(CN) 8 (CT) Digital Images and Image Formats

"This is the Wild West of the
Information Age" - Bart Kosko

(A-heading) The DataBahn

As the Internet connects the world together, and as more users cram to get onto this information thoroughfare, we find that it can take a long time to get data from point A to point B. Never before as now do we need to squeeze information content down in size before it is exchanged or transferred. Luckily the enormous size of digital images have already motivated the creation of a number of space saving image and file formats. Consider this: an uncompressed digital bitmap image of 640x480 pixels with 256 colors takes up 307 KB, or 1/3 MB. It can be frustrating to watch an image file of this size load into your browser from a web site.

In this chapter, you will find a discussion of the what and why of image formats - there are a lot of them ! You will learn some of the latest and greatest formats for the Internet. Finally, you will see details of the formats that are supported in Diffcad.

(A-heading) A Bird's Eye View of Image Formats

There are three broad categories of image formats: vector formats, bitmap formats and other formats.

(B-heading) Vector and Bitmap formats

In describing an image, you can resort to several levels of detail. In the lowest level of abstraction, you may describe each and every element (pixel) of the image. An image that is described in this manner is referred to as bitmapped, since the end result is a map of bits (or pixels). A bitmap image and its associated data are shown in Figures 8.1a and

8.1b.

Figure 8.1a A bitmap format image **Figure 8.1b Bitmap format data**

This is ultimately, how images are represented for display on monitors and viewed. A computer monitor or television has an addressable array of physical pixels or dots. The dot has position , and color information. Color will be talked about further very shortly. Bitmap images are difficult to scale from their original resolution (without DSP!). They can be bulky and hence cumbersome to transport. You can mitigate the size problem with compression (covered soon), but only at the expense of increased time to decode and render.

(B-Heading) Vector formats

Now we consider a way to represent images with a higher level of abstraction. Suppose you store endpoints of line segments to compose a representation of an image. This may be useful to render a CAD drawing for example. You store coordinates for the starting point, a direction and a length and maybe some color information. The rest of the screen that does not have line segments will be a background color. This is a simple vector image file format, and is a good compact format for line drawings. Vector formats are quick to read and are compact, for the types of images they are intended to represent. Vector formats typically store not only line primitives, but also some 2D shapes, such as circles and squares and curved lines , which are higher levels of abstraction. These shapes could be used to compose jet planes or integrated circuit layouts, for example. Figures 8.2a and 8.2b show an example of a simple vector image and its associated data.

Figure 8.2a Vector format image **Figure 8.2b Vector format data**

You could continue on using higher and higher levels of abstraction going to 3D objects as primitives, and Avatars (3D computer puppets) in virtual worlds.

One advantage of a vector format image, is that it is relatively easy to scale the image without loss of detail. Many clipart collections are stored as vector format files so that they can be scaled easily. A disadvantage of vector formats is that it is hard to store very detailed image information such as photographs, where you may need to vary color information on a pixel by pixel basis.

(C-heading) Conversion between vector and bitmap formats

Converting from a vector format image to a bitmap format is easy and straightforward; in fact, this conversion will be very common, since most display output devices are bitmapped. For a very detailed vector format image, it is important to choose a high enough resolution for the destination bitmap, otherwise, some artifacts will appear in the image, such as jagged lines (" the jaggies") instead of straight lines.

Converting from a bitmap format image to a vector image is difficult. In Chapter 9, you will meet this formidable challenge with DSP routines for edge and outline detection. Another issue is the possible loss of color information when you go from a rich bitmap representation to a (possibly) poor vector representation.

(B-heading) Other types of formats

[Murray et al.] describe several other types of formats for digital images. These are described briefly as follows:

· Scene - A scene format file has a condensed representation of an image. It is sometimes hard to tell the difference between this format and a vector format.

