Game Programming Genesis
Part VI : Bitmapped Graphics in DirectDraw

by Joseph "Ironblayde" Farrell

Introduction

At last, the good stuff! You already know enough to make a fully functioning Windows game, but it would have to use GDI. Today we're going to go over the DirectX implementations of everything you know how to do using GDI, and quite a bit more. I'm going to cover loading bitmaps, using the blitter to fill a surface with a color, and copying bitmaps like lightning, with clipping, color keys, and a host of other effects.

None of the material we'll be covering today will draw on the most recent article, so it's not necessary that you've read it. However, the section on pixel formats was very important, and I'll allude to it from time to time, so at least read that part. Aside from that, I assume you've read the first four articles in the series, and have the DirectX 7 SDK. All right? Fire up your compilers, ladies and gentlemen. :)

Loading Bitmaps

Believe it or not, you already know almost everything you need to load a bitmap onto a DirectDraw surface. How can that be? Well, the method you used under Windows GDI will work in DirectDraw as well, with only one little change. To refresh your memory a little bit, what we did was to retrieve a handle to the bitmap using `LoadImage()`, select the bitmap into a memory device context, and then use `BitBlt()` to copy the image from the memory DC to another DC, which was a display device context we had gotten with a call to `GetDC()`. If that destination DC was a device context to a DirectDraw surface, we'd have our DirectX bitmap loader all done! Thankfully, the `IDirectDrawSurface7` interface provides a simple function for retrieving a device context:

```
HRESULT GetDC(HDC FAR *lphDC);
```

The return type is the same as for any DirectDraw function, and the parameter is just a pointer to an `HDC` which will be initialized with the device context handle if the function succeeds. Is that cool or what? This article just started and we can already load a bitmap onto one of our surfaces! Just remember that when you're all done with the surface device context, you need to release it. As you've probably guessed, this is achieved with the `ReleaseDC()` method of the surface interface:

```
HRESULT ReleaseDC(HDC hDC);
```

Just so you don't have to go rooting through the old articles, looking for the GDI bitmap loader, I'll show the modified version to you here. The only difference is that instead of taking a device context as a parameter, it takes a pointer to a DirectDraw surface. Then the function gets the device context from the surface, uses it to copy the image, and releases the device context.

```
int LoadBitmapResource(LPDIRECTDRAWSURFACE7 lpdds, int xDest, int yDest, int nResID)
{
  HDC hSrcDC;           // source DC - memory device context
  HDC hDestDC;          // destination DC - surface device context
```

```
    HBITMAP hbitmap;      // handle to the bitmap resource
    BITMAP bmp;           // structure for bitmap info
    int nHeight, nWidth;  // bitmap dimensions

    // first load the bitmap resource
    if ((hbitmap = (HBITMAP)LoadImage(hinstance, MAKEINTRESOURCE(nResID),
                                  IMAGE_BITMAP, 0, 0,
                                  LR_CREATEDIBSECTION)) == NULL)
      return(FALSE);

    // create a DC for the bitmap to use
    if ((hSrcDC = CreateCompatibleDC(NULL)) == NULL)
      return(FALSE);

    // select the bitmap into the DC
    if (SelectObject(hSrcDC, hbitmap) == NULL)
    {
      DeleteDC(hSrcDC);
      return(FALSE);
    }

    // get image dimensions
    if (GetObject(hbitmap, sizeof(BITMAP), &bmp) == 0)
    {
      DeleteDC(hSrcDC);
      return(FALSE);
    }

    nWidth = bmp.bmWidth;
    nHeight = bmp.bmHeight;

    // retrieve surface DC
    if (FAILED(lpdds->GetDC(&hDestDC)))
    {
      DeleteDC(hSrcDC);
      return(FALSE);
    }

    // copy image from one DC to the other
    if (BitBlt(hDestDC, xDest, yDest, nWidth, nHeight, hSrcDC, 0, 0,
               SRCCOPY) == NULL)
    {
      lpdds->ReleaseDC(hDestDC);
      DeleteDC(hSrcDC);
      return(FALSE);
    }

    // kill the device contexts
    lpdds->ReleaseDC(hDestDC);
    DeleteDC(hSrcDC);

    // return success
    return(TRUE);
  }
```

This function is set to load from a resource, but you can easily modify it to load from an external file. Or better still, you could have it try to load a resource, and if the call fails, it retries as a file. Just remember to

include the LR_LOADFROMFILE flag in the call to LoadImage(). The nicest thing about this function is that BitBlt() performs all the conversions on the pixel formats. That is, you can load a 24-bit bitmap into the memory device context, and blit it to a 16-bit surface, and all the colors will show up correctly, regardless of whether the pixel format is 565 or 555. Convenient, hey?

