Graphics programming with libtiff, Part 2

And now for a little color

Level: Introductory

Michael Still (mikal@stillhq.com), Senior Software Engineer, Tower Software Engineering

01 Jun 2002

TIFF is an extremely common but quite complex raster image format. Libtiff is a standard implementation of the TIFF specification that is free and works on many operating systems. This article shows you how to use libtiff for grayscale and color imaging.

This article -- a continuation of the previous article on black and white graphics programming with libtiff -- covers grayscale and color imaging. It assumes that you have read and understand the code from the black and white article.

First let's review some theory about how the image data is stored for color and grayscale. This theory applies to all imaging formats. Then we'll cover the specifics of using libtiff.

A bit of terminology

Images are made up of pixels. In black and white imaging, the pixel has one of two values, 0 or 1. This can be represented in a single bit. For grayscale and color images however, the pixel needs to store a much greater range of values; if a pixel was to have 255 levels of gray, we would need 8 bits to store that pixel. Each of these values is called a sample. TIFF expresses the size of the value in a tag called `TIFFTAG_BITSPERSAMPLE`. This will be 1 for black and white, and some larger number for grayscale.

For color images, we need to store even more information. For each pixel we will need to store a red, green, and blue value. Each of these values is stored in a separate *sample*. Therefore, we will need to define `TIFFTAG_SAMPLESPERPIXEL`. This will be 1 for black and white, or grayscale, but will normally be 3 for color images. We also need to define the size of each sample, so we'll still need to set a value for `TIFFTAG_BITSPERSAMPLE`.

Theory of color and grayscale storage

The first thing we need to understand to be able to support color and grayscale images is the format of the image data within memory. There are two main representations for color and grayscale images. I'll explain these by describing grayscale, and then extend it to color.

Direct storage of pixel data

If you remember the way pixel information was stored in the black and white images from the previous article, the information was just in the strips. You can also do this with grayscale and color images, but this representation of image data is quite inefficient. For example, in a scenario in which the image has a solid background, there are many pixels with the same value. If the pixel data is stored in the strips, then this value will waste a large amount of space.

Thankfully, there is a more efficient way to store image data. Imagine a simple four-color, 24-bit-per-pixel image. If we build a lookup table of the four color values (the 24-bit values that represent those colors), then we just need to store the relevant entry number of the color in the image strip itself. This can be done in only two bits, instead of the full 24.

The math looks something like this: A 24-bit color image that is 1,000 by 1,000 pixels will take 24 million bits to store. The same image, if it was a four color image, would take 4 million bits for the strip data, and 98 bits for the color table. Neither of these numbers includes header and footer information for the file

format, and the numbers are for uncompressed bitmaps. The advantages of the lookup table are obvious. This style of lookup table is called a *palette*, probably because of those things painters carry around.

This concept works for grayscale images as well. The only difference is that the "colors" in the palette are just shades of gray.

Compression algorithms in libtiff

Several compression algorithms are available within libtiff. The table below helps sort them out.

Table 1. Libtiff compression algorithms

Compression algorithm	Well suited for	TIFFTAG
CCITT Group 4 Fax and Group 3 Fax	This entry is here for completeness. If you're coding for black and white images, then you're probably using the CCITT fax compression methods. These compression algorithms don't support color.	`COMPRESSION_CCITTFAX3, COMPRESSION_CCITTFAX4`
JPEG	JPEG compression is great for large images such as photos. However, the compression is normally lossy (in that image data is thrown away as part of the compression process). This makes JPEG very poor for compressing text which needs to remain readable. The other thing to bear in mind is that the loss is cumulative -- see the next section for more information about this.	`COMPRESSION_JPEG`
LZW	*This is the compression algorithm used in GIF images. Because of the licensing requirements from Unisys, support for this compression codec has been removed from libtiff. There are patches available if you would like to add it back, but the majority of programs your code will integrate with no longer support LZW.*	`COMPRESSION_LZW`
Deflate	This is the gzip compression algorithm, which is also used for PNG. It is the compression algorithm I would recommend for color images.	`COMPRESSION_DEFLATE`

Accumulating loss?