- Metafile - A metafile can store both vector format elements and bitmap format elements. Examples of this type of format file are the PICT format and the CGM format. Because of their versatility, these files are often used to cross the bridge between different hardware or software platforms.
- Animation - Animation formats come in many flavors. The simplest type just stores adjacent frames of an animation sequence in one file for playing. Another type stores not only images but along with them color maps for the images. Changing the color map can give the illusion of motion. Finally, a more sophisticated animation format will store frame difference information along with key frames. This technique is used to store motion video also and exploits the fact that in any given movie or animation, from frame to frame there is on average not a lot of changed information. There is usually a large chunk of the background or features that are static. If you store the data that changes, instead of all the data, you save a lot of space.
- Multimedia - Multimedia formats allow you to store all kinds of different data types and formats together; you could have video information, text information and sound information coexisting peacefully.
- 3D - 3D formats not only support descriptions of lines, shapes and 3D geometries, but also textures, reflections and anything else a rendering program would need to reconstruct a 3D image or world. Objects in a 3D file are sometimes called *scene elements*. Many existing vector file formats have been extended to support 3D. Such formats, such as Autodesk's DXF format, are referred to as *extended vector formats*. VRML is a little more than a 3D format, since it includes support for HTML style linking to other URLs on the World Wide Web.
- Font(bitmap, stroke, outline) - Fonts are special graphic files. They come in their own subsets of types based on bitmap formats or on vector formats (stroke, outline). One additional constraint usually imposed on font files is that they must be very quick to index into. So there is usually a database index associated with the font data placed in

a header or footer of the file.

- PDL - PDL, or Page Description Language formats are usually textual programmatic descriptions of how to render graphics and text. An example of this type of format is the ubiquitous Postscript format. This format is more akin to source code rather than just graphics data, and hence requires a sophisticated program in order to be able to create output.

(B-heading) Color Depth, Palettes and Transparency

Before getting to a monitor, an image, regardless of the format it is stored in by an application, is usually represented by a bitmap in a special memory known as a frame buffer. (The exception would be for a random scan display monitor - a vector image display monitor.) The frame buffer stores the image that is to be displayed as it is being scanned by the graphics processor that feeds a CRT monitor for viewing. En route to the monitor, the graphics processor may be commanded by the host processor (CPU) to manipulate the stream of pixels before they are displayed. One example of processing is to convert *color depth* information by using a *color look-up table* (CLUT), also known as a *color palette*. Color depth refers to the number of bits that are used to represent one pixel, or *bits per pixel* (bpp). For a color depth of 1, you represent a black and white image; a 1 turns on a pixel (white) and a 0 turns off a pixel (black). A color depth of 8 bpp, means that you have 256 (=2^8) possible values for a color.

A CLUT is a table of color values with an index. Use of this table can allow for some image size reduction. You essentially form an indirect addressing scheme. Say you have 256 possible values for a color (indexes). You may store 24 bit per pixel color values in each entry of the CLUT since your hardware supports it. You achieve some data compression since your image data may reference 8 bit CLUT indexes, instead of 24 bit color data. Your image data must also include the CLUT too however. The total data for a 640x480 image is (640x480x8 + 256x24)/8 = 308K bytes. If the CLUT was not used, the total data would be (640x480x24)/8 = 922K bytes. In this example the use of a CLUT

results in a savings of 2/3 in file size.

A CLUT doesn't always make sense to use. If you use a large number of colors in an image, then it may be more space-efficient to store the full pixel value directly. Generally, for images with more than 256 colors, it is better to store *literal* or *absolute* format, because the overhead of a very large CLUT is not worth the space. In fact the size of the CLUT may approach the size of the image itself.

(C-heading) Transparency

In the television world, you often see live video being overlaid onto a static image (like a weather map). In the bitmap world, this is like overlaying two bitmaps onto each other and specifying portions of one bitmap to be transparent, in certain areas, to allow the background image to show through. Similarly, you could use transparency characteristics to do a fade from one video source to another. In this case, there would be degrees of transparency (not just on or off). Transparency is often described in bitmaps on a pixel by pixel basis. Here, transparency information is appended to pixel value information. The TGA format for example uses 5 bits each for R, G, B (red, green, blue) and 1 extra bit for transparency, for a total of 16 bits. When the transparency bit is on, the display hardware must ignore that particular pixel, so that any background image may show through. A 32-bit variant of the TGA format specifies 8 bits for transparency, called the *alpha channel*. Here each of R, G and B use 8 bits and alpha uses an additional 8 bits to specify the degree of transparency (0=completely transparent to 255=completely opaque).

(A-heading) Graphics Formats Menu

This section will give you a directory of some of the more popular image formats and some of their traits. Below, Table 8.1 shows a list of many different formats along with

their type (adapted from [Murray et al.]) Diffcad uses a subset of these formats, namely: GIF, JPEG, VEC, PICT and PPM.