If you want to manipulate the actual bitmap data manually instead of simply using a function to copy, you have two options. First, you can use a modified version of the function above, and use the bmBits member of the BITMAP structure, which is an LPVOID pointing to the bits making up the image. Second, if you really want to have control of how the load is performed, you can write a function that opens the file and reads it in manually, using standard file I/O functions. To do that, you need to know the structure of a bitmap file. I'm not going to go through developing the whole function, since we already have the functionality we need, but I'll show you everything you need to do so.

The Bitmap File Format

The nice thing about writing a bitmap loader is that there are Win32 structures designed to hold the bitmap headers, so loading all the header info as simple as making a few calls to fread(). First up in a bitmap file is the bitmap file header, which contains general information about the bitmap. Not surprisingly, the structure that holds this header is called BITMAPFILEHEADER. Here's what it looks like:

```
typedef struct tagBITMAPFILEHEADER { // bmfh
  WORD    bfType;        // file type - must be "BM" for bitmap
  DWORD   bfSize;        // size in bytes of the bitmap file
  WORD    bfReserved1;   // must be zero
  WORD    bfReserved2;   // must be zero
  DWORD   bfOffBits;     // offset in bytes from the BITMAPFILEHEADER
                           // structure to the bitmap bits
} BITMAPFILEHEADER;
```

I'm not going to detail all the members; I've commented the structure to give you an idea of what everything is. Just use a call to fread() to read this in, and check the bfType member to make sure it is equal to the characters "BM" to ensure you're dealing with a valid bitmap. After that, there's another header file to read, called the info header. It contains image data like the dimensions, compression type, etc. Here's the structure:

```
typedef struct tagBITMAPINFOHEADER{ // bmih
  DWORD  biSize;            // number of bytes required by the structure
  LONG   biWidth;           // width of the image in pixels
  LONG   biHeight;          // height of the image in pixels
  WORD   biPlanes;          // number of planes for target device - must be 1
  WORD   biBitCount;        // bits per pixel - 1, 4, 8, 16, 24, or 32
  DWORD  biCompression;     // type of compression - BI_RGB for uncompressed
  DWORD  biSizeImage;       // size in bytes of the image
  LONG   biXPelsPerMeter;   // horizontal resolution in pixels per meter
  LONG   biYPelsPerMeter;   // vertical resolution in pixels per meter
  DWORD  biClrUsed;         // number of colors used
  DWORD  biClrImportant;    // number of colors that are important
} BITMAPINFOHEADER;
```

A few of the fields need some explanation. First, a note on compression. Most bitmaps that you'll be dealing with will be uncompressed. The most common type of compression for bitmaps is run-length encoding (RLE), but this is only used for 4-bit or 8-bit images, in which case the biCompression member will be

BI_RLE4 or BI_RLE8, respectively. I'm not going to go into run-length encoding, but it's a fairly straightforward method of compression, so it's not too hard to deal with if you come across it.

Second, biClrUsed and biClrImportant will usually be set to zero in high-color bitmaps, so don't worry about them too much. The biSizeImage field may also be set to zero in some BI_RGB uncompressed bitmaps. Finally, the resolution fields are also unimportant for our purposes. Mainly the only things you're interested in from this structure are the width, height, and color depth of the image.

After you've read in the info header, if the bitmap has eight bits per pixel or less, it is palettized, and the palette information immediately follows the info header. The palette information, however, is not stored in PALETTEENTRY structures, but rather in RGBQUAD structures. An RGBQUAD looks like this:

```
typedef struct tagRGBQUAD { // rgbq
  BYTE    rgbBlue;
  BYTE    rgbGreen;
  BYTE    rgbRed;
  BYTE    rgbReserved;
} RGBQUAD;
```

Don't ask me why the red, green, and blue intensities are stored in reverse order, but they are. Just read in the RGBQUADs, and transfer the data into an array full of PALETTEENTRYs to create a DirectDraw palette with. Remember to set the peFlags member of each PALETTEENTRY to PC_NOCOLLAPSE as you do so.

After the palette, or immediately following the info header if there is no palette, you'll find the actual image bits. You'll probably want to simply create a pointer and allocate enough memory to it to hold the image data, and then read it in. Just to be clear, I'll show you the code for doing this, assuming the info header is stored in a BITMAPINFOHEADER structure called info, and your file pointer is called fptr:

```
UCHAR* buffer = (UCHAR*)malloc(info.biSizeImage);
fread(buffer, sizeof(UCHAR), info.biSizeImage, fptr);
```

Just remember that Microsoft warns that biSizeImage may be set to zero in some cases, so check it before running code like the above. If it is set to zero, you'll have to calculate the size of the image by figuring out how many pixels comprise the image, and how many bytes are required for each pixel.