Why does the loss in lossy compression algorithms such as JPEG accumulate? Imagine that you compress an image using JPEG. You then need to add, say, a barcode to the image, so you uncompress the image, add the barcode, and recompress it. When the recompression occurs, a new set of loss is introduced. You can imagine that if you do this enough, then you'll end up with an image that is a big blob.

Whether this is a problem depends on the type of your data. To test how much of a problem this is, I wrote a simple libtiff program that repeatedly uncompresses and recompresses an image. What I found was that with pictures, the data is much more resilient to repeated compression.

Figure 1. The picture before compression

Figure 2. The sample text before compression

```
<sidebar>
<heading refname="" type="sidebar" toc="no">Accu
<p>Why does the loss in lossy compression algori
 add say a barcode to the image, so you uncompre
et of loss is introduced. You can imagine that i

<p>Whether this is a problem depends on the type
repeatedly uncompresses and recompresses an imag
on.</p>

<figure>
<heading refname="picture-start" type="figure" t
<img src="xxx.jpg" width="500" height="200" alt=
</figure>

<figure>
<heading refname="text-start" type="figure" toc=
<img src="xxx.jpg" width="500" height="200" alt=
</figure>
```

The code I used had a "quality" rating of 25% on the JPEG compression, which is a way of tweaking the loss of the compression algorithm. The lower the quality, the higher the compression ratio. The default is 75%.

Figure 3. The picture after 200 recompressions

Figure 4. The text after 200 recompressions

```
<sidebar>
<heading refname="" type="sidebar" toc="no">Accu
<p>Why does the loss in lossy compression algori
 add say a barcode to the image, so you uncompre
et of loss is introduced. You can imagine that i

<p>Whether this is a problem depends on the type
repeatedly uncompresses and recompresses an imag
on.</p>

<figure>
<heading refname="picture-start" type="figure" t
<img src="xxx.jpg" width="500" height="200" alt=
</figure>

<figure>
<heading refname="text-start" type="figure" toc=
<img src="xxx.jpg" width="500" height="200" alt=
</figure>
```

Writing a color image

Now we'll write a color image to disk. Remember that this is a simple example and can be elaborated on greatly.

Listing 1. Writing a color image

```
#include <tiffio.h>
#include <stdio.h>

int main(int argc, char *argv[]){
  TIFF *output;
  uint32 width, height;
  char *raster;
```

```
  // Open the output image
  if((output = TIFFOpen("output.tif", "w")) == NULL){
    fprintf(stderr, "Could not open outgoing image\n");
    exit(42);
  }

  // We need to know the width and the height before we can malloc
  width = 42;
  height = 42;

  if((raster = (char *) malloc(sizeof(char) * width * height * 3)) == NULL){
    fprintf(stderr, "Could not allocate enough memory\n");
    exit(42);
  }

  // Magical stuff for creating the image
  // ...

  // Write the tiff tags to the file
  TIFFSetField(output, TIFFTAG_IMAGEWIDTH, width);
  TIFFSetField(output, TIFFTAG_IMAGELENGTH, height);
  TIFFSetField(output, TIFFTAG_COMPRESSION, COMPRESSION_DEFLATE);
  TIFFSetField(output, TIFFTAG_PLANARCONFIG, PLANARCONFIG_CONTIG);
  TIFFSetField(output, TIFFTAG_PHOTOMETRIC, PHOTOMETRIC_RGB);
  TIFFSetField(output, TIFFTAG_BITSPERSAMPLE, 8);
  TIFFSetField(output, TIFFTAG_SAMPLESPERPIXEL, 3);

  // Actually write the image
  if(TIFFWriteEncodedStrip(output, 0, raster, width * height * 3) == 0){
    fprintf(stderr, "Could not write image\n");
    exit(42);
  }

  TIFFClose(output);
}
```

This code shows some of the things we've discussed in theory. The image has three samples per pixel, each of eight bits. This means that the image is a 24-bit RGB image. If this was a black and white or grayscale image, then this value would be one. The tag `PHOTOMETRIC_RGB` says that the image data is stored within the strips themselves (as opposed to being paletted) -- more about this in a minute.