Later in the section, you will read about general characteristics of graphics formats, such as file organization, compression and progressive display.

Table 8.1 Image formats and type (adapted from [Murray et al.])

Format	Type
Autocad DXF	Vector
Autodesk 3D Studio	Scene description
BMP (Windows)	Bitmap
CGM	Metafile
FLI	Animation
GEM Raster	Bitmap
GEM VDI	Metafile
GIF	Bitmap
Harvard Graphics	Metafile
IFF	Bitmap
Intel DVI	Multimedia
JPEG File Interchange Format	Bitmap
Kodak PhotoCD	Bitmap
MPEG	Multimedia
PCX (Windows)	Bitmap
PICT (Mac)	Metafile
Pixar RIB	Scene Description
PNG	Bitmap
POV	Vector
PPM	Bitmap
QuickTime	Multimedia
Rayshade	Scene Description
SPIFF	Bitmap
Sun Raster	Bitmap
TIFF	Bitmap
TTDDD	Vector and Animation
Utah RLE	Bitmap
VEC	Vector
WMF (Windows)	Metafile

Most graphic formats support some form of data compression. In the next section you will see a discussion of compression methods.

(B-heading) Compression Methods : Making Bits of Bits

There are compression methods that are used on digital information (not just image data), and other methods specifically suited for image data and other special classes of data. First we look at four methods of general data compression: RLE, LZW, Huffman encoding and Arithmetic encoding. Later in this section we look at a compression method that is well suited for image data: DCT or transform based compression.
Compression can be *lossy,* or can be perfect. Lossy compressors discard information, albeit information that is considered to be the least relevant in a particular application. You will read more on this shortly.
Compression can be *symmetric*; that is the process of compression is very similar in complexity, time and methodology to decompression. On the other hand, *asymmetric* compression is a situation where a more complicated process is needed for one direction over the other; an example is the original Intel DVI video format, where a parallel supercomputer is used to compress a video sequence, while a tiny amount of microcode in a video DSP chip is used to decompress the sequence. This is highly asymmetric.
For Bitmap format files, normally only the bitmap data is compressed. Any other information in the file (header, footer) is left uncompressed for easy reading. For Vector format files, there is usually no compression. This is because Vector formats are inherently compact being a higher level of abstraction than a bitmap. Also rendering a vector format file takes a lot of time to begin with and adding decompression would further slow down applications that use vector format files.

Start NOTE

In the discussions below, encoding is usually discussed. The decoding process is just the set of reverse operations to that of encoding.

End NOTE

(C-heading) Run Length Encoding (RLE)

Run length encoding is a general compression method that takes sequences or runs of a particular character, and encodes it more compactly as a number and the character.
For example:

AAAAAAAAAABC

could be coded as: 10A1B1C

The number 10 is the run count and the following letter A is the run value. If each ASCII character takes up 1 byte of storage, then the original uncompressed string takes up 12 bytes, while the compressed string takes up 7 bytes. Notice that even a run length of 1 requires a minimum of 2 characters.

For binary character encoding, there are several choices. You can encode on a bit basis (looking for runs of bits), on a byte basis or on a pixel basis, where a pixel may take up multiple bytes. The overhead of storing a run length for each run value may in some cases cause a file to be larger than the original, which is termed negative compression. One method to minimize the effect of the run length code for small runs is to encode a bit at the beginning of each block that enables run length interpretation for that block. In other words, if the enable bit is set to 1, the block is interpreted as run length encoded. If it is set to 0, then the following data is interpreted as unencoded or *literal* data.
With 2D bitmap data, you have freedom to encode data along rows, which are also referred to as *scan lines*, or along columns, or along some other sub-block partitioning of the data.. You could choose some of the options discussed to achieve the best compression.

