Game Programming Genesis
Part V : Palettes and Pixels in DirectDraw

by Joseph "Ironblayde" Farrell

Introduction

Today we're going to go over basic DirectDraw graphics using both palettized and RGB modes. What's the difference? Well, a palettized screen mode is probably what you're used to if you've been programming in DOS. To plot pixels, you write a single byte to the video memory. That byte is an index into a lookup table full of colors, which is called a palette. RGB modes are different, because there's no lookup table. To plot a pixel in an RGB display mode, you write the values for red, green, and blue directly into the video memory. Any color depth higher than 8-bit is RGB instead of palettized. But now we're getting ahead of ourselves!

In writing this article, I assume you have either read my previous entries in the series, or are otherwise familiar with setting up DirectDraw and creating surfaces. We'll be using version 7 of DirectX, which contains the most recent DirectDraw interfaces. Actually, on a side note, the DirectDraw interfaces in DirectX 7 will be the last ones released! Don't worry, future versions of DirectX will still be backwards-compatible, but future implementations will have DirectDraw and Direct3D integrated into one component. But let's not worry about that now. :)

One last thing I will mention before we get started: the information on palettes is not necessary for anything I'll be covering later in the series, so if you're not interested in palettized modes, feel free to skip over the first part of this article and pick up when we get to the Pixel Formats section. Now that my disclaimers are at an end, let's get going!

Creating a DirectDraw Palette

Before working with graphics in a color depth of 8-bit or less, you must create a palette, which is simply a table full of colors. More specifically, in DirectX, a palette is a table full of PALETTEENTRY structures. To create one, there are three steps we need to take:

1. Create the color lookup table.
2. Get a pointer to an IDirectDrawPalette interface.
3. Attach the palette to a DirectDraw surface.

I'll assume that we're using 8-bit color for the rest of this section; if you wanted to write a game with 16 colors (or less), you wouldn't be reading up on all this crazy Windows stuff, right? Anyway, in 8-bit color depth, you have a palette with 256 entries. So to create our lookup table, we'll need 256 of these:

```
typedef struct tagPALETTEENTRY { // pe
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

The first three members are fairly obvious; they are the intensities of red, green, and blue in the given color. Each one can range from 0 to 255, as BYTE is an unsigned data type. The last member of the structure is a control flag and should be set to PC_NOCOLLAPSE. Basically what this does is guarantees that your colors will be preserved as they are, rather than being matched to an existing system palette. We've all seen what MS Paint does when you try to save a picture as an 8-bit image. You want that happening to your game? I didn't think so. :)

Now, as I said, we're going to need a bunch of these structures, so the logical thing to do is to declare an array that we'll use for the lookup table. It would look like this:

```
PALETTEENTRY palette[256];
```

So now that we've got our array, you can load in your colors. When I'm working in a palettized mode, I'll usually have my colors saved in an external file, and use something like this to load the colors in:

```
FILE* file_ptr;
int x;

if ((file_ptr = fopen("palette.dat", "rb")) != NULL)
{
  fread(palette, sizeof(PALETTEENTRY), 256, file_ptr);
  fclose(file_ptr);
}
```

All right, that does it for step one. Now we need to get the palette interface. This is accomplished by a call to `IDirectDraw7::CreatePalette()`, which looks like this:

```
HRESULT CreatePalette(
  DWORD dwFlags,
  LPPALETTEENTRY lpColorTable,
  LPDIRECTDRAWPALETTE FAR *lplpDDPalette,
  IUnknown FAR *pUnkOuter
);
```

The return type is the standard `HRESULT` we're used to, so you can use the `FAILED()` and `SUCCEEDED()` macros to test whether or not the call succeeds. The parameters are as follows:

`DWORD dwFlags`: Any number of a series of flags describing the palette object. As usual, you can combine the values with the `|` operator. The valid flags are:

`DDPCAPS_1BIT`	1-bit color; this corresponds to a 2-color palette.
`DDPCAPS_2BIT`	2-bit color; this corresponds to a 4-color palette.
`DDPCAPS_4BIT`	4-bit color; this corresponds to a 16-color palette.
`DDPCAPS_8BIT`	8-bit color; this corresponds to a 256-color palette.
`DDPCAPS_8BITENTRIES`	Indicates that the index refers to an 8-bit color index. That is, each color entry is itself an index to a destination surface's 8-bit palette. It must be combined with `DDPCAPS_1BIT`, `DDPCAPS_2BIT`, or `DDPCAPS_4BIT`. This is called an indexed palette. Weird, hey?
`DDPCAPS_ALPHA`	Indicates that the `peFlags` member of each `PALETTEENTRY` should be interpreted as an alpha value. Palettes created with this flag can only be attached to a Direct3D texture surface, since DirectDraw itself doesn't support alpha-blending.
`DDPCAPS_ALLOW256`	Allows all 256 entries of an 8-bit palette to be used. Normally, index 0 is reserved for black, and index 255 is reserved for white.
`DDPCAPS_INITIALIZE`	Indicates that the palette should be initialized with the values of an array of `PALETTEENTRY`s that is passed in another parameter of `CreatePalette()`.
`DDPCAPS_PRIMARYSURFACE`	Indicates that the palette will be attached to the primary surface, so that altering it immediately changes the colors of the display.

DDPCAPS_VSYNC	Forces palette changes to take place only during the vertical blank, minimizing the odd effects normally associated with palette-changing during a drawing cycle. This is not fully supported.

For the most part, you'll want to use `DDPCAPS_8BIT | DDPCAPS_INITIALIZE`, though you can take out the latter if you just want to create an empty palette and set it up later. Or you may want to add `DDPCAPS_ALLOW256` if you really want to change the two normally reserved entries.

`LPPALETTEENTRY lpColorTable`: This is a pointer to the lookup table we created, so just pass the name of the array.

`LPDIRECTDRAWPALETTE FAR *lplpDDPalette`: This is the address of a pointer to an `IDirectDrawPalette` interface, which will be initialized if the call succeeds.

`IUnkown FAR *pUnkOuter`: As always, this is for advanced COM stuff and should be set to `NULL`.

That's not too bad, so now we can go ahead and create our palette object. The last step is to attach that palette to a surface, which requires only a simple function call to `IDirectDrawSurface7::SetPalette()`. The function is shown below:

```
HRESULT SetPalette(LPDIRECTDRAWPALETTE lpDDPalette);
```

Pretty straightforward, isn't it? All you do is pass the interface pointer we got in the last step. So, to put a quick example together, let's assume we already have a lookup table created in an array called palette, like we did above. To create a DirectDraw palette to hold the colors, and attach it to our primary surface (which has also been created already), we do the following:

```
LPDIRECTDRAWPALETTE lpddpal;

// create the palette object
if (FAILED(lpdd7->CreatePalette(DDPCAPS_8BIT | DDPCAPS_INITIALIZE,
                                palette, &lpddpal, NULL)))
{
  // error-handling code here
}

// attach to primary surface
if (FAILED(lpddsPrimary->SetPalette(lpddpal)))
{
  // error-handling code here
}
```

And that's all there is to it. Once your palette is set up, plotting pixels is not too different from the way it's done in RGB modes. From this point on, I'll be covering both RGB and palettized modes. Before we get into actually displaying graphics, though, I need to show you what RGB pixel formats are like.

Pixel Formats

As I said earlier, hen you're writing a pixel into memory in a palettized mode, you write one byte at a time, and each byte represents an index into the color lookup table. In RGB modes, however, you write the actual color descriptors right into memory, and you need more than one byte for each color. The size of the memory write is equivalent to the color depth; that is, for 16-bit color, you write two bytes (16 bits) for each pixel, and so on. Let's start out at the top, because it's easiest to understand. 32-bit color uses a pixel format like this, where each letter is one bit:

```
AAAA AAAA RRRR RRRR GGGG GGGG BBBB BBBB
```

The As are for "alpha," which is a value representing transparency. Those are for Direct3D though; like I said, DirectDraw doesn't support alpha blending. So when you're creating a 32-bit color for DirectDraw, just set the high byte to 0. The next eight bits are the intensity of red, the eight following that are for green, and the low byte is for blue.

A pixel in 32-bit color needs to be 32 bits in size, and so the variable type we use to hold one is a `UINT`, which is an unsigned integer. Usually I use macros to convert RGB data into the correct pixel format, so let me show you one here. Hopefully if you're a little confused at this point, this will clear things up a bit:

```
#define RGB_32BIT(r, g, b)  ((r << 16) | (g << 8) | (b))
```

As you can see, this macro creates a pixel value by shifting the bytes representing red, green, and blue to their appropriate positions. Is it starting to make sense? To create a 32-bit pixel, you can just call this macro. Since red, green, and blue have eight bits each, they can range from 0 to 255. To create a white pixel, you would do this:

```
UINT white_pixel = RGB_32BIT(255, 255, 255);
```

24-bit color is just about the same. As a matter of fact, it is the same, except without the alpha information. The pixel format looks like this:

```
RRRR RRRR GGGG GGGG BBBB BBBB
```

So red, green, and blue still have eight bits each. This means that 24-bit color and 32-bit color actually have the same number of colors available to them, but 32-bit color just has some added information for transparency. So if you don't need the extra info, 24-bit is better than 32-bit, right? Well, not exactly. It's actually kind of a pain to deal with, because there's no data type that's 24 bits. So to write a pixel, instead of just writing in one value, you have to write red, green, and blue each individually. Working in 32-bit color is probably faster on most machines, even though it requires more memory. In fact, many video cards don't support 24-bit color at all, because having each pixel take up three bytes is just too inconvenient.

Now, 16-bit color is a bit tricky, because not every video card uses the same pixel format! There are two formats supported. One of them, which is by far more common, has five bits for red, six bits for green, and five bits for blue. The other format has five bits for each, and the high bit is unused. This is used mostly on older video cards. So the two formats look like this:

565 format: `RRRR RGGG GGGB BBBB`
555 format: `0RRR RRGG GGGB BBBB`

When you're working in a 16-bit color depth, you'll need to determine whether the video card uses 565 or 555 format, and then apply the appropriate technique. It's kind of a pain, but there's no way around it if you're going to use 16-bit color. Since there are two different formats, you'd write two separate macros:

```
#define RGB_16BIT565(r, g, b)  ((r << 11) | (g << 5) | (b))
#define RGB_16BIT555(r, g, b)  ((r << 10) | (g << 5) | (b))
```

In the case of 565 format, red and blue can each range from 0 to 31, and green ranges from 0 to 63. In 555 format, all three components range from 0 to 31. So setting a pixel to white in each mode would look like this:

```
USHORT white_pixel_565 = RGB_16BIT565(31, 63, 31);
USHORT white_pixel_555 = RGB_15BIT555(31, 31, 31);
```

The `USHORT` data type is an unsigned short integer, which is 16 bits long. This whole business of having two pixel formats makes things a bit confusing, but when we actually get into putting a game together, you'll see that it's not as bad as it seems at first. By the way, sometimes 555 format is referred to as 15-bit color. So if I call it that sometime later on, you'll know what I'm talking about instead of thinking I made a typo. :)

Here is probably a good place to show you just how exactly how determine whether a machine is using the 555 or 565 format when you're running in 16-bit color. The easiest way to do it is to call the `GetPixelFormat()` method of the `IDirectDrawSurface7` interface. Its prototype looks like this:

```
HRESULT GetPixelFormat(LPDDPIXELFORMAT lpDDPixelFormat);
```

The parameter is a pointer to a `DDPIXELFORMAT` structure. Just declare one, initialize it, and pass its address. The structure itself is huge, so I'm not going to list it here. Instead, I'll just tell you about three of its fields. The members in question are all of type `DWORD`, and they are `dwRBitMask`, `dwGBitMask`, and `dwBBitMask`. They are bit masks that you logically AND with a pixel value in order to extract the bits for red, green, or blue, respectively. You can also use them to determine what pixel format you are dealing with. If the video card uses 565, `dwGBitMask` will be `0x07E0`. If it uses 555 format, `dwGBitMask` will be `0x03E0`.

Now that we've seen all the pixel formats you can encounter, we can get into actually showing graphics in DirectX. About time, wouldn't you say? Before we can manipulate the actual pixels on a surface, though, we need to lock the surface, or at least a part of it. Locking the surface will return a pointer to the memory the surface represents, so then we can do whatever we want with it.

Locking Surfaces

Not surprisingly, the function we'll use to do this is `IDirectDrawSurface7::Lock()`. Let's take a look at it.