The other interesting thing to discuss here is the planar configuration of the image. Here I've specified `PLANARCONFIG_CONTIG`, which means that the red green and blue information for a given pixel is grouped together in the strips of image data. The other option is `PLANARCONFIG_SEPARATE`, where the red samples for the image are stored together, then the blue samples, and finally the green samples.

Other values for samples per pixel?

In my example, I have three samples per pixel. If this was a black and white image, or a grayscale image, then we would have one sample per pixel.

There are other valid values as well; for instance, sometimes people will store a transparency value for a given pixel, an *alpha channel.* This would result in having four samples per pixel.

It is possible to have an arbitrary number of samples per pixel, which is good if you need to pack in extra information about a pixel. *Note that doing this can break image viewers that make silly assumptions -- I once had to write code for a former employer to strip out alpha channels and the like so that their PDF generator wouldn't crash.*

Writing a paletted color image

So how do we write a paletted version of this image? Well, libtiff makes this really easy -- all we need to do is change the value of `TIFFTAG_PHOTOMETRIC` to `PHOTOMETRIC_PALETTE`. It's not really worth including an example in this article, given it's a one word change.

Reading a color image

Now all we have to do is work out how to read other people's color and grayscale images reliably, and we're home free. Initially, I was very tempted to gloss over the `TIFFReadRGBAStrip()` and `TIFFReadRGBBSTile()` calls, which hide some of the potential ugliness from the caller. However, these

functions have some limitations, which are expressed in the TIFFReadRGBAStrip() man page:

TIFFReadRGBAStrip() man page excerpt

```
   TIFFReadRGBAStrip reads a single strip of a strip-based image into memory,
   storing the result in the user supplied RGBA raster. The raster is
   assumed to be an array of width times rowsperstrip 32-bit entries,
   where width is the width of the image (TIFFTAG_IMAGEWIDTH) and
   rowsperstrip is the maximum lines in a strip (TIFFTAG_ROWSPERSTRIP).

   The strip value should be the strip number (strip zero is the
   first) as returned by the TIFFComputeStrip function, but always for sample 0.

   Note that the raster is assume to be organized such that the pixel
   at location (x,y) is raster[y*width+x]; with the raster origin in the
   lower-left hand corner of the strip. That is bottom to top organization.
   When reading a partial last strip in the file the last line of the image
   will begin at the beginning of the buffer.

   Raster pixels are 8-bit packed red, green, blue, alpha samples. The
   macros TIFFGetR, TIFFGetG, TIFFGetB, and TIFFGetA should be used to
   access individual samples. Images without Associated Alpha matting
   information have a constant Alpha of 1.0 (255).

   See the TIFFRGBAImage(3T) page for more details on how various image types
   are converted to RGBA values.

NOTES

   Samples must be either 1, 2, 4, 8, or 16 bits. Colorimetric samples/pixel
   must be either 1, 3, or 4 (i.e. SamplesPerPixel minus ExtraSamples).

   Palette image colormaps that appear to be incorrectly written as 8-bit
   values are automatically scaled to 16-bits.

   TIFFReadRGBAStrip is just a wrapper around the more general
   TIFFRGBAImage(3T) facilities. It's main advantage over the similar
   TIFFReadRGBAImage() function is that for large images a single buffer
   capable of holding the whole image doesn't need to be allocated, only
   enough for one strip. The TIFFReadRGBATile() function does a similar
   operation for tiled images.
```

There are a couple of odd things about this function. First, it defines (0, 0) to be in a different location than all the other code that we have been writing. In the previous code, the (0, 0) point has been in the top left of the image. This call defines (0, 0) to be in the bottom left. The other limitation is that not all valid values for bits per sample are supported. If you find these quirks unacceptable, then remember that you can still use `TIFFReadEncodedStrip()` in the same manner that I did for the black and white images in the previous article.