Writing your own bitmap loader isn't too bad, but if you want to avoid it, you can always use the code we developed back in the GDI article and modified at the beginning of this one. Now that that's over with, let's get into what DirectDraw's all about: using the blitter!

Using the Blitter

The blitter is a part of the video card hardware that is used for manipulating bitmap data. You can also use it to do color fills, as we'll see in a minute, and any number of other cool tricks if the hardware supports them. Having easy access to hardware acceleration is one of the best parts about DirectX. Also, remember that most of the things we'll be doing will be taken care of in the HEL if there's no hardware support. There are some things that don't have counterparts in the HEL though, which is why you have to be careful to always check whether your calls succeed or not.

There are two main functions for accessing the blitter in DirectDraw: Blt() and BltFast(). Both are methods of the IDirectDrawSurface7 interface. The difference is that BltFast() doesn't do clipping, scaling, or any of the other interesting things that Blt() does. The advantage is about a 10% speed

increase over Blt() if the HEL is being used. If hardware acceleration is supported, though -- and it almost always is for blitting operations -- there is no speed difference for average blits, so I use Blt() for just about everything. Let's take a look at it:

```
HRESULT Blt(
  LPRECT lpDestRect,
  LPDIRECTDRAWSURFACE7 lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwFlags,
  LPDDBLTFX lpDDBltFx
);
```

Because Blt() can do all sorts of special effects, as evidenced by that last parameter, it's got a few long lists of flags associated with it. I'll show you what I've found to be the most useful ones. Also, note that when you're blitting from one surface to another, you call the Blt() method of the destination surface, not the source surface. All right? Here are the function's parameters:

LPRECT lpDestRect: This is the destination RECT to blit to. If it differs in size from the source RECT, Blt() will automatically scale the image in the source RECT to fit the destination RECT! If the destination is the entire surface, you can set this to NULL.

LPDIRECTDRAWSURFACE7 lpDDSrcSurface: This is the source surface of the blit. If you're using the blitter to do a color fill on the destination surface, you can set this parameter to NULL.

LPRECT lpSrcRect: This is the source RECT to blit from. If you mean to blit the entire contents of the surface, set this to NULL.

DWORD dwFlags: There's a huge list of flags for this parameter, which can be logically combined with the | operator. Quite a few of them have to do with Direct3D stuff (like alpha information), so I'll show you a partial list here.

DDBLT_ASYNC	Performs the blit asynchronously through the FIFO (first in, first out) in the order received. If no room is available in the FIFO hardware, the call fails, so be careful using this one.
DDBLT_COLORFILL	Fills the destination rectangle with the color denoted in the dwFillColor member of the DDBLTFX structure.
DDBLT_DDFX	Uses the dwDDFX member of the DDBLTFX structure to specify effects used for the blit.
DDBLT_DDROPS	Uses the dwDDROPS member of the DDBLTFX structure to specify raster operations (ROPs) that are not part of the Win32 API.
DDBLT_KEYDEST	Uses the color key associated with the destination surface.
DDCLT_KEYDESTOVERRIDE	Uses the dckDestColorKey member of the DDBLTFX structure as the color key for the destination surface.
DDBLT_KEYSRC	Uses the color key associated with the source surface.
DDCLT_KEYSRCOVERRIDE	Uses the dckDestColorKey member of the DDBLTFX structure as the color key for the source surface.

DDBLT_ROP	Uses the dwROP member of the DDBLTFX structure for the ROP for the blit. These ROPs are those defined in the Win32 API.
DDBLT_ROTATIONANGLE	Uses the dwRotationAngle member of the DDBLTFX structure as the rotation angle (specified in hundreths of a degree) for the surface.
DDBLT_WAIT	Waits for the blitter to be available instead of returning an error, if the blitter was drawing at the time this blit was called. The function returns when the blit is complete, or another error occurs.

I almost always use DDBLT_WAIT. The color key flags are also important; we'll get to them in just a bit. Now, though, here's the last parameter to Blt():

