom to the top of the image. This means that the first byte in the file represents the pixels in the lower-left corner or origin of the bitmap and the last byte represents the pixels in the upper-right corner.

The format of the file depends on the number of bits used to represent each pixel with the most significant bit field corresponding to the leftmost pixel. Details of formats using less than 24 bits can be found in the references given.

2.2.2 A Typical File Layout

The following file header information used for illustrative purposes is a 512×256 bit map image using 24-bit colour. The file type indicators of 'B' and 'M' are in the first two file locations. The file size is calculated from the double word of the next four bytes with the least significant byte occurring first (393,270 bytes). The author notes that this value can be found using the old 'dir' command in the DOS command window where Windows explorer will only give an approximation to this file size. The file size can be alternatively verified from the image size, bytes per pixel and the header offset.

$512 \times 256 \times 3 + 54 = 393270 \text{ bytes}$

66	B	Information Header	24	Bit colour
77	M		0	
54		File size	0	
0		”	0	
3		”	0	
0		”	0	
0			0	
0			0	
0			6	
0			0	
54		Offset to data from start of header	35	Resolution pixels / metre
0			11	
0			0	
0			0	
40		Image Header Size from here	35	Resolution
0			11	
0			0	
0			0	
0			0	
2		Image width (512 pixels)	0	
0			0	
0			0	
0			0	
1		Image height (256 pixels)	0	
0			0	
0			0	End of Header
1			245 234 212	Start of image pixel data
0			245 234 212	

2.2.3 Reading and Drawing a Bitmap File

The following program is for the purpose of understanding pixel data and 24 bit binary images. Later on we shall generalise the reading of the headers with the introduction of classes and functions to simplify the repetition of code but for now we will hard code some values for simplicity in understanding of the file structure. Many of the examples in the text together with data files are given on the web site supporting the book.

The image data file header is given in the preceding section where we know both the width and height of the image. This can be found from many Windows applications such as Paint in the Image Attributes part of the menu. Because most graphics and imaging applications will require the use of all bits in representing colours we characterise the input data files as binary and define the bytes for storing the red, green and blue (RGB) pixel colour components as unsigned bytes. Further

we can skip the first 54 bytes, as we know this is defined for the header information of bitmap files, although in a real system we would check the various components of the header for verification of file type and the reading of the rest of the file. The size of the image is (512×256) pixels. Each byte of the image data is read as an *unsigned character* as all bits contribute to the colour level and were converted to integers to afford explanation of the file structure in the previous section and output to the file debug.txt. Pixels would be plotted as green-blue-red at each position on a scan line and there are 3 bytes per pixel for the 24-bit colour image.

```
void listing (void) {
unsigned char pixel;
int x, y, xs, xf, ys, i, pix, xpack;
ofstream debug("c:\\rob\\graphics\\lectures\\book\\programs
              \\debug.txt", ios::out);
ifstream inpaint( "c:\\rob\\images\\earth512256.bmp",ios::binary);
if(!inpaint) {
    cout << "File not opened\n";
    exit(1);
}
for(x=1; x<=54; x++) {
    inpaint.get(pixel);           //go to start of data after header stuff
    pix = pixel;
    debug << pix << endl;       //file data for explanation
}
debug << "End of Header?" << endl;
xs = 1;
xf = 512;                        //will get x&y from header later
xpack = xf % 4;                  //multiples of 4 bytes/x line?
ys = 0;
for (y = ys; y<= 255; y++) {
    for (x = xf; x>= xs; x--) {
        for (i=0; i<=2; i++) { //3 bytes = 1 pixel in 24 bit colour
            inpaint.get(pixel);
            pix = pixel;
            debug << pix << " ";
        }
        debug << endl;
    }
}
if (xpack != 0) for (i=1; i<=xpack; i++) inpaint.get(pixel);
}
// next raster line y
}
```

In this program example, output and input was via disc files and console screen. We are now going to modify the program to provide graphical output to gain a clearer understanding of the data in the file. Later on we shall provide a fuller explanation of OpenGL graphics but for the present we shall use a very simple subset of facilities in order to get started and further understand bit map files.

