DirectDraw Programming Tutorial

by Lan Mader

Contents

Overview of DirectDraw

DirectDraw Programming

A Few Details

Overview of DirectDraw

This section will give a high level overview of DirectDraw, explaining the concepts needed to understand what DirectDraw achieves.

1. DirectX API groups

Microsoft's DirectX APIs are made up of the following groups:

- DirectDraw - Direct access to video memory
- DirectSound - Direct access to Sound hardware
- DirectPlay - Support for networked multiplayer gaming
- DirectInput - Support for gaming input devices such as joysticks

They are all designed to give the programmer APIs for directly accessing hardware. This paper will focus on techniques for using DirectDraw functions so you can create fast, state of the art programs.

2. What is DirectDraw

DirectDraw is essentially a video memory manager. Its most important capability is to allow the programmer to store and manipulate bitmaps directly in video memory. It lets you take advantage of the video hardware's blitter to blit these bitmaps within video memory. Being able to blit from video memory to video memory using the video hardware's blitter is much faster than blitting from system memory to video memory. This is especially true with today's 64 bit video cards that provide a 64 bit data path within video memory. Also, the hardware blit operates independently of the CPU and therefore frees up the CPU to continue working. Additionally, DirectDraw supports other hardware acceleration features of the video card, such as hardware support for sprites and z-buffering. DirectDraw 1.0 is currently available for Windows 95 and will be available for Windows NT this summer.

3. Relationship to WinG

Up until now programmers have used the WinG or CreateDIBSection technology to do fast animation in Windows. This approach gave the programmer direct access to the bitmap in system memory so that one can use optimized routines for drawing to the bitmap. WinG is better than using GDI for all operations that draw to the bitmap, because GDI could never be as fast as the optimized routines used by game programmers.

Once the WinG scene is composed on the bitmap, it is blitted from system memory to video memory, which causes it to be displayed. This technique is not as fast as Direct Draw because blitting from system memory to video memory is slower than blitting from video memory to video memory. Both the DirectDraw and WinG techniques can and will coexist in many complex games or applications since video memory is a limited resource.

4. DirectDraw surfaces - how you access video memory

In DirectDraw, the goal is to put as many bitmaps in video memory as possible. With DirectDraw, all of video memory is available to you. You can use it both to store various bitmaps, and for the primary and offscreen video buffers. All of these pieces of video memory are referred to as surfaces in Direct Draw. When you load a bitmap that will represent a sprite, and store it in video memory, you first create a surface, which is a piece of video memory, and then blit the bitmap into this surface, thereby effectively copying the bitmap into video memory. This bitmap can now live in video memory for you to use, for as long as you like.

The video memory that is currently displaying something onscreen is also a surface, and is referred to as the primary surface. This surface is as big as the amount of memory needed for the current display mode. If the display mode is 640x480 with 256 colors (8 bits per pixel), then the primary surface uses 307,200 bytes of video memory. You usually also create at least one offscreen surface that is the same size as the primary surface for page flipping. This means that you need 614,400 bytes of display memory just to get started, without even loading any bitmaps.

So, one of the most important hardware requirements for fast DirectDraw games is lots of video memory. When you run out of video memory, your DirectDraw surfaces will get created in system memory, and all the benefits of the hardware blits won't be available to these surfaces. Most video cards these days have at least 1 Meg of video memory, however 1 Meg is barely enough to get going. A 2 Meg video card is probably the practical minimum for getting even a fairly simple app working at its best.

DirectDraw provides functions for determining at run time the amount of video memory available, so the application can optimize its use of video memory.

5. The page flipping approach to animation

A scene is composed by blitting the bitmaps from their video memory buffer (surface) to another offscreen buffer (surface) also in video memory. The scene is then displayed by using the hardware's ability to flip the visible video surface to the offscreen surface so that the newly composed scene is now visible. This is known as page flipping and it is very fast. A page flip is fast because there is no memory copying involved. It is simply a matter of setting a register on the video card that tells it where to scan the video memory that will be used to send the signal to the monitor. DirectDraw shields the programmer from the hardware specifics of such operations, and provides a very simple Flip() API function to do the work. To do animation with page flipping:

1. Compose the scene in the offscreen surface
2. Flip to the offscreen surface to display it.
3. The previously displayed video buffer is now the offscreen buffer
4. Repeat step 1.