LPDDBLTFX lpDDBltFx: A pointer to a DDBLTFX structure, which can contain all sorts of special effects information. If no effects are specified using this structure, you can pass NULL. Let's take a look at the structure. I'm warning you, it's massive!

```
typedef struct _DDBLTFX{
  DWORD dwSize;
  DWORD dwDDFX;
  DWORD dwROP;
  DWORD dwDDROP;
  DWORD dwRotationAngle;
  DWORD dwZBufferOpCode;
  DWORD dwZBufferLow;
  DWORD dwZBufferHigh;
  DWORD dwZBufferBaseDest;
  DWORD dwZDestConstBitDepth;
  union {
    DWORD                dwZDestConst;
    LPDIRECTDRAWSURFACE lpDDSZBufferDest;
  };
  DWORD dwZSrcConstBitDepth;
  union {
    DWORD                dwZSrcConst;
    LPDIRECTDRAWSURFACE lpDDSZBufferSrc;
  };
  DWORD dwAlphaEdgeBlendBitDepth;
  DWORD dwAlphaEdgeBlend;
  DWORD dwReserved;
  DWORD dwAlphaDestConstBitDepth;
  union {
    DWORD                dwAlphaDestConst;
    LPDIRECTDRAWSURFACE lpDDSAlphaDest;
  };
  DWORD dwAlphaSrcConstBitDepth;
  union {
    DWORD                dwAlphaSrcConst;
    LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
  };
  union {
    DWORD                dwFillColor;
    DWORD                dwFillDepth;
    DWORD                dwFillPixel;
```

```
        LPDIRECTDRAWSURFACE lpDDSPattern;
      };
      DDCOLORKEY ddckDestColorkey;
      DDCOLORKEY ddckSrcColorkey;
    } DDBLTFX, FAR* LPDDBLTFX;
```

If I went through this whole structure, we'd all be standing in line for our AARP cards by the time I got finished. So I'll just go over the important ones. Thankfully, most of this structure is for z-buffers and alpha information, which don't concern us. It makes my job a little easier. :)

DWORD dwSize: Like all DirectX structures, this member has to be set to the size of the structure when you initialize it.

DWORD dwDDFX: These are kinds of effects that can be applied to the blit. The list isn't too long, don't worry.

DDBLTFX_ARITHSTRETCHY	The blit uses arithmetic stretching along the y-axis.
DDBLTFX_MIRRORLEFTRIGHT	The blit mirrors the surface from left to right.
DDBLTFX_MIRRORUPDOWN	The blit mirrors the surface from top to bottom.
DDBLTFX_NOTEARING	Schedules the blit to avoid tearing.
DDBLTFX_ROTATE180	Rotates the surface 180 degrees clockwise during the blit.
DDBLTFX_ROTATE270	Rotates the surface 270 degrees clockwise during the blit.
DDBLTFX_ROTATE90	Rotates the surface 90 degrees clockwise during the blit.

The only one that might need some explanation is DDBLTFX_NOTEARING. Tearing is what can happen when you blit a surface out of sync with the vertical blank. If you blit to the entire primary surface while it's being drawn, you can see the top half of the old frame along with the bottom half of your updated frame. In my experience, this has almost never been a problem. On with the DDBLTFX structure...

DWORD dwROP: You use this flag to specify Win32 raster operations, like those that can be used with the GDI functions BitBlt() and StretchBlt(). Most of them have to do with combining source and destination images using Boolean operators. You can call IDirectDraw7::GetCaps() to retrieve a list of supported raster ops, among other things.

DWORD dwRotationAngle: This is an angle by which to rotate the bitmap, specified in hundredths of a degree. This is pretty cool, but unfortunately, rotation is only supported in the HAL, which means that if the user's video card doesn't support accelerated rotation, it won't work at all. Hardware support for this is not too common, either, so the bottom line is that you shouldn't use this in a program you're going to distribute. If you really need rotation, you'll have to write your own code for it. Doing that is a topic for another entire tutorial, though, so we'll just pass it by for now. But note that rotations by multiples of ninety degrees are easy, so using DDBLTFX_ROTATE90, etc. instead of using dwRotationAngle will work no matter how lousy the user's video card is.

DWORD dwFillColor: When you're using the blitter to perform a color fill, you must set the color to be used in this parameter.

DDCOLORKEY ddckDestColorKey, ddckSrcColorKey: Specify these members when you want to use a destination or source color key other than the one specified for the surface involved. These guys are important, but we can't talk about them just yet because we haven't covered color keys. It's coming soon

enough...

That about does it for the immediately useful members of the DDBLTFX structure, which means you now have enough information to go ahead and start blitting images! If it's all rather confusing at this point, don't worry, it will become second nature once you use it a couple of times. Let's look at an example or two. Suppose you have a back buffer called lpddsBack, and you want to blit its contents to the primary surface. Here's the call:

```
lpddsPrimary->Blt(NULL, lpddsBack, NULL, DDBLT_WAIT, NULL);
```

How easy is that? To refresh your memory a bit, the first and third parameters are the destination and source RECTs for the blit, respectively. Since I have specified NULL for each, the entire surface is copied. Now, let's say you have a tile that's 16x16 in size in an offscreen surface called lpddsTileset, and you want to blit it to the back buffer, scaling it up to 32x32. Here's what you might do:

```
RECT dest, src;
SetRect(&src, 0, 0, 16, 16);   // the coordinates of the tile
SetRect(&dest, 0, 0, 32, 32);  // where you want it to end up on the back buffer
lpddsBack->Blt(&dest, lpddsTileset, &src, DDBLT_WAIT, NULL);
```

