Using DIB sections to perform bulk color mapping

devblogs.microsoft.com/oldnewthing/20061116-00

November 16, 2006

Raymond Chen

When doing dithering, one operation you have to do for every pixel is map it (more accurately, map a modified version of it) to the nearest color in your available palette. Since this is part of the dithering inner loop, you need this operation to be as fast as possible.[1] A common technique for this is to precompute the nearest palette index for a dense sampling of colors. Any time you need to convert a pixel, you can find a nearby entry in the sampling and look up the precomputed nearest palette index. This won't give you the absolute best match for colors that are very close to the halfway point between two palette colors, but error diffusion dithering is an approximation anyway; if you choose your dense sampling to be "dense enough", these errors are small and are accounted for in the error diffusion algorithm.

But how do you build up this table mapping each color in your dense sampling to the palette? One way is to call `GetNearestPaletteIndex` for every pixel in the dense sampling. But the dense sampling by its nature has a large number of entries, and each call to `GetNearestPaletteIndex` is a ring transition. If only there were a way to do a bulk call to `GetNearestPaletteIndex` where you pass a whole bunch of `COLORREF` s at once.

But there is a way to do that, and that's the idea kernel for today. After all, GDI does it when you do a 24-bit to 8-bit blit. You can harness this energy with the aid of DIB sections: Create a source bitmap that consists of all the color values in your dense sample and a destination bitmap that is an 8bpp DIB section with your palette as its color table. Blit the source onto the destination, and the result is a destination that is exactly the mapping table you need!

Let's code this up. For the sake of illustration, our dense sampling will consist of 32768 data points distributed throughout the 555 color space. In that way, we can take an RGB value and map it to our 8-bit palette by means of the following expression:

```
extern BYTE table[32][32][32];
index = table[red >> 3][green >> 3][blue >> 3];
```

Since bitmaps are two-dimensional, we can't generate a three-dimensional table like the one given above. Let's view it not as a 32 × 32 × 32 array but rather as a one-dimensional array with 32768 elements. (This is, after all, how it's represented in memory anyway, so it's not like we’re really changing anything physically.) With that minor change of point of view, we’re ready to generate the desired table:

```
void CreateMappingTable(HPALETTE hpal)
{
 struct {
  BITMAPINFOHEADER bmiHeader;
  union {
   RGBQUAD bmiColors[256]; // when in palette mode
   DWORD rgMasks[3];       // when in BI_BITFIELDS mode
  };
 } bmi;
 PALETTEENTRY rgpe[256];
 UINT cColors = GetPaletteEntries(hpal, 0, 256, rgpe);
 if (cColors) {
  for (UINT i = 0; i < cColors; i++) {
   bmi.bmiColors[i].rgbRed = rgpe[i].peRed;
   bmi.bmiColors[i].rgbBlue = rgpe[i].peBlue;
   bmi.bmiColors[i].rgbGreen = rgpe[i].peGreen;
   bmi.bmiColors[i].rgbReserved = 0;
  }
  bmi.bmiHeader.biSize = sizeof(bmi.bmiHeader);
  bmi.bmiHeader.biWidth = 32768;
  bmi.bmiHeader.biHeight = 1;
  bmi.bmiHeader.biPlanes = 1;
  bmi.bmiHeader.biBitCount = 8;
  bmi.bmiHeader.biCompression = BI_RGB;
  bmi.bmiHeader.biSizeImage = 32768;
  bmi.bmiHeader.biClrImportant = cColors;
  bmi.bmiHeader.biClrUsed = 0;
  bmi.bmiHeader.biXPelsPerMeter = 0;
  bmi.bmiHeader.biYPelsPerMeter = 0;
  void *pv8bpp;
  HBITMAP hbmTable = CreateDIBSection(NULL, (BITMAPINFO*)&bmi,
                        DIB_RGB_COLORS, &pv8bpp, NULL, 0);
  if (hbmTable) {
   WORD rgw555[32768];
   for (int i = 0; i < 32768; i++) {
       rgw555[i] = (WORD)i;
   }
   bmi.bmiHeader.biSize = sizeof(bmi.bmiHeader);
   bmi.bmiHeader.biWidth = 32768;
   bmi.bmiHeader.biHeight = 1;
   bmi.bmiHeader.biPlanes = 1;
   bmi.bmiHeader.biBitCount = 16;
   bmi.bmiHeader.biCompression = BI_BITFIELDS;
   bmi.bmiHeader.biSizeImage = sizeof(rgw555);
   bmi.bmiHeader.biClrImportant = 0;
   bmi.bmiHeader.biClrUsed = 0;
   bmi.bmiHeader.biXPelsPerMeter = 0;
   bmi.bmiHeader.biYPelsPerMeter = 0;
   bmi.rgMasks[0] = 0x7C00;    // 5 red
   bmi.rgMasks[1] = 0x03E0;    // 5 green
   bmi.rgMasks[2] = 0x001F;    // 5 blue
   if (SetDIBits(NULL, hbmTable, 0, 1, rgw555,
                (BITMAPINFO*)&bmi, DIB_RGB_COLORS)) {
     CopyMemory(table, pv8bpp, 32768);
    }
    DeleteObject(hbmTable);
   }
  }
 }
```