We can extend the file-listing program and plot the pixel colours at each location of the image using the included program (BMcol.cpp). The setting up, management and refreshing of the graphical window where the image is drawn is performed through using the GLUT library described in the previous chapter and using system defaults to reduce the number of functions for purposes of clarity.

```
//Software to recreate 24 bit image from file

GLsizei wh = 256; // height of window
GLsizei ww = 512; // width of window
void MyInit (void) {
    glPointSize(1.0);
    gluOrtho2D( 0.0, (GLdouble)ww, 0.0, (GLdouble)wh );
}
void paint (void) {           //read and draw a bit map file
    unsigned char pixel;
    GLint x, y, xs, xf, ys, i, pix, xpack;
    GLfloat pixtmp[3];
    ofstream debug( "c:\\rob\\graphics\\lectures\\book\\programs
                    \\debug.txt", ios::out);
    glClear (GL_COLOR_BUFFER_BIT );
    ifstream inpaint( "c:\\rob\\images\\earth512256.bmp", ios::binary);
    //w=640
    if(!inpaint) {
        cout << "File not opened\n";
        exit(1);
    }
    for(x=1; x<=54; x++) {
        inpaint.get(pixel);    //go to start of data after header stuff
        pix = pixel;
        debug << pix << endl; //file data for explanation purpose
    }
    debug << "End of Header?" << endl;
    xs = 1;
    xf = 512;                //should get x & y from header really
    xpack = xf%4;            //multiples of 4 bytes/x line
    ys = 0;
    glBegin(GL_POINTS);
    for (y=ys; y<= 255; y++) {
        for (x = xf; x>= xs; x--) //earth_r.bmp
            for (i=0; i<=2; i++) { //3 bytes = 1 pixel in 24 bit colour
                inpaint.get(pixel);
                pix = pixel;
                debug << pix << " ";
                pixtmp[i] = float(pix)/255.0;
            }
        debug << endl;
    }
}
```

```

        glColor3f(pixtmp[2], pixtmp[1], pixtmp[0]); //G B R !
        glVertex2i(x,y);
    }
    if(xpack != 0) for (i=1; i<=xpack; i++) inpaint.get(pixel);
    } //next raster line y
    glEnd();
    glFlush();
}

```

The Program (BMcol.cpp)

```
int main(int argc, char **argv) {
```

Within our main routine the GLUT code

```
glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB );
```

specifies the type of display used in the graphical window to be created. In this case the display is using RGB colour mode (GLUT_RGB) and all graphics is performed in the window; single buffered (GLUT_SINGLE).

2.2.4 Creating a Display Window

The window is created with the code

```
glutInitWindowSize ( ww, wh );
glutCreateWindow ("24 bit BMP display");
```

and has the title in the top bar of the window with the size mapping directly to the size of the image. The function `MyInit ()`; initialises the pixel drawing size and `gluOrtho2D(0.0, (GLdouble)ww, 0.0, (GLdouble)wh)`; creates a matrix for projecting the bitmap onto the screen clipped by the region (0.0, ww, 0.0, wh).

The image of the bitmap file (earth512256.bmp) is reconstructed by the function `paint()`; This function is called by GLUT when the window is drawn.

```
glutDisplayFunc( paint );
```

and is referred to as a ‘callback’. The final GLUT function used is

```
glutMainLoop( );
```

which initiates the GLUT system display processes. For now that’s all we need to know about the display window.

2.2.5 Creating the Image for Display

We now have to read the binary bitmap and transfer the byte colour values to appropriate locations in the display window. Each pixel position is defined by a point with three bytes contributing to the resultant colour of the location. We plot one scan line (y) at a time and read the three bytes that contribute to the colour at each point (x) along the scan line. The colour is given by the green, blue and red components represented as floating point numbers using the code

```
glColor3f(pixtmp[2], pixtmp[1], pixtmp[0]); // G B R
glVertex2i(x,y);
```

at each (x, y) integer location. The colours can have 256 intensity levels stored in each byte and if we use floating-point representation, then this must be normalised between 0 (off) and 1 (maximum intensity). At the end of each scan line we check that we end on a 32-bit boundary and read dummy bytes out if required. Pixel plotting takes place using

```
glBegin(GL_POINTS);
glEnd();
```

to delimit the operations. The resultant image is shown in Fig. 2.1.