The only difference between this example and the last one is that we've specified the coordinates to be used in the blit. Since the two RECTs are of different sizes, Blt() scales the image appropriately. Finally, to illustrate using the DDBLTFX structure, let's do a color fill. Suppose you're in 16-bit color, with a 565 pixel format, and you want to fill your back buffer with blue. This is all you need:

```
DDBLTFX fx;
INIT_DXSTRUCT(fx);                          // zero out the structure and set dwSize
fx.dwFillColor = RGB_16BIT565(0, 0, 31);  // set fill color to blue
lpddsBack->Blt(NULL, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL, &fx);
```

Note that since there is no source surface involved, the second parameter is set to NULL. Got it? All right, let's take a look at the alternative blitting function, BltFast(). It's basically a stripped-down version of Blt(), so we don't need to spend nearly as much time on it.

```
HRESULT BltFast(
  DWORD dwX,
  DWORD dwY,
  LPDIRECTDRAWSURFACE7 lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwTrans
);
```

As you can see, it's very similar to Blt(). This is also a member function of the IDirectDrawSurface7 interface, and should be called as a method of the destination surface. Let's have a look at the parameters:

DWORD dwX, dwY: Here's a difference between Blt() and BltFast(). BltFast() uses these x- and y-coordinates to specify the destination of the blit, rather than a RECT. This is because you can't do scaling with BltFast().

LPDIRECTDRAWSURFACE7 lpDDSrcSurface: This is the source surface, just like before.

LPRECT lpSrcRect: This is also what we used in `Blt()`, the source `RECT` for the blit.

DWORD dwTrans: This parameter takes one or more of the flags listed below, and specifies the type of transfer to perform. The list is very simple; there are only four possible values you can use.

DDBLTFAST_DESTCOLORKEY	Specifies a transparent blit that uses the destination's color key.
DDBLTFAST_NOCOLORKEY	Specifies a normal copy blit with no transparency.
DDBLTFAST_SRCCOLORKEY	Specifies a transparent blit that uses the source's color key.
DDBLTFAST_WAIT	Waits for the blitter to be available instead of returning an error, if the blitter was drawing at the time this blit was called. The function returns when the blit is complete, or another error occurs.

That's it! `BltFast()` supports color keys, and that's about it. Just as a quick demonstration, let's look at how you would do the first example I gave above with `BltFast()`. Here's how you would copy your entire back buffer onto the primary surface:

```
lpddsPrimary->BltFast(0, 0, lpddsBack, NULL, DDBLTFAST_WAIT);
```

Now that you're an expert on using the blitter, there are a few things we can cover that are very useful in any DirectDraw program: color keys and clipping. Since you've seen a million different flags concerning color keys in some way and are probably wondering how to make use of them, let's cover that first.

Color Keys

A color key is a method of blitting one image onto another whereby not all of the pixels are copied. For instance, let's say you have a character sprite you want to blit onto a background image. Chances are that your hero isn't shaped precisely like a rectangle (unless your game is a celebration of "modern art"), so copying the `RECT` that encloses him is going to produce some unwanted effects. Take a look at this example picture from Terran to see what I mean:

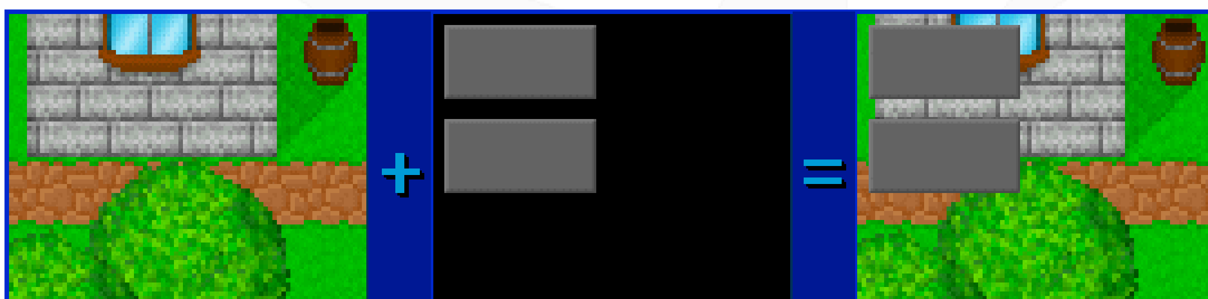
Now this isn't really the way the game is drawn. In reality, the characters are blitted to the display before the background has finished being drawn, so that things like the tops of trees can obscure the player if he's standing behind them. But that's unimportant for our purposes here; we'll get to it in the next article. The important point is that without color keying, all of your non-rectangular images like this character would have boxes around them.