(C-heading) Lempel, Ziv, and Welch Compression (LZW)

A very widely used algorithm for data compression was invented by Lempel, Ziv and Welch and is known as LZW. Actually there are several algorithms: LZ77, LZ78 and LZW are all patented; use of these may be subject to licensing fees and a lot of legal headaches. It is possible to adapt LZ77 so that you do not infringe on its patent (see PKZIP below). Unfortunately, several widespread programs and formats made use of these patented algorithms, like the Compuserve GIF file format. As a backlash against having to pay for what used to be in the public domain and hence free, several alternatives to the patented LZW algorithms were developed and offered to the public. The popular archiving compressor, PKZIP replaced the original LZW compressor with a compressor based on an adapted non-infringing variation of the LZ77 algorithm. GIF is still oppressed with infringement problems. The world is still full of GIF images however. There are many freeware utilities to convert GIF files to other formats, such as PPM. The PNG graphics format was created specifically as an alternative to GIF and again is also based on a non-infringing variation of the LZ77 algorithm.

How does LZW work ? The LZ family of compressors are *dictionary-based encoding algorithms.* As data is read by a compressor, a table or *data-dictionary ,* is built that has entries for patterns that occur in the input data stream. If new data that is read, is not in the dictionary, then a new entry is made in the table for it. When data that has a dictionary entry is read in, then the entry, which has a smaller size than the original data, is copied to the output (compressed) data stream. A key feature of LZW is that the dictionary does not need to be stored for the decoder; the decoder will be able to reconstruct the dictionary because of the way that the data is organized. This can save a lot of overhead and space. LZW is a lossless compression scheme.

(C-heading) Huffman Encoding

Like LZW, Huffman encoding is based on code words. Here, shorter codes are chosen to represent the most commonly occurring sequences in a data stream, while longer codes are used for less frequent sequences. The letter A, if used very frequently in some input text, may be coded with 2 bits instead of the usual ASCII 8 bits, while the letter Q, which occurs very infrequently may be coded with 12 bits, as an example. The dictionary used for encoding is required for the decoder to do its work. There is no on-the-fly construction of a dictionary as you saw in the LZW compressor. The data stream that is produced from Huffman Encoding has a subtle requirement: Each code word should not be the prefix of any other code word. This will allow a decoder to uniquely determine each entry of the table based on a sequential read of the data stream. An improvement on Huffman Encoding is Arithmetic Encoding, which is discussed next. Both Huffman Encoding and Arithmetic Encoding are lossless compression schemes.

(C-heading) Arithmetic Encoding

Arithmetic Encoding, or *entropy coding,* improves on Huffman Encoding in a couple of ways: (1) you can have fractional codes, that is you can have a 4.18 bit long code (this is defined in a statistical way) and (2) more complex statistics are used that look at context information to derive a code for an input pattern - a U may be assigned a long code because it does not occur too often, while a U following a Q may be assigned a short code, since a U is very likely to follow a Q. A brand of Arithmetic Encoding, called a Q-coder is patented by IBM and AT&T and is subject to licensing considerations. An extension of the JPEG compression standard uses the Q-coder.

(C-heading) DCT based or Transform based Compression

The Discrete Cosine Transform (DCT) converts image data to the frequency domain, much like the DFT, the Discrete Fourier Transform which is discussed at length in the

next chapter. The DCT is a special case of the DFT [see Netravali et al.]. The transform yields a set of values that correspond to magnitudes of frequency components. The human eye cannot distinguish very high frequency color changes, and this information may be discarded without a great loss in detail of an image. Also, transform values which are zero or close to zero may be effectively compressed with a lossless compression scheme such as Huffman encoding as may be done in JPEG. The overall JPEG compression method is lossy.

(B-heading) Progressive Display and the Internet

For users of the Internet and the WWW, it is very useful to allow for progressive display of graphics images. This means that when a user is navigating the Web and goes to a new destination, if graphics data is loaded for display, it is shown while it is loading, so the user can immediately recognize the image, instead of waiting for the entire image file to load before seeing anything. A few formats allow for this: GIF and its patent-free successor, PNG, and JPEG. In GIF, there is an option to store graphics data with every eighth line of data, then every fourth line, then every second line and finally every line, for a total of four passes over the data. You can see a preview of an image with only one-eighth of the complete data in this scheme. This storage option is called the interlaced option for GIF. The non-interlaced storage option just stores rows sequentially and does not allow for progressive display.

Figure 8.3 Interlaced and Non-interlaced storage in GIF

The PNG format goes one up on GIF. PNG has an interlaced format, where every eighth pixel of every eighth line is first transmitted. This allows an image to be viewed with only 1/64 of the full image data.