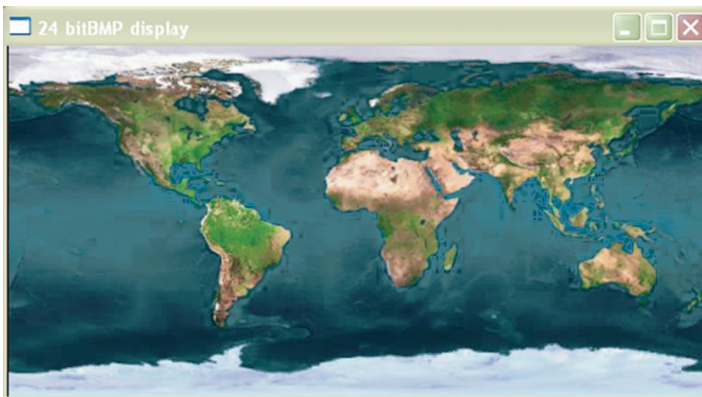


Fig. 2.1 Displaying a bit-map file

2.2.6 Monochrome Bitmaps

We shall briefly consider the monochrome bitmap since it is useful in the storage of line drawings and printed text where colour is not a requirement. The

BITMAPFILEHEADER and the **BITMAPINFOHEADER** are as already described with only the change in the information header where the bits / colour is now 1 as only black and white is present. Since we are working with a monochrome image each pixel can be represented by 1 bit and each byte will store eight possible locations along each raster line. The code following is for an 800×600 pixel image where the colour black is represented by zero. If the whole byte is zero then we can draw a black line eight pixels long. If the byte is not zero indicating it is part of the edge of a shape or dots, then we must plot the individual bits that are zero that make up the byte. We accomplish this by using modulo 2 arithmetic; to test each bit position in the byte and plot depending on the value of the remainder. The code following, replaces the equivalent code in the paint function used to draw the 24-bit colour image already described.

```
for (y = 1; y<= 600; y++) {
    nx = 0;
    for (x = 1; x<= 100; x++) {
        //100 bytes = 800 pixels in monochrome BMP
        inpaint >> pixel;
        pixtmp = pixel;    //char to integer value
        nx = nx + 8;
        if (pixtmp == 0) { //8 pixel line
            glBegin(GL_LINES);
            glVertex2i(nx-8,y);
            glVertex2i(nx,y);
            glEnd();
        }
        else{
            //whole byte is not black (>0 & < 255) near edges of image | spots etc!
            for (i = 1; i<= 8; i++) {
                remain = pixtmp % 2;
                if (remain == 0) { // black - plot it
                    glBegin(GL_POINTS);
                    glVertex2i(nx-i,y);
                    glEnd();
                }
            }
            pixtmp = pixtmp / 2; //next pixel in byte
        }
    }
} //next raster line y
```

2.2.7 A General Class for Input of Bit Map Files

We will now provide a class for the input of 24 bit map files used throughout this book, as it will save on space by repetitively using the same code where such files

are required. The file (Read_bmp.cpp) is referenced in later programs by a #include "Read_bmp.cpp" throughout the text. The only input required to the class is the name of the bit map file and it will be read and stored in an array identified as a global variable by

```
static GLubyte Image[Height][Width][4];
```