To solve this problem, we use a source color key. The source color key tells the blitter which colors not to copy. A color key consists of two values: a lower color value, and an upper color value. When the color key is applied to a blit, any colors lying between the two values, including the values themselves, are not copied. There's a structure in DirectX that goes along with this, called `DDCOLORKEY`. Take a look:

```
typedef struct _DDCOLORKEY{
  DWORD dwColorSpaceLowValue;
  DWORD dwColorSpaceHighValue;
} DDCOLORKEY, FAR* LPDDCOLORKEY;
```

I just told you what the members are so there's no need to go over them further. Let me show you what the above blit would look like with a color key activated. In Terran, I use color keys with the low and high values both set to black. Thus, black is the only color that doesn't get copied. When this color key is applied to the blit, this is the result:

Much better, hey? That's the look we're going for! Now before I show you how to create and use a color key, I want to briefly mention destination color keys, although they're not often used. Whereas source color keys define which colors are not copied, destination color keys define which colors can't be written to. Sound weird? It is. :) Destination color keying would be used if you wanted to blit one image to a position where it looks like it's underneath another. Suppose that for some odd reason, you wanted to draw text boxes on your blank back buffer, and then copy the background image afterwards, but you still wanted the text boxes to appear on top of the map. What you could do is set a destination color key to include every color except black. Thus, the only areas on the back buffer that can be written to are those pixels which are currently black. Take a look at this illustration to see what I mean:



Like I said, I'm not sure why you'd want to do that, but there it is. Maybe you can think up a good use for it. If you do, be sure to let me know. :) Now that you know what color keys are, let's see how you use them.

Setting Color Keys

There are two ways to use a color key in DirectDraw. First, you can attach one (or two, if you're using source and destination) to a surface, and then specify the `DDBLT_KEYSRC`, `DDBLT_KEYDEST`, `DDBLTFAST_SRCCOLORKEY`, or `DDBLTFAST_DESTCOLORKEY` flag when blitting, depending on which blit function and what type of color key you're using. Second, you can create a color key and pass it to the blit operation through the `DDBLTFX` structure. The first method is recommended if you're going to be using a color key over and over, whereas the second method is ideal if you only want to use a certain color key once.

You can either attach a color key to a surface that has been created, or you can create the color key at the same time you create the surface. I'll show you how to do both. Let's assume you're working in a 16-bit display mode, with a 565 pixel format, and you want a source color key on your back buffer that includes black only. If your back buffer has already been created, you simply create a DDCOLORKEY structure like we saw before, and pass it to IDirectDrawSurface7::SetColorKey(), shown below:

```
HRESULT SetColorKey(
  DWORD dwFlags,
  LPDDCOLORKEY lpDDColorKey
);
```

Remember to test the return value of this function with the FAILED() macro when you call it, to make sure everything happens the way you want it to. The parameters are straightforward:

DWORD dwFlags: One or more of a series of simple flags describing the color key. There are only three you'll use:

DDCKEY_COLORSPACE	Specifies that the color key describes a range of colors, not just one.
DDCKEY_DESTBLT	Specifies that the color key should be used as a destination color key for blit operations.
DDCKEY_SRCBLT	Specifies that the color key should be used as a source color key for blit operations.

LPDDCOLORKEY lpDDColorKey: This is a pointer to the DDCOLORKEY structure you filled out.

And that's all there is to it. From that point on, you just specify the appropriate flag on blits which you want to use the color key. Note that just because a surface has a color key attached to it, doesn't mean you have to use it every time. If you specify blits with only the DDBLT_WAIT or DDBLTFAST_WAIT flag and nothing else, any color keys will be ignored. So here's how you'd set up the color key we described earlier:

```
DDCOLORKEY ckey;
ckey.dwColorSpaceLowValue = RGB_16BIT565(0, 0, 0);  // or we could just say '0'
ckey.dwColorSpaceHighValue = RGB_16BIT565(0, 0, 0);
if (FAILED(lpddsBack->SetColorKey(DDCKEY_SRCBLT, &ckey)))
{
  // error-handling code here
}
```