Listing 2. Reading a color image with TIFFReadEncodedStrip()

```
#include <stdio.h>
#include <tiffio.h>

int main(int argc, char *argv[]){
  TIFF *image;
  uint32 width, height, *raster;
  tsize_t stripSize;
  unsigned long imagesize, c, d, e;

  // Open the TIFF image
  if((image = TIFFOpen(argv[1], "r")) == NULL){
    fprintf(stderr, "Could not open incoming image\n");
    exit(42);
  }

  // Find the width and height of the image
  TIFFGetField(image, TIFFTAG_IMAGEWIDTH, &width);
  TIFFGetField(image, TIFFTAG_IMAGELENGTH, &height);
  imagesize = height * width + 1;

  if((raster = (uint32 *) malloc(sizeof(uint32) * imagesize)) == NULL){
    fprintf(stderr, "Could not allocate enough memory\n");
    exit(42);
  }
```

```
  // Read the image into the memory buffer
  if(TIFFReadRGBAStrip(image, 0, raster) == 0){
    fprintf(stderr, "Could not read image\n");
    exit(42);
  }
// Here I fix the reversal of the image (vertically) and show you
// how to get the color values from each pixel
  d = 0;
  for(e = height - 1; e != -1; e--){
    for(c = 0; c < width; c++){
      // Red = TIFFGetR(raster[e * width + c]);
      // Green = TIFFGetG(raster[e * width + c]);
      // Blue = TIFFGetB(raster[e * width + c]);
    }
  }

  free(raster);
  TIFFClose(image);
}
```

Advanced topics

Well, now that we've covered reading and writing basically any image format we can think of, there are two final topics.

Storing TIFF data in places other than files

All the examples to this point have read and written with files. There are many scenarios in which you wouldn't want to store your image data in a file, but would still want to use libtiff and tiff. For example, you might have customer pictures for id cards, and these would be stored in a database.

An expanded example

If you need more information about hooking the file input and output functions within libtiff, take a look at the images.c file in Panda, my PDF library. The Web pages for Panda can be found in Resources later in this article.

The example I am most familiar with is PDF documents, where you can embed images into the document. These images can be in a subset of TIFF if desired, and TIFF is clearly the choice for black and white images.

Libtiff allows you to replace the file input and output functions in the library with your own. This is done with the `TIFFClientOpen()` method. Here's an example (please note this code won't compile, and is shown only to describe the main concepts):