Frame rates for page flipping are quite high and are limited by the refresh rate of the monitor, since a page flip will only occur with the vertical retrace signal. This means that if your monitor is set for 72 Hz refresh, you could conceivably achieve frame rates up to 72 frames per second. Waiting for the vertical retrace is nice because it means that you won't see tearing with DirectDraw page flipping. Tearing is the phenomenon witnessed when you move a sprite on screen half way through the refresh of the screen. The sprite appears to tear as it is moved to its new location. For instance, the top half of an image may appear in one location, while the bottom half may be moved slightly to the right or left, which gives the appearance that the image is broken or "torn", right through the middle.

Graphics programmers have developed some sophisticated techniques for optimizing the performance of sprites, and these techniques can still be used in DirectDraw applications. However, learning these techniques won't help one learn or understand DirectDraw. I will, therefore, focus on the page flipping approach to animation. Page flipping is probably the easiest approach to animation, as well as the best one for demonstrating how DirectDraw works.

6. DirectDraw and COM

DirectX is implemented using COM objects. Using COM objects in DLLs has many benefits over simply exporting flat APIs from DLLs, including allowing for polymorphism and versioning. The main thing for the DirectDraw programmer to know is that using the COM objects is no harder than using any other API, especially from a C++ or Object Pascal app.

You don't need to mess with initializing COM, or calling QueryInterface to get the interfaces. The DirectDraw header file declares the C++ classes for the various DirectX objects. Instances of these classes are created for you by calling various create functions. From there its simply a matter of learning the various member functions.

DirectDraw Programming

This section will present code samples to demonstrate how to use DirectDraw. The code samples will be fully functional, however they will be as bare bones as possible to keep the point clear.

1. What you need to compile and link

First and foremost, you will want the DirectX SDK. This is currently available only on MSDN level II or above, and with MS Visual C++ 4.1. The SDK provides a descent help file, and excellent example programs.

To compile and link a DirectDraw application, you need the DDRAW.DLL, DDRAW.H, and you need to create an import library from DDRAW.DLL with the IMPLIB.EXE utility. Since DirectX is only a Win32 technology, you will need a compiler capable of generating Win32 apps. Borland C++ 4.52 and Borland C++ 5.0 provide a great platform for DirectX development.

To run DirectX technology, you must have the DirectX drivers installed on your system. Since this is a commonly used gaming technology, you may find that DirectX is already installed on your system. Look in your Windows/System directory to see if you have a copy of DDRAW.DLL already installed on your system. (Remember, this technology will not work on the 3.X versions of Windows NT.)

2. Initializing DirectDraw

The first thing to learn about DirectDraw is how to initialize it. Read the detailed comments in the following source code for explanations at each step.

```
// globals (ugh)
LPDIRECTDRAW lpDD; // DirectDraw object defined in DDRAW.H

/*
 * Function to initialize DirectDraw
 * Demonstrates:
 *   1) Creating the Direct Draw Object
 *   2) Setting the Cooperative level
 *   3) Setting the Display mode
 *
 */
bool DirectDrawInit(HWND hwnd)
{
    HRESULT ddrval;

   /*
    * Create the main DirectDraw object.
    *
    * This function takes care of initializing COM and Constructing
    * the DirectDraw object.
    */
    ddrval = DirectDrawCreate( NULL, &lpDD, NULL );
    if( ddrval != DD_OK )
    {
        return(false);
    }

   /*
    * The cooperative level determines how much control we have over the
    * screen. This must at least be either DDSCL_EXCLUSIVE or DDSCL_NORMAL
    *
    * DDSCL_EXCLUSIVE allows us to change video modes, and requires
    * the DDSCL_FULLSCREEN flag, which will cause the window to take over
    * the fullscreen. This is the preferred DirectDraw mode because it allows
    * us to have control of the whole screen without regard for GDI.
    *
    * DDSCL_NORMAL is used to allow the DirectDraw app to run windowed.
    */
    ddrval = lpDD->SetCooperativeLevel( hwnd, DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
    if( ddrval != DD_OK )
    {
        lpDD->Release();
        return(false);
    }

   /*
    * Set the video mode to 640x480x8
    * This is allowed because we have set exclusive mode above
    */
    ddrval = lpDD->SetDisplayMode( 640, 480, 8);
    if( ddrval != DD_OK )
    {
        lpDD->Release();
        return(false);
    }

    return(true);
```

```
}
```

We have now initialized DirectDraw and set the display mode.