If you want to create a surface with the color key already specified, there are just a few things you have to do. First, when you're specifying the valid fields of the DDSURFACEDESC2 structure, you need to include DDSD_CKSRCBLT or DDSD_CKDESTBLT in the dwFlags member, depending on which type of color key you want to use. Looking back at the DDSURFACEDESC2 structure, it contains two DDCOLORKEY structures. One is called ddckCKSrcBlt, and the other is called ddckCKDestBlt. Fill out the appropriate structure, create the surface, and you're all set! Here's the example code for an offscreen surface that's 640x480 in size.

```
// set up surface description structure
INIT_DXSTRUCT(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT | DDSD_CKSRCBLT;
ddsd.dwWidth = 640;                                          // width of the surface
ddsd.dwHeight = 480;                                         // and its height
ddsd.ddckCKSrcBlt.dwColorSpaceLowValue  = RGB_16BIT(0,0,0);  // color key low value
```

```
ddsd.ddckCKSrcBlt.dwColorSpaceHighValue = RGB_16BIT(0,0,0);  // color key high value
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;                // type of surface

// now create the surface
if (FAILED(lpdd7->CreateSurface(&ddsd, &lpddsBack, NULL)))
{
  // error-handling code here
}
```

That wraps up color keys, so now we can cover the final topic of this article, which is clipping.

The IDirectDrawClipper Interface

Suppose you have a graphic that you want to appear such that only half of it is on the screen. How would you do it? If you've programmed games in DOS, you've probably done clipping the hard way. Well, in DirectX, it's trivial! First of all, it's pretty easy to do manually, since DirectX uses `RECT`s for blitting, and changing the coordinates of a `RECT` is much easier than figuring out which parts of the graphic's memory should be copied, like you would have to do in DOS. But second, DirectDraw provides an entire interface to take care of this for you, called `IDirectDrawClipper`.

The clipping capabilities included in DirectDraw are about as flexible as you could ask for. Not only can you clip to any `RECT` area on any surface, but you can clip to multiple areas! That is, if you wanted a main viewing window, a status bar on the side of your screen, and a text area on the lower part of the screen, with a black divider partitioning the screen into those three areas, you could set up a DirectDraw clipper that would clip to all three regions. How cool is that?

There are several steps involved in creating a clipper to do this kind of work for you. The first is to actually retrieve a pointer to the `IDirectDrawClipper` interface. Not surprisingly, this is accomplished with a call to `IDirectDraw7::CreateClipper`, as shown here:

```
HRESULT CreateClipper(
  DWORD dwFlags,
  LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,
  IUnknown FAR *pUnkOuter
);
```

Before you make this call, you'll need to declare a pointer of type `LPDIRECTDRAWCLIPPER`, so you can pass its address to the function. Remember to test for failure, as always. Here are the parameters:

`DWORD dwFlags`: Nice and easy -- this isn't used yet and should be set to 0.

`LPDIRECTDRAWCLIPPER FAR *lplpDDClipper`: Pass the address of your `LPDIRECTDRAWCLIPPER` pointer.

`IUnknown FAR *pUnkOuter`: You know what to do. Just say no... er, `NULL`. :)

Once that's out of the way and you have your interface pointer, the next thing to do is to create a clip list. A clip list is basically a list of the `RECT`s you want to clip to. The structure that's used is called an `RGNDATA`, and it contains enough information to define an arbitrary region, which can be made up of multiple components. Let's look at the structure.

```
typedef struct _RGNDATA {
```

```
    RGNDATAHEADER rdh;
    char          Buffer[1];
  } RGNDATA;
```

It's not at all clear what those parameters are for, so I'll go over them in detail.

`RGNDATAHEADER rdh`: This is a structure nested within the `RGNDATA` that contains information about the second parameter, `Buffer`. Its fields include things like how many areas comprise the region, what shape those areas are, etc. We'll cover that structure in just a second.

`char Buffer[1]`: This isn't actually meant to be a single-valued array; it's going to be an area in memory of arbitrary size that holds the data for the actual clipping areas. As such, instead of declaring an `RGNDATA` structure, what we do is to declare a pointer to the structure, and then use `malloc()` to set enough memory aside for the `RGNDATAHEADER`, and the clip list itself. One thing I'll mention now: `RECT`s in the clip list should be ordered from top to bottom, then from left to right, and must not overlap.

I realize that this is all a little hazy right now, but all will be made clear. To that end, here's the `RGNDATAHEADER` structure, which is relatively easy to understand.

```
typedef struct _RGNDATAHEADER {
  DWORD dwSize;
  DWORD iType;
  DWORD nCount;
  DWORD nRgnSize;
  RECT  rcBound;
} RGNDATAHEADER;
```

`DWORD dwSize`: This is the size of the structure in bytes. Simply set it to `sizeof(RGNDATAHEADER)`.

`DWORD iType`: This represents the shape of each of the areas which make up the region. It is included so that it may be expanded upon later. Right now, the only valid setting for this member is `RDH_RECTANGLES`, which is what we want anyway.