Listing 3. Using TIFFClientOpen

```
#include <tiffio.h>
#include <pthread.h>

// Function prototypes
static tsize_t libtiffDummyReadProc (thandle_t fd, tdata_t buf, tsize_t size);
static tsize_t libtiffDummyWriteProc (thandle_t fd, tdata_t buf, tsize_t size);
static toff_t libtiffDummySeekProc (thandle_t fd, toff_t off, int i);
static int libtiffDummyCloseProc (thandle_t fd);

// We need globals because of the callbacks (they don't allow us to pass state)
char *globalImageBuffer;
unsigned long globalImageBufferOffset;

// This mutex keeps the globals safe by ensuring only one user at a time
pthread_mutex_t convMutex = PTHREAD_MUTEX_INITIALIZER;

...

TIFF *conv;

// Lock the mutex
pthread_mutex_lock (&convMutex);

globalImageBuffer = NULL;
```

```
globalImageBufferOffset = 0;

// Open the dummy document (which actually only exists in memory)
conv = TIFFClientOpen ("dummy", "w", (thandle_t) - 1, libtiffDummyReadProc,
            libtiffDummyWriteProc, libtiffDummySeekProc,
            libtiffDummyCloseProc, NULL, NULL, NULL);

// Setup the image as if it was any other tiff image here, including setting tags

...

// Actually do the client open
TIFFWriteEncodedStrip (conv, 0, stripBuffer, imageOffset);

// Unlock the mutex
pthread_mutex_unlock (&convMutex);

...

////////////////// Callbacks to libtiff

...

static tsize_t
libtiffDummyReadProc (thandle_t fd, tdata_t buf, tsize_t size)
{
  // Return the amount of data read, which we will always set as 0 because
  // we only need to be able to write to these in-memory tiffs
  return 0;
}

static tsize_t
libtiffDummyWriteProc (thandle_t fd, tdata_t buf, tsize_t size)
{
  // libtiff will try to write an 8 byte header into the tiff file. We need
  // to ignore this because PDF does not use it...
  if ((size == 8) && (((char *) buf)[0] == 'I') && (((char *) buf)[1] == 'I')
     && (((char *) buf)[2] == 42))
    {
    // Skip the header -- little endian
    }
  else if ((size == 8) && (((char *) buf)[0] == 'M') &&
       (((char *) buf)[1] == 'M') && (((char *) buf)[2] == 42))
    {
    // Skip the header -- big endian
    }
  else
    {
    // Have we done anything yet?
    if (globalImageBuffer == NULL)
    if((globalImageBuffer = (char *) malloc (size * sizeof (char))) == NULL)
        {
          fprintf(stderr, "Memory allocation error\n");
          exit(42);
        }

    // Otherwise, we need to grow the memory buffer
    else
    {
      if ((globalImageBuffer = (char *) realloc (globalImageBuffer,
                               (size * sizeof (char)) +
                               globalImageBufferOffset)) == NULL)
        fprintf(stderr, "Could not grow the tiff conversion memory buffer\n");
            exit(42);
    }

    // Now move the image data into the buffer
    memcpy (globalImageBuffer + globalImageBufferOffset, buf, size);
    globalImageBufferOffset += size;
    }

  return (size);
}

static toff_t
libtiffDummySeekProc (thandle_t fd, toff_t off, int i)
{
  // This appears to return the location that it went to
  return off;
}

static int
libtiffDummyCloseProc (thandle_t fd)
{
  // Return a zero meaning all is well
  return 0;
}
```

Converting color to grayscale

How do you convert color images to grayscale? My first answer was to just average the red, green, and blue values. That answer is wrong. The reality is that the human eye is much better at seeing some colors than others. To get an accurate grayscale representation, you need to apply different coefficients to the color samples. Appropriate coefficients are 0.299 for red, 0.587 for green, and 0.114 for blue.

Conclusion

In this article I've discussed how to program with libtiff for grayscale and color images. I've shown you some sample code that should help to get you started. You should now know enough to have a great time coding with libtiff.

Resources

- Read Michael's previous article on black and white graphics programming with libtiff (*developerWorks*, March 2002).

- Download the source files for performing the tasks mentioned in this article:
  - Reading a color impage: read.c
  - Writing a color image: write.c
  - Customizing file input and output functions: client.c
  - Compressing repeatedly: recompress.c

- Download the libtiff source and perhaps find a binary package for your operating system of choice at the libtiff Web site.

- For more information about hooking the file input and output functions within libtiff, take a look at the images.c file on Michael's Panda page.

- Check out Poynton's Color FAQ for a discussion of converting to grayscale.

- Find more Linux articles in the *developerWorks* Linux zone.

About the author

Michael has been working in the image processing field for several years, including a couple of years managing and developing large image databases for an Australian government department. He currently works for Tower Software, which manufactures a world-leading EDMS and records management package called TRIM. Michael is also the developer of Panda, an open source PDF-generation API, as well as the

maintainer of the comp.text.pdf USENET frequently asked questions document. You can contact Michael at mikal@stillhq.com.