3. Creating the primary surface and setting up for page flipping

Next we create the primary surface with one back buffer. This is called a complex surface in DirectDraw. The BackBuffer is "attached" to the primary surface. When the cleaning up at the end of the program, you only need to call Release on the primary surface pointer, and the backbuffer is freed for you.

```
// globals (ugh)
LPDIRECTDRAWSURFACE lpDDSPrimary; // DirectDraw primary surface
LPDIRECTDRAWSURFACE lpDDSBack; // DirectDraw back surface

/*
 * Create a Primary surface that is flippable, with an
 * attached backbuffer
 */
bool CreatePrimarySurface()
{
    DDSURFACEDESC ddsd;
    DDSCAPS ddscaps;
    HRESULT ddrval;

    // Create the primary surface with 1 back buffer
    memset( &ddsd, 0, sizeof(ddsd) );
    ddsd.dwSize = sizeof( ddsd );

    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP | DDSCAPS_COMPLEX;
    ddsd.dwBackBufferCount = 1;

    ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
    if( ddrval != DD_OK )
    {
        lpDD->Release();
        return(false);
    }

    // Get the pointer to the back buffer
    ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
    ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps, &lpDDSBack);
    if( ddrval != DD_OK )
    {
        lpDDSPrimary->Release();
        lpDD->Release();
        return(false);
    }

    return true;
}
```

We have now created our double buffered page flipping environment.

4. Loading a bitmap into video memory

The next step is to load the bitmaps for the game. Ideally there is enough video memory so that all the bitmaps live in video memory. If not, CreateSurface will create the surfaces in system memory for you. Everything will still work, the blits from these surfaces will simply be a bit slower.

```
/*
 * Function to create an offscreen surface and load a bitmap from
 * disk into it. The szBitmap field is the filename of the Bitmap.
 */
IDirectDrawSurface * DDLoadBitmap(IDirectDraw *pdd, LPCSTR szBitmap)
{
    HBITMAP hbm;
    BITMAP bm;
    IDirectDrawSurface *pdds;

    // LoadImage has some added functionality in Windows 95 that allows
    // you to load a bitmap from a file on disk.
    hbm = (HBITMAP)LoadImage(NULL, szBitmap, IMAGE_BITMAP, 0, 0,
    LR_LOADFROMFILE|LR_CREATEDIBSECTION);

    if (hbm == NULL)
        return NULL;

    GetObject(hbm, sizeof(bm), &bm); // get size of bitmap

   /*
    * create a DirectDrawSurface for this bitmap
```

```
    * source to function CreateOffScreenSurface() follows immediately
    */

    pdds = CreateOffScreenSurface(pdd, bm.bmWidth, bm.bmHeight);
    if (pdds) {
        DDCopyBitmap(pdds, hbm, bm.bmWidth, bm.bmHeight);
    }

    DeleteObject(hbm);

    return pdds;
}

/*
 * Creates a DirectDraw Surface of a specified size
 * This surface will be in video memory if enough is
 * available, otherwise it will be created in system memory
 */
IDirectDrawSurface * CreateOffScreenSurface(IDirectDraw *pdd, int dx, int dy)
{
    DDSURFACEDESC ddsd;
    IDirectDrawSurface *pdds;

    //
    // create a DirectDrawSurface for this bitmap
    //
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = dx;
    ddsd.dwHeight = dy;

    if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
    {
        return NULL;
    } else {
        return pdds;
    }
}

/*
 * This function copies a previously loaded bitmap into a DirectDraw surface.
 * Notice that we can obtain a GDI style device context for a
 * DirectDraw surface, with which to BitBlt the bitmap into the
 * surface.
 */
HRESULT DDCopyBitmap(IDirectDrawSurface *pdds, HBITMAP hbm, int dx, int dy)
{
    HDC hdcImage;
    HDC hdc;
    HRESULT hr;
    HBITMAP hbmOld;

    //
    // select bitmap into a memoryDC so we can use it.
    //
    hdcImage = CreateCompatibleDC(NULL);
    hbmOld = (HBITMAP)SelectObject(hdcImage, hbm);

    if ((hr = pdds->GetDC(&hdc)) == DD_OK)
    {
        BitBlt(hdc, 0, 0, dx, dy, hdcImage, 0, 0, SRCCOPY);
        pdds->ReleaseDC(hdc);
    }

    SelectObject(hdcImage, hbmOld);
    DeleteDC(hdcImage);

    return hr;
}
```

At this point we have a fully functioning DirectDraw surface with a bitmap loaded into it. We will use this surface later to blit it onto the scene we are composing.