```
HRESULT Lock(
  LPRECT lpDestRect,
  LPDDSURFACEDESC lpDDSurfaceDesc,
  DWORD dwFlags,
  HANDLE hEvent
);
```

Remember to check the function call for success or failure, or it could lead to problems. If the lock fails, then the pointer to the surface won't be correct, and who knows what part of memory you'd be messing with then? The parameters for this function are:

`LPRECT lpDestRect`: This is a `RECT` that represents the area on the surface we want to lock. If you want to lock the entire surface, simply pass `NULL`.

`LPDDSURFACEDESC2 lpDDSurfaceDesc`: This is a pointer to a `DDSURFACEDESC2` structure, which is the big baddie we covered last time. All you need to do is initialize the structure, then pass the pointer. If `Lock()` succeeds, it fills in some of the members of the structure for you to use.

`DWORD dwFlags`: What kind of DirectX function would this be if it didn't have a list of flags do go along with it? Here are the most useful ones, which you can combine in the usual way:

`DDLOCK_READONLY`	Indicates that the surface being locked will only be read from, not written to.
`DDLOCK_SURFACEMEMORYPTR`	Indicates that a valid memory pointer to the upper-left corner of the specified `RECT` should be returned in the `DDSURFACEDESC2` structure. Again, if no `RECT` is specified, the pointer will be to the upper-left corner of the surface.
`DDLOCK_WAIT`	If a lock cannot be obtained because a blit is in progress, this flag indicates to keep retrying until a lock is obtained, or a different error occurs.

DDLOCK_WRITEONLY	Indicates that the surface being locked will only be written to, not read from.

Since we'll be using the lock to manipulate pixels, you'll always want to use DDLOCK_SURFACEMEMORYPTR. And even though we haven't gotten to using the blitter yet, it's usually a good idea to include DDLOCK_WAIT as well.

HANDLE hEvent: This parameter is not used and should be set to NULL.

Once we lock the surface, we need to take a look at the DDSURFACEDESC2 structure to get some information about the surface. We went over all of the members of this structure last time, but there are only two that we need to concern ourselves with at this point. Since both are very important, I'll list those two here again.

LONG lPitch: The lPitch member represents the number of bytes in each display line. You'd think this would be obvious. For example, in 640x480x16, there are 640 pixels in each line, and each one requires 2 bytes for color information, so the pitch (also called the "stride") should be 1280 bytes, right? Well, on some video cards, it will be greater than 1280. The extra memory on each line doesn't hold any graphical data, but sometimes it's there because the video card can't create a perfectly linear memory mode for the display mode. This will happen on a very small percentage of video cards, but you need to take it into account.

LPVOID lpSurface: This is a pointer to the memory represented by the surface. No matter what display mode you're using, DirectDraw creates a linear addressing mode you can use to manipulate the pixels of the surface.

The lpSurface pointer is pretty easy to understand, but are you following me on this whole pitch thing? It's an important value to remember, because you'll have to use it to calculate the offset to a particular pixel. We'll get back to it in just a minute. There's one thing I want to get out of the way first. When you're done plotting pixels, you need to unlock the surface that you locked. The prototype for IDirectDrawSurface7::Unlock() is this:

```
HRESULT Unlock(LPRECT lpRect);
```

The parameter is the same RECT you passed to Lock(). All right, let's plot some pixels.

Plotting Pixels

The first thing is to typecast the pointer we got from the Lock() function. Logically, we're going to want a pointer that's the same size as the pixels we're writing. So we want a UCHAR* for 8-bit color depth, a USHORT* for 16-bit, and a UINT* for 32-bit. But what about 24-bit? Since there's no 24-bit data type, we'll need to use a UCHAR* for this also, and do things a little differently.

We should also convert the lPitch member so it's in the same units as our pointer. Remember, when we first retrieve lPitch from the DDSURFACEDESC2 structure, it's in bytes. For 16-bit mode, we should divide it by 2 to get the pitch in terms of USHORTs, and for 32-bit mode, we divide by 4 to get it in terms of UINTs.

Let's stop for a second and look at some example code. Suppose we are in 32-bit mode and want to lock the primary surface for plotting pixels. Here's what we would do:

```
// declare and initialize structure
DDSURFACEDESC2 ddsd;
INIT_DXSTRUCT(ddsd);

// lock the surface
lpddsPrimary->Lock(NULL, &ddsd, DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR, NULL);

// now convert the pointer and the pitch
UINT* buffer = (UINT*)ddsd.lpSurface;
```

```
    UINT nPitch = ddsd.lPitch >> 2;
```

Now let me go ahead and show you a pixel-plotting function, and then I'll explain it in detail:

```
    inline void PlotPixel32(int x, int y, UINT color, UINT *buffer, int nPitch)
    {
      buffer[y*nPitch + x] = color;
    }
```

All right, let's take it apart. First of all, notice that I have declared it as an inline function. This is to eliminate the overhead of passing all the parameters every time we want to do something as simple as plotting one pixel. Now, the only line in the function locates the pixel of interest and sets it to the color we want. Note that the color is just one value, not one each for red, green, and blue, so we'll need to use our `RGB_32BIT()` macro to create the color for this function.

The formula used to locate the pixel at location (`x`, `y`) is `y*nPitch + x`. This makes sense because `nPitch` is the number of `UINT`s in a line. Multiplying that by the `y` value yields the correct row, and then we just need to add `x` to locate the correct column. That's all you need to know! Pretty simple, hey? Let me show you a couple of functions for plotting pixels in 16-bit and 8-bit modes, because they're very similar:

```
    inline void PlotPixel8(int x, int y, UCHAR color, UCHAR* buffer, int byPitch)
    {
      buffer[y*byPitch + x] = color;
    }

    inline void PlotPixel16(int x, int y, USHORT color, USHORT* buffer, int nPitch)
    {
      buffer[y*nPitch + x] = color;
    }
```

The only things different about these functions are the data types used for the parameters. Also remember that for the 8-bit version, the pitch is in bytes, and for the 16-bit version, the pitch is in `USHORT`s. That just leaves one mode left, and that's 24-bit. Since there's no data type that's 24 bits, we need to pass and write values for red, green, and blue individually. The function might look like this:

```
    inline void PlotPixel24(int x, int y, UCHAR r, UCHAR g, UCHAR b, UCHAR* buffer, int byPitch)
    {
      int index = y*byPitch + x*3;

      buffer[index] = r;
      buffer[index+1] = g;
      buffer[index+2] = b;
    }
```

As you can see, this is going to be a bit slower because we have an extra multiply, and three memory writes. You can speed it up a little bit by replacing `x*3` with `(x+x+x)` or `(x<<1)+x`, but that probably won't do a lot for you. Still, it can't hurt to throw it in for good measure. Now you can see why working with 24-bit color is a bit of a pain.

Notes on Speed

There are a few things you should do to make sure this all goes as quickly as possible. First of all, locking a surface is not the fastest thing in the world, so you should try to keep the number of times you lock a surface per frame to a minimum. For many things, including a simple pixel-plotting demo, one lock per frame is all you'll need.

Second, let's say you're working in 640x480x16. The pitch will almost always be 1,280 bytes. You still need to take into consideration that it might not always be that way, but you may want to add some optimized code to take advantage of the fact that the pitch will almost always be 1,280. More specifically, you can use a pixel-plotting function that uses bit shifts instead of multiplication to locate the correct pixel. The function we were using earlier used this line of code:

```
buffer[y*nPitch + x] = color;
```

We can speed that up if we know `nPitch` is going to be 640 (since `nPitch` is in `USHORT`s, not bytes). 640 isn't a power of two, but 512 is 2^9, and 128 is 2^7. And guess what? 512 + 128 = 640. :) Convenient, hey? The line above will execute just a tad faster if we use this instead:

```
buffer[(y<<9) + (y<<7) + x] = color;
```

Most resolutions you'll use have widths that are either powers of two, or a sum of two powers of two. No such luck if you're in 800x600 (512 + 256 + 32 = 800), but it's a good trick to keep in mind. Bit shifts are some of the fastest operations you could ask for.

Finally, if you are going to be using two functions -- one using multiplication, and the other using bit-shifts -- keep the comparison outside of the plotting loop. That is, don't do this:

```
for (x=0; x<1000; x++)
{
  if (nPitch == 640)
    PlotPixelFast16();
  else
    PlotPixel16();
}
```

Putting the decision statement inside the loop is just detracting from the advantage you gain with the fast function. Do it this way instead:

```
if (nPitch == 640)
{
  for (x=0; x<1000; x++)
    PlotPixelFast16( parameters );
}
else
{
  for (x=0; x<1000; x++)
    PlotPixel16( parameters );
}
```

Make sense? Whenever you've got a big for loop, keep as much as you can on the outside of it. There's no sense in making a calculation a thousand times that's going to come up the same way every time. :) Also, if you're going to be plotting pixels in such a way that the locations are in a pattern, like a horizontal or vertical line (or even a diagonal one), there's no need to re-calculate the location every time. Look at this example, for drawing a line of randomly colored pixels:

```
for (x=0; x<640; x++)
    PlotPixel16(x, 50, color, buffer, pitch);
```

The function is re-calculating the correct line every time, when all you need to do is advance the pointer by one each time. This is a faster way of doing it:

```
// get the address of the line
USHORT* temp = &buffer[50*pitch];

// plot the pixels
```

```
    for (x=0; x<640; x++)
    {
      *temp = color;
      temp++;
    }
```

None of these things are going to make a huge difference in your program's speed unless you're plotting thousands and thousands of pixels each frame -- which you might be doing -- but even if not, it's always good policy to make any little optimization you can find, no matter how insignificant it seems.

Considering how long the previous articles in this series have been, stopping this one here would seem like kind of a rip-off, wouldn't it? Now that we know how to plot pixels, let's take a look at some of the things you can use it for.

Fading Out

One of the most commonly used screen transitions in games is the fade to black, or the fade in from black. Each one is achieved in the same manner. You simply draw your frame, then apply some sort of transformation that alters the brightness of the image. To fade out, you decrease the brightness from 100% to 0%, and to fade in, you increase it from 0% to 100%. If you're working in a palettized mode, you've got it easy! You just change the colors in your palette, and the result shows up onscreen. If you're working in RGB mode, you have to consider other methods.

Now, I'll say right here that manually fading the screen has some better alternatives. You can use Direct3D, which supports alpha-blending, to set each frame as a texture, then set the transparency level. Or, even easier, you can use the DirectDraw color/gamma controls. But if you only want to fade part of the screen, or fade to a color besides black, and you're not a Direct3D expert -- I'm certainly not! -- then knowing how to do the manual transform might come in handy.

Basically all you do is to read each pixel and break it up into red, green, and blue. Then you multiply each of the three values by the level of fading to apply, recombine the RGB values, and write the new color back out to the buffer. Sound complicated? It's not too bad. Take a look at this (highly unoptimized) sample code. It applies a fade to a viewport of size 200x200 in the upper-left corner of the display, in a 16-bit color depth using the 565 pixel format.

```
    void ApplyFade16_565(float pct, USHORT* buffer, int pitch)
    {
      int x, y;
      UCHAR r, g, b;
      USHORT color;

      for (y=0; y<200; y++)
      {
        for (x=0; x<200; x++)
        {
          // first, get the pixel
          color = buffer[y*pitch + x];

          // now extract red, green, and blue
          r = (color & 0xF800) >> 11;
          g = (color & 0x0730) >> 5;
          b = (color & 0x001F);