JPEG data streams have an option for progressive display. Rather than being based on scan lines, the image is sent in progressively more detailed layers. That is, approximations of the original image are sent in sequence, so that the viewer sees the whole image right away, and the quality of the image improves with time. Each scan of progressive JPEG takes a full JPEG decompression cycle to display, which can be CPU intensive however. Another extension of JPEG provides for hierarchical storage of the same image at multiple resolutions, where a complete image is available at different resolutions to match the resolution of the display or print hardware.

(A-heading) Details of several formats

In this section you will read about details of some file formats, including those that are used by DiffCAD. Note that the Sun Java AWT class library provides built-in support for reading and writing JPEG and GIF image files. DiffCAD provides wrapper classes for some of this functionality. See the classes: WriteGIF, ReadGIF and VSImage.

(B-heading) GIF

GIF is a file format that uses the LZW compressor, as mentioned previously. There are two versions, GIF87a, the original and GIF89a. GIF89a may be incompatible with software that reads only GIF87a images, so most modern readers are expected to be able to read both formats. The formats are similar but GIF89a has further extensions. The file layout is shown below for both formats in Figure 8.4, and this highlights the differences:

Figure 8.4 GIF87a and GIF89a file layout

There are several pieces to the file format which are discussed in detail below:

- Header - the header is 6 bytes in size. The first 3 bytes are “GIF” to identify the format as GIF. The next three bytes are the version “87a” or “89a”.

- Logical Screen Descriptor - This is a fixed size group of bytes that contain information about the minimum screen resolution (height and width), and color information to reproduce the image. If the screen is smaller than the screen parameters, then some scaling will need to be performed by the application to display the image.

- Global Color Table - This is an optional section that contains a CLUT of up to 256 entries.

- Local Image Descriptor - This section has characteristics of the image data that follows including, where on the display the image should start and the image resolution and color information.

- Local Color Table - GIF is expandable to be able to include more than one image, though this is rarely used. There is therefore the provision to include a color table (termed “local”) for each of the images. This is an optional table for specifying a CLUT for the image data that follows. This table, if present, supercedes, the Global Color Table.

- Image Data - Image data when compressed by LZW usually comes out as a stream of data that must be read from beginning to end. GIF splits the data into a series of sub-blocks. Each sub-block starts with a count byte, which can range in value from 1 to 255. The count byte value specifies the number of data bytes that will follow. At the end of the sub-block a byte of value zero is used to terminate the sub-block.

(B-heading) JPEG/JFIF

The JPEG standard leaves some ambiguities that make it an incomplete file format standard. C-Cube Microsystems created a file format called JPEG File Interchange Format (JFIF) that fills in the gaps. It is completely based on the baseline JPEG standard. JPEG is generally best applied to high-resolution full color (24bpp) images. This is because the transform-based coding will have more latitude for compression with more color information. Keep in mind that sharp edges, such as those created by overlaid text, can become blurry. When compressing with JPEG, an application usually presents a quality setting that you may change to trade off compression to quality. For high frequency detail in your source image, you may want a high quality setting. This will result in lower compression however. The tradeoff between quality and compression is a thorny and persistent issue for JPEG.

A JPEG encoder uses the following steps:

1. Create header information
2. Read in the source image data in RGB
3. Transform data to YUV color space - Y is black and white intensity information, and the U and V channels have color information. This is done with a linear transformation (see Chapter 9 for color space conversion).

4. Subsample the U and V channels - that is throw away some color information because it should be imperceptible to the viewer; use fewer samples of U and V for every sample of Y.
5. Perform the DCT on the Y, U and V data.
6. Quantize the resulting coefficients into different bins (this performs some compression by reducing the number of different possible values; more aggressive quantization is used for the color components).
7. Huffman encode the quantized data and produce an output data stream.

Both a raw JPEG file and a JFIF file start with the the bytes 255 and 232 to signify the start of image marker. For a JFIF file, you will see the bytes 255 and 240 followed by the characters “JFIF”, and information about the image. Data that follows the first block is standard JPEG data as defined by the specification. For detailed information, obtain the specification from the American National Standards Institute [see ANSI].

(B-heading) PPM

PPM is a bitmap format that is used as an intermediate format when converting from one system or file format to another. There are a set of portable freeware utilities written by Jeff Poskanzer that convert to and from PPM to many other graphic file formats. For example, *ppmtogif* converts from the PPM format to GIF.