where the RGB components of each pixel are captured as unsigned bytes. The class reads the header, calculates the width (ww) and height (wh) of the picture using binary to decimal conversion and the values to read the bytes into the above array. The function WidthHeight() converts the high and low pixels from the file to integers for defining the size of a window.

```
class ip_bmp_file
{
public:

long int WidthHeight(int pix1, int pix2) {
long int num=0, n=8 ,rem;
do {    //high byte only
    rem = pix2 % 2;
    num = num + rem * pow(2, n);
    pix2 = pix2 /2;
    n++;
} while(n <= 16);
num = num + pix1;
return(num);
}

void header_data ( void) {
unsigned char pixel;
char filename[25];
int i, j,x, pixlow, pixhigh, pix3, pix4, xpack, BMok;
cout << "Picture file name: ";
cin >> filename;
ifstream inpaint( filename, ios::binary);
if(!inpaint) {
    cout << "File not opened / found\n";
    exit(1);
}
BMok = 0;
inpaint >> pixel;
if(pixel == 'B') BMok++;
inpaint >> pixel;
if(pixel == 'M') BMok++;
```

```

if(BMok == 2) cout<<"is a valid bit map file\n";
    else exit(1);
for(x=3; x<=54; x++) {
    inpaint >> pixel; //go to start of data after header stuff
    if(x == 19) pixlow = pixel; //low
    if(x == 20) pixhigh = pixel; //high
    if(x == 23) pix3 = pixel; //low
    if(x == 24) pix4 = pixel; //high
}
xw = WidthHeight( pixlow, pixhigh);
yh = WidthHeight( pix3, pix4);
cout << "\nx width = " << xw << "y height = " << yh << endl;
xpack = ww % 4; // multiples of 4 bytes/x line
for (j = 0; j < yh; j++) {
    for (i = 0; i < xw; i++) {
        Image[i][j][0] = inpaint.get( );
        Image[i][j][1] = inpaint.get( );
        Image[i][j][2] = inpaint.get( );
    }
    if (xpack != 0) for (i=1; i<=xpack; i++) inpaint.get(pixel);
}
}
};

```

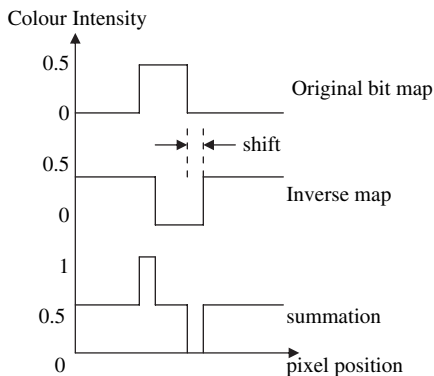
Readers may like to generalise this class to include tests for the file type and output details such as file size.

The fourth component of each pixel location is set to 255 and initialises what is called the alpha channel and will be considered further when we come to consider texturing in Chapter 7.

2.2.8 Manipulating Bit Maps – Embossing

The image given in Fig. 2.1 is a typical picture of the world, which looks rather flat and uninteresting. We can provide the illusion of a three dimensional view through the simple method of embossing, by superimposing the original bit map image with an inverted version of the map that is slightly shifted or displaced in spatial coordinates. This produces an enhanced lightening a darkening at colour discontinuities that then appears to show highlights and give the impression of protrusions to a surface. For our example we shall use a bit map image of a brick wall and the colour levels (0–255) will be used to simulate a ‘height’ map at each location of the image. Using floating-point representation the corresponding colour values (0.0–1.0) are clamped between 0.0 and 0.5. We now make a second map of pixel values after subtracting the clamped values from 0.5. This in effect inverts the original colour

Fig. 2.2 Summation of original and inverse bit map



levels so that black becomes white and vice-versa. The final stage of the process is to add the original clamped image to the inverse with a small positional shift, which gives the effect of moving the inverse map slightly towards a light source in relation to the original image. Another way to view this operation is that of finding the first derivative of the image intensity. The addition of the pixel intensities after the shift in x and y of the inverse map is shown in Fig. 2.2. The highlight and shadow effect is seen in the pixel intensity after the summation in Fig. 2.2.

The number of pixels specified for the shift should not be too large or distortions of the image occur when parts of the object (mortar joints in this case) are missed. The method averages the summation about 0.5 and thus the embossed result is a grey level image with highlights at the colour discontinuities as shown in Fig. 2.3.

The lower part of Fig. 2.3 is a plain bit map image of a brick wall, while the upper part of the image results from the embossing process with a shift of two pixels in both the x and y directions for each pixel. Implementation of this procedure is supplied through the program `emboss.cpp` and readers can experiment using the data file `daniel.bmp` and consider results for the embossing of letters. Similar results can be obtained by performing the operations separately on each RGB component as illustrated in the right hand image of Fig. 2.3.