          // apply the fade
          r = (UCHAR)((float)r * pct);
          g = (UCHAR)((float)g * pct);
          b = (UCHAR)((float)b * pct);
          // write the new color back to the buffer
          buffer[y*pitch + x] = RGB_16BIT565(r, g, b);
        }
      }
    }
```

Now, there are a number of things wrong with this function. First, not only is the calculation of the pixel location inside the inner loop, it's in there twice! You can get away with calculating it **once** in the entire function, but in this example it's calculated 80,000 times. :) Here's what you should do. At the very top of the function, you should declare another USHORT*, and set it equal to buffer. Inside the inner loop, increment the pointer by one to get to the next pixel. And inside the outer loop, increment it by whatever it takes to get down to the next line. Here's the considerably better example:

```
void ApplyFade16_565(float pct, USHORT* buffer, int pitch)
{
  int x, y;
  UCHAR r, g, b;
  USHORT color;
  USHORT* temp = buffer;
  int jump = pitch - 200;

  for (y=0; y<200; y++)
  {
    for (x=0; x<200; x++, temp++) // move pointer to next pixel each time
    {
      // first, get the pixel
      color = *temp;

      // now extract red, green, and blue
      r = (color & 0xF800) >> 11;
      g = (color & 0x0730) >> 5;
      b = (color & 0x001F);

      // apply the fade
      r = (UCHAR)((float)r * pct);
      g = (UCHAR)((float)g * pct);
      b = (UCHAR)((float)b * pct);

      // write the new color back to the buffer
      *temp = RGB_16BIT565(r, g, b);
    }

    // move pointer to beginning of next line
    temp+=jump;
  }
}
```

This is better. The value of jump is the number of USHORTs from the end of a line on the 200-pixel-wide viewport to the beginning of the next line. Still, there's not much you can do to speed up those floating-point multiplies and all the extracting/recombining of colors. Or is there? Take a look at this:

```
USHORT clut[65536][20];
```

If you asked a DOS programmer to put an array that size into one of his programs, he'd probably cry out in pain. He might even drop dead on the spot, or spontaneously combust. But in Windows, it's not really a problem if you need to do it, because you have all the system's free RAM available to you. Wouldn't it be nice if you could replace the entire inner loop with this one line?

```
*temp = clut[*temp][index];
```

That would speed things up a bit, right? :) Instead of passing a float to the function, you could pass an integer that's between 0 and 100. If the value is 100, no fade is needed, so return without doing anything. If the value is 0, just use a `ZeroMemory()` call to do all the work. Otherwise, divide the value by 5, and use it for the second subscript into the giant lookup table.

If you're wondering where I got the dimensions for the lookup table, 65536 is 2^16, so that's how many colors there are in a 16-bit color depth. Since our color values are unsigned values, they range from 0 to 65535. The 20 is just the number of increments you'd be using for a fade. It seemed a decent choice to me, considering the memory involved.

For 24-bit and 32-bit color, you obviously can't create a lookup table to span every color combination, so what you could do is use three smaller tables:

```
UCHAR red[256];
UCHAR green[256];
UCHAR blue[256];
```

Then, when you read each pixel, split it up into its color components, look each component up in its table, and recombine them into the resulting color. There are all sorts of ways to do things like this. The best way to determine what works well for your purposes is just to throw some code together, and then play around with it for awhile. With that in mind, I will briefly go over one other transformation you might find useful.

Basic Transparency

Overlaying a transparent image on an opaque one is not something you can use a lookup table for, because it would have to have 65,536 entries in **both** dimensions. It's going to be awhile before the average computer has the 8.6 GB of RAM it takes to hold that monster. :) So you'll have to do all the calculations for each pixel. I'll give you the basic idea. Suppose you want to place image A on top of image B, and image A is to have a transparency percentage of `pct`, which is a floating-point number between 0 and 1, where 0 is fully transparent (invisible) and 1 is fully opaque. Then, let's call a pixel in image A `pixelA`, and its counterpart in image B we'll call `pixelB`. You would apply the following equation:

```
color = (pixelA * pct) + (pixelB * (1-pct));
```

Basically, this is just a weighted average of the two pixel colors. By multiplying the colors, I mean to multiply the intensities for red, green, and blue. So you're actually looking at six floating-point multiplications per pixel. You can use some small lookup tables to lighten the workload a little. Experiment with it!

The other thing you might want to do is to create a window of a solid color that's partially transparent. If you've seen a demo or screenshots of my upcoming RPG, Terran, you know what I mean. An effect like that can be done entirely with a lookup table, because in Terran's case, I just needed to provide a color of blue for every possible color on the screen. In fact, a lookup table is exactly how I do the effect. I'll show you exactly what I mean.