`DWORD nCount`: This is the number of rectangles that make up the region. In other words, it's the number of `RECT`s you plan to use in your clip list.

`DWORD nRgnSize`: Set this to the size of the buffer that will be receiving the region data itself. Since we're using `RECT`s, this size will be `sizeof(RECT) * nCount`.

`RECT rcBound`: This is a `RECT` that bounds all the rectangles in your clip list. Usually you'll set this to the dimensions of the surface on which the clipping will occur.

Now that we've seen all of the structures involved, we can generate a clip list. First we declare an `LPRGNDATA` pointer and allocate enough memory to it to hold our clip list, then simply fill out the fields of each structure according to their descriptions above. Let's look at the simplest case, which you'll probably use often, which is that of only a single clipping area. Furthermore, let's make it size of the whole screen, in a 640x480 display mode. Here's the code that will get the job done.

```
// first set up the pointer -- we allocate enough memory for the RGNDATAHEADER
// along with one RECT. If we were using multiple clipping area, we would have
```

```
// to allocate more memory.
LPRGNDATA lpClipList = (LPRGNDATA)malloc(sizeof(RGNDATAHEADER) + sizeof(RECT));

// this is the RECT we want to clip to: the whole display area
RECT rcClipRect = {0, 0, 640, 480};

// now fill out all the structure fields
memcpy(lpClipList->Buffer, &rcClipRect, sizeof(RECT));  // copy the actual clip region
lpClipList->rdh.dwSize = sizeof(RGNDATAHEADER);          // size of header structure
lpClipList->rdh.iType = RDH_RECTANGLES;                  // type of clip region
lpClipList->rdh.nCount = 1;                              // number of clip regions
lpClipList->rdh.nRgnSize = sizeof(RECT);                 // size of lpClipList->Buffer
lpClipList->rdh.rcBound = rcClipRect;                    // the bounding RECT
```

Once you have a clip list, you need to set it as such with your clipper. The call you want is `SetClipList()`, which is of course a method of the `IDirectDrawClipper` interface. Here's what it looks like:

```
HRESULT SetClipList(
  LPRGNDATA lpClipList,
  DWORD dwFlags
);
```

All you need to do is pass a pointer to the `RGNDATA` structure you just filled out. The `dwFlags` parameter is not used, so just set it to 0. Now that the clip list is set, there's just one more step, and that's attaching the clipper to the surface you want to do your clipping on. This requires a call to `SetClipper()`, which is a method of the surface you want to attach the clipper to, not the clipper itself.

```
HRESULT SetClipper(LPDIRECTDRAWCLIPPER lpDDClipper);
```

And you know what to do with that: just pass your interface pointer and you're all set. Anytime you try to blit to a surface that has a clipper attached to it, the clipper does all the work. So if you want to show a tile that's half-on, half-off the screen, go ahead and blit to a `RECT` like {-10, -10, 6, 6}, or whatever it happens to be. Pretty sweet, hey?

The last thing I'll say about clippers is that you should remember to `free()` the memory that you allocated with `malloc()`, no matter what happens with your clipper. That is, if the call to `SetClipList()` or `SetClipper()` fail for some reason, make sure you're still freeing the memory before you return an error code or whatever you do to handle errors. You won't be needing the `LPRGNDATA` pointer anymore after you use it to set the clip list, so its memory should be deallocated immediately.

Closing

That just about wraps up the series on general DirectDraw stuff! Can you believe how much we've covered in just six articles? Congratulate yourself if you're still reading these; you've come a long way. :) To better illustrate some of the things we've covered today, I've thrown a little demo program together for you. It demonstrates loading a bitmap resource; using the blitter for image copying, color filling, and scaling; and using a clipper to make it all easy. The program is available here.

There are still some things we haven't covered that didn't quite fit into the articles like I wanted them to, like page flipping as an alternative to double-buffering, and using DirectDraw in a windowed app, but that's all right, because we'll just pick them up as we go on.

Now that the initiation is over, I'm going to be shifting the focus of these articles off of general Windows programming, and onto developing a tile-based RPG. Future articles will include such things as developing a good input mechanism with DirectInput, writing a basic scripting engine, playing background sound and music, developing utilities to help you with game design, etc. Next time, we'll take a look at developing a simple scrolling engine for a tile-based game. It's easier than you might think!

As always, until then, feel free to send me your questions at ironblayde@aeon-software.com, or reach me on ICQ at UIN #53210499. Practice up on the techniques we've covered so far, because you're going to need them all. :) Later, everyone!

Copyright © 2000 by Joseph D. Farrell. All rights reserved.

Discuss this article in the forums

© 1999-2001 Gamedev.net. All rights reserved. Terms of Use Privacy Policy
Comments? Questions? Feedback? Send us an e-mail!