[Previous (ch03_03.htm)] [Next (ch03_05.htm)]

Bitmap Data

The actual bitmap data usually makes up the bulk of a bitmap format file. In the following discussion, we'll assume that you've read Chapter 2, *Computer Graphics Basics* (ch02_01.htm), and that you understand about pixel data and related topics like color.

In many bitmap file formats the actual bitmap data is found immediately after the end of the file header. It may be found elsewhere in the file, however, to accommodate a palette or other data structure which also may be present. If this is the case, an offset value will appear in the header or in the documentation indicating where to find the beginning of the image data in the file.

One thing you might notice while looking over the file format specifications is the relative absence of information explaining the arrangement of the actual bitmap data in the file. To find out how the data is arranged, you usually have to figure it out from related information pertaining to the structure of the file.

Fortunately, the structuring of bitmap data within most files is straightforward and easily deduced. As mentioned above, bitmap data is composed of pixel values. Pixels on an output device are usually drawn in scan lines corresponding to rows spanning the width of the display surface. This fact is usually reflected in the arrangement of the data in the file. This exercise, of deducing the exact arrangement of data in the file, is sometimes helped by having some idea of the display devices the format creator had in mind.

One or more scan lines combined form a 2D grid of pixel data; thus we can think of each pixel in the bitmap as located at a specific logical coordinate. A bitmap can also be thought of as a sequence of values that logically maps bitmap data in a file to an image on the display surface of an output device. Actual bitmap data is usually the largest single part of any bitmap format file.

How Bitmap Data Is Written to Files

Before an application writes an image to a file, the image data is usually first assembled in one or more blocks of memory. These blocks can be located in the computer's main memory space or in part of an auxiliary data collection device. Exactly how the data is arranged then depends on a number of factors, including the amount of memory installed, the amount available to the application, and the specifics of the data acquisition or file write operation in use. When bitmap data is finally written to a file, however, only one of two methods of organization is normally used: scan-line data or planar data.

Scan-line data

The first, and simplest, method is the organization of pixel values into rows or scan lines, briefly mentioned above. If we consider every image to be made up of one or more scan lines, the pixel data in the file describing that image will be a series of sets of values, each set corresponding to a row of the image.

Multiple rows are represented by multiple sets written from start to end in the file. This is the most common method for storing image data organized into rows.

If we know the size of each pixel in the image, and the number of pixels per row, we can calculate the offset of the start of each row in the file. For example, in an 8-bit image every pixel value is one byte long. If the image is 21 pixels wide, rows in the file are represented by sets of pixel values 21 bytes wide. In this case, the rows in the file start at offsets of 0, 21, 42, 63, etc. bytes from the start of the bitmap data.

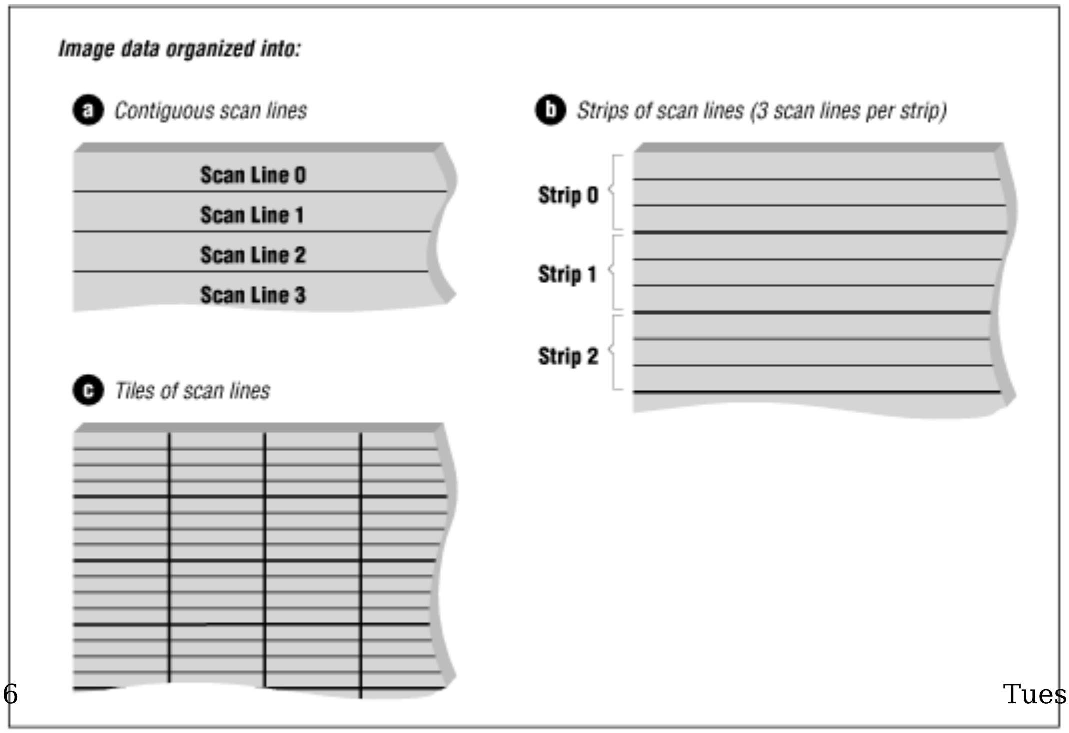
On some machines and in some formats, however, rows of image data must be certain even-byte multiples in length. An example is the common rule requiring bitmap row data to end on long-word boundaries, where a long word is four bytes long. In the example mentioned in the preceding paragraph, an image 21 pixels wide would then be stored in the file as sets of pixel values 24 bytes in length, and the rows would start at file offsets 0, 24, 48, 64. The extra three bytes per row are padding. In this particular case, three bytes of storage in the file are wasted for every row, and in fact, images that are 21 pixels wide take up the same amount of space as images 24 pixels wide. In practice, this storage inefficiency is usually (but not always) compensated for by an increase of speed gained by catering to the peculiarities of the host machine in regard to its ability to quickly manipulate two or four bytes at a time. The actual width of the image is always available to the rendering application, usually from information in the file header.