5. Loading the palette

We need to set the palette for the primary surface. This implies that all of the bitmaps that are created for the game should all have been created with the same palette. Here is a function to create a DirectDraw palette for a given bitmap.

```
/*
 * Creates a DirectDraw palette for a given bitmap on disk.
 * The parameter szBitmap is the file name of the bitmap.
```

```
 *
 */
IDirectDrawPalette * DDLoadPalette(IDirectDraw *pdd, LPCSTR szBitmap)
{
    IDirectDrawPalette* ddpal;
    int i;
    int n;
    int fh;
    PALETTEENTRY ape[256];

    if (szBitmap && (fh = _lopen(szBitmap, OF_READ)) != -1)
    {
        BITMAPFILEHEADER bf;
        BITMAPINFOHEADER bi;

        _lread(fh, &bf, sizeof(bf));
        _lread(fh, &bi, sizeof(bi));
        _lread(fh, ape, sizeof(ape));
        _lclose(fh);

        if (bi.biSize != sizeof(BITMAPINFOHEADER))
            n = 0;
        else if (bi.biBitCount > 8)
            n = 0;
        else if (bi.biClrUsed == 0)
            n = 1 << bi.biBitCount;
        else
            n = bi.biClrUsed;

      //
      // a DIB color table has its colors stored BGR not RGB
      // so flip them around.
      //
        for(i=0; i<n; i++ )
        {
            BYTE r = ape[i].peRed;
            ape[i].peRed = ape[i].peBlue;
            ape[i].peBlue = r;
        }
    }
    if (pdd->CreatePalette(DDPCAPS_8BIT, ape, &ddpal, NULL) != DD_OK)
    {
        return NULL;
    } else {
        return ddpal;
    }
}
```

Next we set this palette to the primary surface like this:

```
lpDDPal = DDLoadPalette(lpDD, szBitmap); // Call the function above to load the palette

if (lpDDPal)
    lpDDSPrimary->SetPalette(lpDDPal); // this sets the palette for the primary surface
```

6. Color Keying

DirectDraw uses the concept of color keying to allow for using transparent colors during blits. That is, you can make part of your bitmap appear to be transparent, which is a useful technique to use when blitting sprites onto a background surface. The color key specifies a range of palette entries in the source or the destination bitmap that will not be copied during a blit. Typically you set a the 0th or the 255th palette entry to be the transparent color on the source bitmaps, and also construct our bitmaps such that the areas that need to be transparent use this color of the palette.

```
// Set the color key for this bitmap.
//
// Whatever color is at entry 255 in the palette will be the
// transparent color for blits. This color is black unless
// you used the DDPCAPS_ALLOW256 flag when the palette was
// created

DDCOLORKEY ddck;
ddck.dwColorSpaceLowValue = 0xff;
ddck.dwColorSpaceHighValue = 0xff;

// lpDDSSomeBitmapSurface is a DirectDraw surface with a
// bitmap loaded into it. This needs to be done for each
// surface containing a bitmap.
lpDDSSomeBitmapSurface->SetColorKey( DDCKEY_SRCBLT, &ddck );
```

7. Composing the Scene

At this point we are ready to start creating a game, simulation or other graphics based program. The simple approach is