The file organization is extremely simple for PPM. You start with an ASCII header, and the bitmap data follows as either ASCII data or binary data. No compression is used. Data elements are separated by white space (space, tab, carriage return or linefeeds).

The PPM header looks like the following:

MagicValue	P3= ASCII data, P6= binary data
ImageWidth	Width of image in pixels (ASCII decimal value)
ImageHeight	Height of image in pixels (ASCII decimal value)
MaxGrey	Maximum color value (ASCII decimal value)

The MaxGrey value specifies the maximum value for a color component. Each pixel is specified by three values for R, G and B components.

Here is an example file:

```
# example of a 3 x 3 bitmap
P3
3 3
255
0 0 0          0 0 0          0 0 255
0 0 128        0 7 0          0 1  89
9 0  0         0 9  9         0  0  0
```

Comments may be included in a file starting with the # character. The bitmap is for 3 pixels tall by 3 pixels across. The third pixel of the second row has RGB values of (0, 1, 89).

Because PPM is a simple format, the entire source code is shown below in listing 8.1 - how DiffCad implements a PPM reader.

Listing 8.1 Reading the PPM format: The ReadPPM class

```
/**
 * ReadPPM is a class that reads an image from
 * a PPM format file.
 *
 * Victor Silva (victor@cse.bridgeport.edu).
 *
 */
```

```
import java.io.*;
import java.awt.image.*;

public class ReadPPM
{
  public ReadPPM(InputStream in)
  {
  }

  private int type;
  private static final int PBM_ASCII = 1;
  private static final int PGM_ASCII = 2;
  private static final int PPM_ASCII = 3;
  private static final int PBM_RAW = 4;
  private static final int PGM_RAW = 5;
  private static final int PPM_RAW = 6;

  private int width = -1, height = -1;
  private int maxval;

  /// Subclasses implement this to read in enough of the image stream
  // to figure out the width and height.
  void readHeader(InputStream in) throws IOException
  {
    char c1, c2;

    c1 = (char) readByte( in );
    c2 = (char) readByte( in );

    if (c1 != 'P')
    {
      throw new IOException( "not a PBM/PGM/PPM file" );
    }
    switch(c2)
    {
    case '1':
      type = PBM_ASCII;
      break;

    case '2':
      type = PGM_ASCII;
      break;

    case '3':
      type = PPM_ASCII;
      break;
```

```
    case '4':
      type = PBM_RAW;
      break;

    case '5':
      type = PGM_RAW;
      break;

    case '6':
      type = PPM_RAW;
      break;

    default:
      throw new IOException( "not a standard PBM/PGM/PPM file" );
    }
    width = readInt( in );
    height = readInt( in );
    if ( type != PBM_ASCII && type != PBM_RAW )
    {
      maxval = readInt( in );
    }
  }

  int getWidth()
  {
    return width;
  }

  int getHeight()
  {
    return height;
  }

  void readRow( InputStream in, int row, int[] rgbRow ) throws IOException
  {
    int col, r, g, b;
    int rgb = 0;
    char c;

    for(col=0; col<width; col++)
    {
      switch(type)
      {
      case PBM_ASCII:
        c = readChar( in );
```

```
      if ( c == '1' )
      {
        rgb = 0xff000000;
      }
      else
      {
        if ( c == '0' )
        {
          rgb = 0xffffffff;
        }
        else
        {
          throw new IOException( "illegal PBM bit" );
        }
      }
      break;
    case PGM_ASCII:
      g = readInt(in);
      rgb = makeRgb(g, g, g);
      break;
    case PPM_ASCII:
      r = readInt( in );
      g = readInt( in );
      b = readInt( in );
      rgb = makeRgb( r, g, b );
      break;
    case PBM_RAW:
      if ( readBit( in ) )
      {
        rgb = 0xff000000;
      }
      else
      {
        rgb = 0xffffffff;
      }
      break;
    case PGM_RAW:
      g = readByte( in );
      if ( maxval != 255 )
      {
        g = fixDepth( g );
      }
      rgb = makeRgb( g, g, g );
      break;
    case PPM_RAW:
      r = readByte( in );
```

```
          g = readByte( in );
          b = readByte( in );
          if ( maxval != 255 )
          {
            r = fixDepth( r );
            g = fixDepth( g );
            b = fixDepth( b );
          }
          rgb = makeRgb( r, g, b );
          break;

        default:
          break;
        }
        rgbRow[col] = rgb;
      }
    }

    private static int readByte(InputStream in) throws IOException
    {
      int b = in.read();

      // if end of file
      if (b == -1)
      {
        throw new EOFException();
      }
      return b;
    }

    private int bitshift = -1;
    private int bits;

    private boolean readBit( InputStream in ) throws IOException
    {
      if ( bitshift == -1 )
      {
        bits = readByte( in );
        bitshift = 7;
      }
      boolean bit = ( ( ( bits >> bitshift ) & 1 ) != 0 );
      --bitshift;
      return bit;
    }

    /// Utility routine to read a character, ignoring comments.
```

```
  private static char readChar( InputStream in ) throws IOException
  {
    char c;

    c = (char) readByte( in );
    if ( c == '#' )
    {
      do
      {
        c = (char) readByte( in );
      }
      while ( c != '\n' && c != '\r' );
    }

    return c;
  }

   /// Utility routine to read the first non-whitespace character.
  private static char readNonwhiteChar( InputStream in ) throws IOException
  {
    char c;

    do
    {
      c = readChar( in );
    }
    while ( c == ' ' || c == '\t' || c == '\n' || c == '\r' );

    return c;
  }

   /// Utility routine to read an ASCII integer, ignoring comments.
  private static int readInt( InputStream in ) throws IOException
  {
    char c;
    int i;

    c = readNonwhiteChar( in );
    if ( c < '0' || c > '9' )
    {
      throw new IOException( "junk in file where integer should be" );
    }

    i = 0;
    do
    {
```

```
      i = i * 10 + c - '0';
      c = readChar( in );
    }
    while ( c >= '0' && c <= '9' );

    return i;
  }

  /// Utility routine to rescale a pixel value from a non-eight-bit maxval.
  private int fixDepth( int p )
  {
    return ( p * 255 + maxval / 2 ) / maxval;
  }

  /// Utility routine make an RGBdefault pixel from three color values.
  private static int makeRgb( int r, int g, int b )
  {
    return 0xff000000 | ( r << 16 ) | ( g << 8 ) | b;
  }
}
```