Nearly all of this function is just preparation for the actual work that happens at the very end.

First, we get the colors in the palette and have the annoying job of converting them from `PALETTEENTRY` structures (which is what `GetPaletteEntries` gives you) to `RGBQUAD` entries (which is what `CreateDIBSection` wants). Why the two can't use the same format I will never know.

Next, we create our destination bitmap, an 8bpp bitmap with the palette entries as the color table, one pixel tall and 32768 pixels wide. Since this is a DIB section, GDI gives us a pointer (`pv8bpp`) to the actual bits in memory. Since we specified a 1 × 32768 bitmap, the format of the pixel data is just a sequence of 32768 bytes, each one corresponding to a palette index. Wow, that's exactly the format we want for our final table!

Building the source "bitmap" involves a few tricks. The naive approach is to have a 32768-element array of `RGBQUAD` s, each one describing one of the pixels in our dense sample set. Filling that array would have gone something like this:

```
for (r = 0; r < 31; r++)
 for (g = 0; g < 31; g++)
  for (b = 0; b < 31; b++) {
   rgrgb[(r << 10) | (g << 5) | b].rgbRed = r << 3;
   rgrgb[(r << 10) | (g << 5) | b].rgbGreen = g << 3;
   rgrgb[(r << 10) | (g << 5) | b].rgbBlue = b << 3;
   rgrgb[(r << 10) | (g << 5) | b].rgbReserved = 0;
  }
```

The first trick is to realize that we're just manually converting our 555 pixel data into RGB, something GDI is perfectly capable of doing on its own. Why not save ourselves some effort and just give GDI the 555 bitmap and let it do the conversion from 555 to RGB itself. (Besides, it might not even need to do that conversion; who knows, maybe there's a 555-to-8bpp optimized blit code path inside GDI we can take advantage of.) Using a 555 bitmap gives us this loop instead:

```
for (r = 0; r < 31; r++)
 for (g = 0; g < 31; g++)
  for (b = 0; b < 31; b++)
   rgw555[(r << 10) | (g << 5) | b] = (r << 10) | (g << 5) | b;
```

The second trick is strength-reducing this triple loop to simply

```
for (i = 0; i < 32768; i++) {
 rgw555[i] = i;
}
```

Now that we have the bitmap data and the `BITMAPINFO` that describes it, we can use `SetDIBits` to make GDI do all the work. The function takes our "bitmap" (one row of 32768 pixels, each in a different color and collectively exhausting our dense sample set) and sets it into our DIB section. By the magic of `BitBlt`, each pixel is mapped to the nearest matching color in the destination palette, and its index is stored as the pixel value.

And wow, that's exactly the format we want in our `table`! A little `CopyMemory` action and we're home free.

If you think about it in just the right way, this all becomes obvious. You just have to realize that `BitBlt` (or here one of its moral equivalents, `SetDIBits`) does more than just copy bits; it maps colors too. And then realize that you can extract the results of that mapping via a DIB section. Since you're handing in an entire bitmap instead of just a single color, you can map all 32768 colors at once.

Footnote 1: You might consider taking the technique in this article in another direction and simply blitting the entire 24bpp bitmap to a palettized DIB, thereby avoiding the intermediate translation table. The problem with this technique is that parenthetical "more accurately, map a modified version of it". The colors that need to be mapped to the palette are typically not the ones in the source bitmap but instead have been modified in some way by the dithering algorithm. In the case of an error-diffusion dither, the color values being mapped aren't even known until the preceding pixels have already been dithered. As a result, you can't blit all the pixels at once since you don't even know what color values you need to map until you have the result of previous mappings.

[Updated 9:30am to fix 6's, 3's, 5's and 10's. -Raymond]

Raymond Chen

Follow