Fig. 2.3 Embossing using a bit map of a brick wall

2.3 Audio/Video Interleaved (AVI) File Format

The AVI file format originated at the Microsoft Corporation and conceptually is a very flexible and simple structure for the storage and display video data and reproduction of sound from audio information. The format has continued to evolve over many years and what is developed in this section is but a small part of a very much larger system definition. Readers are cautioned that writing software for applications dependent on the large variety of AVI devices available is not always as straightforward as the file format definitions might imply. A number of different codes exist for the same field and this can be confusing; in writing software the collecting of all the codes can be time consuming and many are only documented on web sites of frequently asked questions! Further, files of the same type may have different information interspersed between the audio and video data dependent on device manufacturer. The files count16.avi and capture.avi used in developing AVI displays later in this section are provided to illustrate the issue. The following examples are designed to give readers an understanding and starting point, from which they can develop image-processing applications.

The AVI file format is part of what is known as the Resource Interchange File Format (RIFF) and is used for multimedia devices such as CDROM's, and Web cameras. We shall only be concerned with graphics/video files although mention will also be made of the sound component of the file. RIFF is a binary file *framework* within which existing and different file formats may be embedded and nested as data structures (chunks) *with no fixed positions* in the file. This results in a flexible vendor controlled file where one reads data according to the four character code chunk name. The data stored in a RIFF file is indicated by the file extension: Interleaved audio/visual (.AVI), Sound (.WAV), MIDI information (.RMI) etc.

The following is a typical example of the beginning of a RIFF file with the four character code (fcc) headers from a web camera producing frames of 160 × 120 with the three primary colours stored in 16 bits and read using a modification of the FileHeader.cpp program previously described. The user needs to check for the character sequences outlined above and use the appropriate structure/class to define the reading of the file.

82 R	0	0	0	
73 I	0	202 Ê	0	
70 F	104 h	8 €	1	//video
70 F	100 d	0	0	//only
32 //file size	114 r	0	0	
206 Î // in bytes	108 l	0	0	
1 €	97 a //avi	0	0	
0	118 v //header	0	0	
65 A	105 i	16 € //flags	0	
86 V	104 h	0	0	
73 I	56 8 //size	1 €	160	//width
32	0	0	0	

```

76 L          0          2 €    //no of  0
73 I          0-----a    0    //frames  0
83 S          160         0          120 //height
84 T          134 †        0          0
92 \  //size    1 €        0    //no inter 0
1 €          0          0    //leave  0
0            0          0          0
0            0          0          0
0            0          115 s        0
0            0          116 t        118 v
0            0-----a    114 r        105 i
0            76 L          108 l        100 d
0            73 I          115 s    //stream 115 s
0            83 S          116 t    //header 0
0            84 T          114 r        0
0            124 |  // size    104 h        0
0            0            64 @    //size  0

```

The general form of the RIFF chunks is:

```

struct Chunk
{
DWORD ChunkID;           //4 character identifier (fcc)
DWORD ChunkSize;         //in bytes
BYTE ChunkData[ChunkSize]; //word aligned with NULL byte
};                       //if odd (not in ChunkSize) pad with NULL

```

The RIFF identifier is the top-level chunk and the rest are sub chunks in an AVI file.

Chunks are identified by a 4 character name (RIFF, AVI, WAV, LIST, hdrl, avih, strl, strh, strf, ISFT, IDIT, JUNK, movi etc and if 3 characters then a NULL byte is added before the size which follows the chunk identifier. The size is the length of chunk in bytes (stored as a double word) and does not include any padding that may have been required. The data component is word aligned and a NULL byte is added if the data is of an odd byte length. Software to read a file requires we search through for the chunk types noting the length of what follows the chunk identifier. The JUNK chunk is used to pad out a file to fit special boundaries that exist on devices such as CD-ROM's (2048 byte boundaries). Each chunk is followed by a data length indicator and the data. The top level RIFF identifier is followed by the total file size less any padding bytes.