In a 24-bit image, each image pixel corresponds to a 3-byte long pixel value in the file. In the example we have been discussing, an image 21 pixels wide would require a minimum of 21 * 3 = 63 bytes of storage. If the format requires that the row starts be long-word aligned, 64 bytes would be required to hold the pixel values for each row. Occasionally, as mentioned above, 24-bit image data is stored as a series of 4-byte long pixel values, and each image row would then require 21 * 4 = 84 bytes. Storing 24-bit image data as 4-byte values has the advantage of always being long-word aligned, and again may make sense on certain machines.

In a 4-bit image, each pixel corresponds to one-half byte, and the data is usually stored two pixels per byte, although storing the data as 1-byte pixel values would make the data easier to read and, in fact, is not unheard of.

Figure 3-1 (ch03_04.htm#X058-9-C03-FG-1) illustrates the organization of pixel data into scan lines.

Figure 3-1: Organization of pixel data into scan lines (24-bit image)



Planar data

The second method of pixel value organization involves the separation of image data into two or more planes. Files in which the bitmap data is organized in this way are called planar files. We will use the term *composite image* to refer to an image with many colors (i.e., not monochrome, not gray-scale, and not one single color). Under this definition, most normal colored images that you are familiar with are composite images.

A composite image, then, can be represented by three blocks of bitmap data, each block containing just one of the component colors making up the image. Constructing each block is akin to the photographic process of making a separation--using filters to break up a color photograph into a set of component colors, usually three in number. The original photograph can be reconstructed by combining the three separations. Each block is composed of rows laid end to end, as in the simpler storage method explained above; in this case, more than one block is now needed to reconstruct the image. The blocks may be stored consecutively or may be physically separated from one another in the file.

Planar format data is usually a sign that the format designer had some particular display device in mind, one that constructed composite color pixels from components routed through hardware designed to handle one color at a time. For reasons of efficiency, planar format data is usually read one plane at a time in blocks, although an application may choose to laboriously assemble composite pixels by reading data from the appropriate spot in each plane sequentially.

As an example, a 24-bit image two rows by three columns wide might be represented in RGB format as six RGB pixel values:

```
(00, 01, 02) (03, 04, 05) (06, 07, 08)
(09, 10, 11) (12, 13, 14) (15, 16, 17)
```

but be written to the file in planar format as:

```
(00) (03) (06)
(09) (12) (15)
red plane
(01) (04) (07)
(10) (13) (16)
green plane
(02) (05) (08)
(11) (14) (17)
blue plane
```