```
void Init_CLUT(void)
{
  int x, y, bright;
  UCHAR r, g, b;

  // calculate textbox transparency CLUT
  for (x=0; x<65536; x++)
  {
    // transform RGB data
    if (color_depth == 15)
    {
      r = (UCHAR)((x & 0x7C00) >> 10);
```

```
          g = (UCHAR)((x & 0x03E0) >> 5);
          b = (UCHAR)(x & 0x001F);

        }
        else  // color_depth must be 16
        {
          r = (UCHAR)((x & 0xF800) >> 11);
          g = (UCHAR)((x & 0x07E0) >> 6);   // shifting 6 bits instead of 5 to put green
          b = (UCHAR)(x & 0x001F);          // on a 0-31 scale instead of 0-63
        }

        // find brightness as a weighted average
        y = (int)r + (int)g + (int)b;
        bright = (int)((float)r * ((float)r/(float)y) +
                       (float)g * ((float)g/(float)y) +
                       (float)b * ((float)b/(float)y) + .5f);

        // write CLUT entry as 1 + one half of brightness
        clut[x] = (USHORT)(1 + (bright>>1));
      }
    }
```

This is the code from Terran that creates the lookup table with which the text boxes are generated. There are typecasts everywhere just for safety, and it could be sped up quite a bit, but I didn't bother because it's only called once, in the very beginning of the game. First, the values for red, green, and blue are extracted. Since this is in 16-bit color, notice that I have a variable called `color_depth` that takes into account whether the pixel format is 565 or 555. Then, the brightness of the pixel is calculated using this formula:

```
    y = r + g + b;
    brightness = r*(r/y) + g*(g/y) + b*(b/y);
```

It's just another weighted average. I'm not sure if that's really how the brightness of a color is defined, but it seems logical and it produces a nice effect. Anyway, at the end of the equation I've added .5, because when you cast a `float` to an `int`, the decimal is truncated. Adding .5 turns that truncation into rounding. Finally, I divide the brightness in half and add one. The division is so the text boxes don't get too bright, and the one is so the box is never totally black. Since the low bits of a 16-bit color descriptor are for blue, I can just set the color by setting the value of blue, instead of having to use a macro. Make sense? Finally, before I wrap this up, I'll also show you how I create a text box:

```
    int Text_Box(USHORT *ptr, int pitch, LPRECT box)
    {
        int x, y, jump;
        RECT ibox;

        // leave room for the border
        SetRect(&ibox, box->left+3, box->top+3, box->right-3, box->bottom-3);

        // update surface pointer and jump distance
        ptr += (ibox.top * pitch + ibox.left);
        jump = pitch - (ibox.right - ibox.left);

        // use CLUT to apply transparency
        for (y=ibox.top; y<ibox.bottom; y++)
        {
            for (x=ibox.left; x<ibox.right; x++, ptr++)
                *ptr = clut[*ptr];
            ptr += jump;
        }
        return(TRUE);
    }
```

Now that it's just a lookup table, this looks a lot like the code for fading. The only difference is that the lookup table holds different values. And it's only one column instead of 20. :) The declaration for the lookup table, by the way, looks like this:

```
    USHORT clut[65536];
```

With that, you should be able to produce some rather interesting effects. To get you started, check out the sample code that comes with this article. It's available here. You might try modifying it so that it fills the screen with pixels, then plots a transparent box over the top of them.

Closing

That does it for pixel-based graphics. Next time around, we'll be working with bitmaps! Believe it or not, working with bitmaps is easier than all this pixel stuff. Seems a little backwards, doesn't it? You'll find out. In the meantime, send me any questions you might have and I'll be happy to help you out. My E-mail address is ironblayde@aeon-software.com, and my ICQ UIN is 53210499. Oh, one other thing... the next article will be the last one covering general DirectX techniques. After that, we'll get into specific applications that you can use for developing an RPG. More details to follow. :) Later!

Discuss this article in the forums