2.3.1 *Chunk Hierarchy*

The list of structures are of the following general form

```

RIFF - AVI                                //An AVI file
    LIST - hdlr                            //Header list
        Avih                              //main AVI header
    LIST - strl                            //video stream list
        strh                              //video stream header
        strf                              BITMAPINFOHEADER
        strd

JUNK

    LIST movi                              //main data
        00db                              //Video frame
        .....
        idx1                              //Index

```

The first LIST chunk contains the AVI header (hdlr) chunk followed by the characters ‘avih’ and a structure of the form

```

structretypedef struct {
    DWORD dwMicroSecPerFrame;    //Time between frames
    DWORD dwMaxBytesPerSec;      //AVI data rate
    DWORD dwReserved1;
    DWORD dwFlags;               //type of data parameters
    DWORD dwTotalFrames;         //Number of frames
    DWORD dwInitialFrames;       //preview frames
    DWORD dwStreams;             //Number of data streams in chunk
    DWORD dwSuggestedBufferSize; //Minimum playback buffer size
    DWORD dwWidth;               //frame width in pixels
    DWORD dwHeight;              //frame height in pixels
    DWORD dwScale;               //time units
    DWORD dwRate;                //playback rate
    DWORD dwStart;               //Start time of AVI data
    DWORD dwLength;              //Size of AVI data
} MainAVIHeader;

```

with the meaning of each variable given in the comment. The hdlr AVI header also contains one or more LIST chunks with the strl four character identifier. Each data stream will be associated with such a LIST chunk (1 for video, 2 for audio and video).

The second LIST chunk contains the AVI header (strl) chunk followed by the characters ‘vids’ or ‘auds’. Three sub chunks are stored within the strl chunk: the stream header strh, the stream format strf and optionally another stream data

chunk strd. The length of the structure less any padding bytes follows each of the character identifiers.

The following structure defines the stream header strh:

```
typedef struct {
    DWORD fccType;           //vids = video, auds = audio
    DWORD fccHandler;        //compressor used
    DWORD dwFlags;
    DWORD dwReserved1;
    DWORD dwInitialFrames;   //if interleaved no of preview frames
    DWORD dwScale;           //playback stream characteristics
    DWORD dwRate;
    DWORD dwStart;
    DWORD dwLength;
    DWORD dwSuggestedBufferSize;
    DWORD dwQuality;
    DWORD dwSampleSize;
} AVIStreamHeader;
```

The stream format (strf) is of the form used for storing bitmap files and contains the following structure

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;            //40
    LONG biWidth;            //image width
    LONG biHeight;           //image height
    WORD biPlanes;           //1
    WORD biBitCount           //bits / pixel 1,4,8,16,24,32
    DWORD biCompression;     //0,1,2,3, CRAM etc
    DWORD biSizeImage;
    LONG biXPelsPerMeter;     //resolution
    LONG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

For video this is the same as the BITMAPINFO structure described for bit maps. For audio streams the data is defined in the WAVEFORMATEX or PCMWAVE-FORMAT structure (<http://msdn2.microsoft.com/en-us/library/ms712832.aspx>).

The biBitCount specifies the number of bits used to illuminate a pixel and must be used in conjunction with the biCompression. The values of biCompression govern how the colour components are extracted from biBitCount.

BiCompression	BI_RGB	BI_RLE8	BI_RLE4	BI_BITFIELDS
	0	1	2	3

This field may also contain the character sequence ‘CRAM’ for BI_RLE8 from earlier systems. Pixel format depends on the number of bits per pixel. For single bit pixels each byte has eight fields, 4-bit pixels have two fields per byte while 8-bit pixels use the complete byte for the colour.

In the example which we will use to illustrate the display of AVI files on a frame by frame basis we shall use a web cam file where each pixel is represented by a 2-byte integer and BI_RGB = 0. With this definition the 16 bits are divided into three 5-bit colour components with the sixteenth bit being ignored. True colour uses 24 bits per pixel with one byte storing each of the RGB components and we shall also present this facility.

The optional stream data chunk strd may follow the stream format structure. The information is not needed to read and display AVI files and generally contains configuration information from the vendor.

Finally before we encounter the RGB colour values that make up each frame we may find a JUNK or other named chunks used to pad out data to device specific boundaries. We need to read past this information when we encounter the characters ‘JUNK’ in the file. The JUNK structure is of the form

```
typedef struct {
    DWORD   JunkID;           //four characters JUNK
    DWORD   JunkSize;        //Padding Size bytes
    BYTE    Padding[JunkSize];
} JUNKHeader
```

The final component is the movi chunk, which contains the image colour data for each pixel. Uncompressed RGB data is identified by four characters ‘***db’ (abbreviation for DIB) followed by the size in bytes of the chunk. The ‘***’ represents the stream identifier (00 for the first hdlr). Both the JUNK and ‘db’ chunks may be found in video and audio data and must be tested for and separated from the image bytes, which contribute to the display.