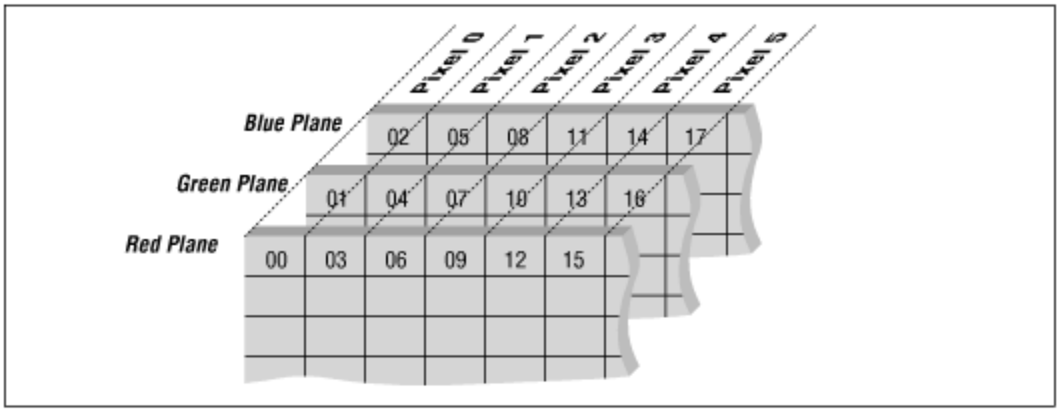
Notice that the exact same data is being written; it's just arranged differently. In the first case, an image consisting of six 24-bit pixels is stored as six 3-byte pixel values arranged in a single plane. In the second, planar, method, the same image is stored as 18 1-byte pixel values arranged in three planes, each plane corresponding to red, green, and blue information, respectively. Each method takes up exactly the same amount of space, 18 bytes, at least in this example.

It's pretty safe to say that most bitmap files are stored in non-planar format. Supporting planar hardware, then, usually means disassembling the pixel data and creating multiple color planes in memory, which are then presented to the planar rendering subroutine or the planar hardware.

Planar files may need to be assembled in a third buffer or, as mentioned above, laboriously set (by the routine servicing the output device) one pixel at a time.

Figure 3-2 (ch03_04.htm#X058-9-C03-FG-2) illustrates the organization of pixel data into color planes.

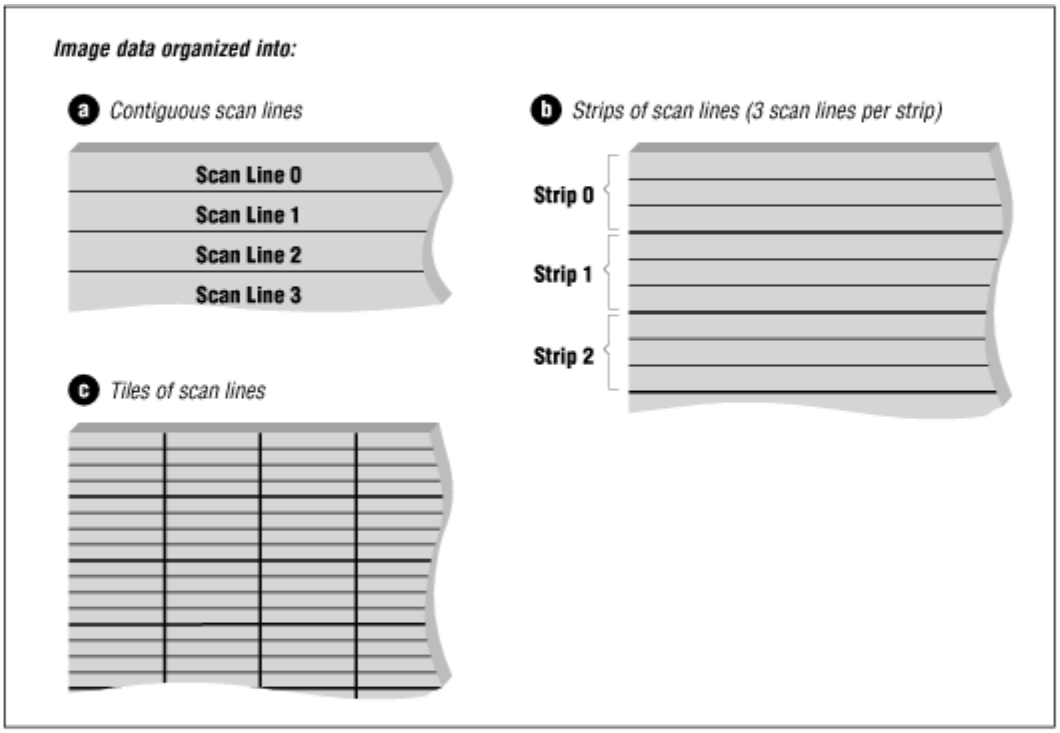
Figure 3-2: Organization of pixel data into color planes



Different Approaches to Bitmap Data Organization

Normally, we consider an image to be made up of a number of rows, each row a certain number of pixels wide. Pixel data representing the image can be stored in the file in three ways: as *contiguous data*, as *strips* or as *tiles*. Figure 3-3 (ch03_04.htm#X058-9-C03-FG-3) illustrates these three representations.