(B-heading) VEC

The VEC format is a simple native vector file format used by DiffCad. Data in the file is integer ASCII data. There are no other markers in the file. There are two types of use in the format: POINT and VECTOR. First here is the POINT type:

Points are stored as ASCII decimal numbers and are comprised of two coordinates, x and y that are separated by spaces or tabs. Points are separated by newlines (linefeeds). Here is a file describing a 3 pixel square.

```
0 0
3 0
3 3
0 3
```

Storing separate points can be useful to describe shapes and objects. Edge detection (see Chapter 9) of an image can create this sort of output. Also, this format may be used for

computer vision and vector display applications.

The VECTOR type of data is specified in a similar manner, except that two points are defined per line as follows:

x1 y1 x2 y2
x3 y3 x4 y4
...

Here (x1, y1) defines the tail of a vector and (x2, y2) defines the head:
(x1, y1) (x2,y2)

Begin NOTE
Diffcad has a routine to convert from xy points to vectors.
End NOTE

(B-heading) PICT

The PICT format is a Macintosh metafile format. It can incorporate both bitmap and vector data. Diffcad can write PICT vector data only (the reader is referred to the savepict.java class)The PICT format is a fairly complex format and few details will be shown here. PICT can use two different forms of compression: JPEG and PackBits. PackBits is an RLE type of encoding scheme.

(A-heading) Summary

Digital image file formats are defined by several characteristics: the type of file format it is (vector, bitmap or other), the size efficiency based on the compression technology it uses, the number of colors it can handle and the resolution of images that it supports. Another factor useful for Internet based graphics is progressive display, which is the display of a partial image as a graphics file is downloaded. Three formats have specific provisions for this : GIF, PNG and JPEG. Although the universe of graphic file formats is

very large there is a great deal of similarity between formats of the same type. One bitmap format is likely to be as capable as another. Metafile formats provide the capability for vector and bitmap representations in a single format. Higher levels of abstraction are available in vector formats ultimately leading to representation of 3D objects, scenes and worlds in 3D formats. DiffCad supports GIF, JPEG, PPM, VEC, and a subset of PICT.