In general when processing the image data readers are advised to search for any possible four character headers. It is to be noted that the supplied files (capture.avi and count16.avi) have quite different information stored between each frame and in the authors experience this is device manufacturer dependent. The situation where the number of frames as indicated in the file header (BITMAPINFOHEADER) is not the same as the image data sometimes occurs due to frames not being captured because the computer was busy. The file capture.avi was included to demonstrate the effect of lost frames. Attention to detail in searching, correcting removing non-image data is essential for image processing applications.

2.3.2 An Example of Web Cam Display

The Web Camera is a low cost device for image capture which students can use as an introductory tool for image processing. Readers of this text will be aware of a

number of systems such as the ‘Real Player’ which display AVI files and in this section we explore some of the construction that went into such software to provide frame by frame access for image processing applications. Data from cameras is provided via a USB port and stored as bytes in a binary file. A small file of half a dozen frames (capture.avi) is supplied on the Web site to aid the development of this example. The file was collected with BI_RGB set to zero and 16 bits per pixel location, giving 5-bit colour resolution for each RGB component. A second file (capframe.avi) containing two frames in 24-bit colour is also supplied for test purposes while a third test data file count16.avi came with my XP system.

Extraction of the 5-bit components is achieved by setting each byte to an unsigned integer so as not to omit the sign bit in the low order byte and finding the binary representation of each by repeated division by 2.

```
struct bytecolour          //RGB colour components (0->1)
{
    float r;
    float g;
    float b;
};

bytecolour byte2int( unsigned char bchar[2] ) {
int  rem[16], i, i1, i2, j, ip, byte2;
unsigned int value;
double temp[3];
bytecolour subytecol;
bytecolour *addrsbcol;
addrsbcol = &subytecol;
value = int(bchar[0]);          //get sign bit also!
for(i=0; i<16; i++) rem[i] = 0;
byte2 = 0;
i = 0;
while(byte2 < 2) {              //binary values of each byte
    while( value != 0) {         //rem[0] m sig bit
        rem[i] = value % 2;
        value = value / 2;
        i++;
    }
    i = 8;
    byte2++;
    value = int(bchar[1]);      //now the second byte
}
i1 = 0;          // red = rem[14-10] green = rem[9-5] blue = rem[4-0]
for(j = 0; j < 3; j++) {      //colour components
    i2 = i1 + 5;
    temp[j] = 0.0;
    ip = 0;
    for(i=i1; i<i2; i++) {
```

```

        temp[j] = temp[j] + double(rem[i]) * pow(2.0, double(ip));
        ip++;
    }
    i1 = i1 + 5;          // next 5 bits for next colour
}
addrsbcol->r = temp[2]*0.03125;
addrsbcol->g = temp[1]*0.03125;    // levels 1/32 = 0.03125 for 5 bits
addrsbcol->b = temp[0]*0.03125;
return (subytecol);
}

```

The function `byte2int()` takes as argument two bytes read from a file generated by the web camera and returns through the structure definition of `bytecolour`, the three RGB colour components. Readers should note the order of the byte colours as this changes with different compressions.

Software to display an AVI file now requires two further components; the decoding of the RIFF header and the search and removal of any four character chunks within the display data. Formally AVI display software is described in Fig. 2.4.