Figure 3-3: Examples of bitmap data organization (contiguous scan lines, strips, and tiles)



Contiguous data

The simplest method of row organization is where all of the image data is stored contiguously in the file, one row following the last. To retrieve the data you read the rows in file order, which delivers the rows in the order in which they were written. The data in this organizational scheme is stored in the file equivalent of a 2D array. You can index into the data in the file knowing the width of the row in pixels and the storage format and size of the pixel values. Data stored contiguously in this manner can be read quickly, in large chunks, and assembled in memory quite easily.

Strips

In the second method of file organization, images are stored in strips, which also consist of rows stored contiguously. The total image, however, is represented by more than one strip, and the individual strips may be widely separated in the file. Strips divide the image into a number of segments, which are always just as wide as the original image.

Strips make it easier to manage image data on machines with limited memory. An image 1024 rows long, for instance, can be stored in the file as eight strips, each strip containing 128 rows. Arranging a file into strips facilitates buffering. If this isn't obvious, consider an uncompressed 8-bit image 1024 rows long and 10,000 pixels wide, containing 10 megabytes of pixel data. Even dividing the data into eight strips of 128 rows leaves the reading application with the job of handling 1.25 megabytes of data per strip, a chore even for a machine with a lot of flat memory and a fast disk. Dividing the data into 313 strips, however, brings each strip down to a size which can be read and buffered quickly by machines capable of reading only 32K of data per file read pass.

Strips also come into play when pixel data is stored in a compressed or encoded format in the file. In this case, an application must first read the compressed data into a buffer and then decompress or decode the data into another buffer the same size as, or larger than, the first. Arranging the compression on a per-strip basis greatly eases the task of the file reader, which need handle only one strip at a time.

You'll find that strips are often evidence that a file format creator has thought about the limitations of the possible target platforms being supported and has wanted to not limit the size of images that can be handled by the format. Image file formats allowing or demanding that data be stored in strips usually provide for the storage of information in the file header such as the number of strips of data, the size of each strip, and the offset position of each strip within the file.

Tiles

A third method of bitmap data organization is tiling. Tiles are similar to strips in that each is a delineation of a rectangular area of an image. However, unlike strips, which are always the width of the image, tiles can have any width at all, from a single pixel to the entire image. Thus, in one sense, a contiguous image is actually one large tile. In practice, however, tiles are arranged so that the pixel data corresponding to each is between 4Kb and 64Kb in size and is usually of a height and width divisible by 16. These limits help increase the efficiency with which the data can be buffered and decoded.

When an image is tiled, it is generally the case that all the tiles are the same size, that the entire image is tiled, that the tiles do not overlap, and that the tiles are all encoded using the same encoding scheme. One exception is the CALS Raster Type II format (/format/cals/egff.htm), which allows image data to be composed of both encoded and unencoded image tiles. Tiles are usually left unencoded when such encoding would cause the tile data to increase in size (negative compression) or when an unreasonable amount of time would be required to encode the tile.

Dividing an image into tiles also allows different compression schemes to be applied to different parts of an image to achieve an optimal compression ratio. For example, one portion of an image (a very busy portion) could be divided into tiles that are compressed using JPEG (/format/jpeg/egff.htm), while another portion of the same image (a portion containing only one or two colors) could be stored as tiles that are run-length encoded. In this case, the tiles in the image would not all be the same uniform size; the smallest would be only a few pixels, and the largest would be hundreds or thousands of pixels on a side.

Tiing sometimes allows faster decoding and decompression of larger images than would be possible if the pixel data were organized as lines or strips. Because tiles can be individually encoded, file formats allowing the use of tiles will contain tile quantity, size, and offset information in the header specifications. Using this information, a reader that needs to display only the bottom-right corner of a very large image would have to read only the tiles for that area of the image; it would not have to read all of the image data that was stored before it.

Certain newer tile-oriented compression schemes, such as JPEG, naturally work better with file formats capable of supporting tiling. A good example of this is the incorporation of JPEG in later versions of the TIFF file format. For more information about the use of tiles, see the article on the TIFF (/format/tiff/egff.htm) file format.

Palette

Many bitmap file formats contain a color palette. For a discussion of palettes of different kinds, see Chapter 2 (ch02_01.htm).

[Previous (ch03_03.htm)] [Next (ch03_05.htm)]