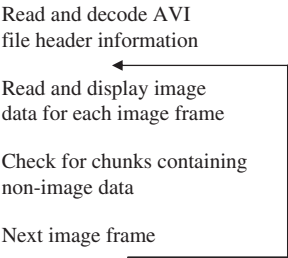
Display only takes place for data, which follows the four character code 'oodb', and the following four bytes, which indicate the size in bytes of the data. The display data comes in different forms defined by the `biBitCount` field of the `BITMAPINFO-HEADER` structure. The 5-bit colour extraction from two bytes has already been described. For 24-bit colour extraction the issue is simpler in that we used three consecutive bytes to represent the RGB values. Code fragments for the two conditions is as follows

```

//16 bit display
if(biBitCount == 16) {
    if(dbcharcount != 4 && jcharcount != 4) { //junk & db fcc's ok?
        inpaint.get(chari[3]);
        for (j=0; j<2; j++) {                //get RGB
            for (i=0; i<2; i++) pixels8[i] = chari[i+j*2];
            bytecols = byte2int( pixels8 );
            red = cols->r;
            green = cols->g;
            blue = cols->b;
            glColor3f(red, green, blue );
            glVertex2i(x,y);
            x++;
        }
    }
}
// 24 bit colour
if(biBitCount == 24) {
    if(dbcharcount == 3) { // first 3 of db fcc ok?
        inpaint.get(pixel); // b
    }
}

```

Fig. 2.4 Decoding AVI files



```
if(pixel == 'b') {
    dbcharcount++;
    chari[3] = pixel;
}
}
if(dbcharcount != 4 && jcharcount != 4) {
    glColor3ub(chari[0], chari[1], chari[2]);
    glVertex2i(x,y);
    x++;
}
}
```

and a complete program (AVIASCH2.cpp) together with the AVI test files is provided on the accompanying Web site.

2.3.3 Compression

The preceding example is software that is ‘in progress’ and provides a basis for readers to start further work. A useful student exercise may be based on the discussion in this section. The bitmap format supports run-length encoding (RLE) of four and eight bit per pixel images and implementation of this facility will expand the range of files that can be processed.

RLE is a form of encoding suitable for images where large numbers of pixels are all the same as might occur for instance in an engineering drawing. In this case the background paper colour is white with black lines making up the geometry of an object and type written text aiding the description. Considering the image on a line by line basis we can represent a line by alternate counts of the numbers of white and black pixels as we move from left to right along the line as illustrated in Fig. 2.5 where

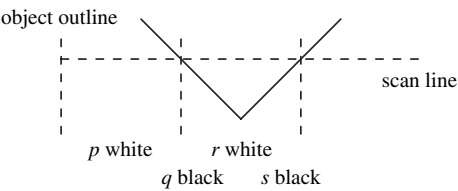


Fig. 2.5 Black and white pixels along a scan line

p , q , r and s are the corresponding number of black or white pixels that contribute to the image along the scan line. The applicability of the method depends upon the nature of the image concerned, where for example an image exhibiting a lot of colour variation or detail will provide little compression.

RLE8

This form of run length encoding is stored in two-byte pairs where the second byte gives the value of the pixel intensity to be repeated by the number of times stored in the first byte. For an eight bit pixel image

$0A_{16} \ 66_{16}$ represents $66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16} \ 66_{16}$

A 00_{16} in the data indicates a new line and follows at the end of each scan line.

2.4 Exercises

1. Using the program AVIASCI2.cpp as a starting point integrate software to display files where 4 and 8-bit RLE encoding has been performed.
2. Using the programs provided for display of bit maps, provide additional mouse interaction (see Chapter 3) to develop a freehand drawing system.

2.5 Conclusion

In this chapter we have described two commonly available file structures, which are widely used in texturing (see Chapter 7), for imaging (see Chapter 3) and for data capture. Understanding the bitmap file structure is a prerequisite for image processing systems where we will filter and enhance images base on individual pixel distributions. Pixel maps in OpenGL generally require a size in x and y to be a power of 2 and it is convenient to scale all images to this size prior to processing. We have processed images using 24-bit and 16-bit colour demonstrating the division of two bytes into three 5-bit colour components.

Building on our knowledge of bitmaps we describe the AVI file structure and this understanding will facilitate later activities in image processing. Because this is a real time capture system, the frames have been separated so that readers will be able to perform tracking and timing measurements from images. It is not the purpose of this text to develop the processing of the audio component of AVI files. Where such data occurs it has been separated out and for those interested in these 'WAV' parts of the file, it may be processed separately.



<http://www.springer.com/978-1-84800-022-3>

OpenGL Graphics Through Applications

Whitrow, R.

2008, XV, 330 p., Softcover

ISBN: 978-1-84800-022-3