to blit a background bitmap onto the back buffer, then blit some sprites (also just bitmaps) onto the back buffer at their appropriate locations, and flip. Then repeat the above steps, blitting the sprites into new locations.

```
// This function should be called repeatedly during the idle time of your
// PeekMessage()loop.

void updateFrame( void )
{
    RECT rcRect;
    HRESULT ddrval;
    int xpos, ypos;

    // Blit the stuff for the next frame
    SetRect(&rcRect, 0, 0, 640, 480);

    // blit the background bitmap. This is a 640x480 bitmap,
    // so it will fill the screen (remember, we set the video
    // mode to 640x480x8.
    // The parameter lpDDSOne is a hypothetical surface with this
    // background bitmap loaded. LpDDSBack is our backbuffer.
    ddrval = lpDDSBack->BltFast( 0, 0, lpDDSOne, &rcRect,
    DDBLTFAST_NOCOLORKEY | DDBLTFAST_WAIT);

    if( ddrval != DD_OK )
    {
        return;
    }
    SetRect(&rcRect, 0, 0, 32, 32); // our fictitious sprite bitmap is 32x32 in size

    // xpos and ypos represent some appropriate location for the sprite.
    // lpDDSMySprite is just a surface with the bitmap for our sprite.
    ddrval = lpDDSBack->BltFast( xpos, ypos, lpDDSMySprite,
    &rcRect, DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT );
    if( ddrval != DD_OK )
    {
        return;
    }

    // Flip the surfaces
    ddrval = lpDDSPrimary->Flip( NULL, DDFLIP_WAIT );
} /* updateFrame */
```

A couple of things to note here. In the calls to BltFast(), and Flip(), the last parameter included a DDXX_WAIT flag. This is necessary because the each of these functions is asynchronous. That is, they will return when the operation was successfully started (but not yet finished), or when an error caused the function to not be able to start successfully. The most common error returned is an error indicating that the surface is busy completing a previous operation on the surface. You use the DDXX_WAIT flag to tell DirectDraw to keep trying the operation as long as the surface is busy, or until some other error occurs.

Notice that the first call to BltFast(), the one that is blitting the background, uses the flag DDBLTFAST_NOCOLORKEY. This tells BltFast to ignore the color key, and can improve the performance of the blit on some cards. This means that any transparent areas on this bitmap will be copied and therefore not be transparent, but this is OK since this is the background bitmap and not a sprite like image.

When the Flip() function is called, the surface objects associated with the video memory underneath are swapped. This means that what was once the primary buffer is now the backbuffer and vice versa. The result is that you can just keep using your backbuffer pointer to compose the scenes, and call Flip(), without having to worry about keeping track of where the buffers are.

8. Cleaning up

To cleanup in DirectX, the main thing is to call the Release() member function on any DirectX object you created. You would do something like this:

```
/*
 * Call Release on all objects we created to clean up
 */
void finiObjects( void )
{
    if( lpDD != NULL )
    {
        if( lpDDSPrimary != NULL )
        {
            lpDDSPrimary->Release();
            lpDDSPrimary = NULL;
        }

        if( lpDDSOne != NULL )
        {
            lpDDSOne->Release();
            lpDDSOne = NULL;
```

```
        }

        if( lpDDPal != NULL )
        {
            lpDDPal->Release();
            lpDDPal = NULL;
        }
    }
    lpDD->Release();
    lpDD = NULL;
} /* finiObjects */
```

When you call release on the actual DirectDraw object (the instance of LPDIRECTDRAW) the reference count will go to zero, and the object will be destroyed. When this happens, the original screen mode is restored.

A Few Details

1. Debugging DirectDraw

When a DirectDraw app is running full screen, GDI is not available. This means that you cannot set breakpoints in your app and expect to see the debugger when they are hit. There are some fancy solutions involving debuggers that support dual monitors, or remote debugging, but the simplest in terms of hardware requirements is to simply write your app such that it can run windowed as well as full screen.

2. Running Windowed

When you want to run windowed you must initialize DirectDraw differently. Since you have to coexist with GDI, you are not allowed to change the display mode. The main thing to keep in mind is that when you create your primary surface, you are getting a pointer to the display memory currently visible and in use by GDI. You can, if you want, scribble anywhere you want on the screen. So you have to be careful to keep your drawing inside of your window. You can use a DirectDraw clipper to help with this if you like.

The following function initializes DirectDraw for running in a window, and doesn't use page flipping.

```
bool IsWindowed = false;

// function to initialize DirectDraw in windowed mode
bool DDGameInitWindowed(THIS_ HWND hwnd)
{
    DDSURFACEDESC ddsd;
    DDSCAPS ddscaps;
    HRESULT ddrval;

    IsWindowed = true;

   /*
    * create the main DirectDraw object
    */
    ddrval = DirectDrawCreate( NULL, &lpDD, NULL );
    if( ddrval != DD_OK )
    {
        return(false);
    }

    // using DDSCL_NORMAL means we will coexist with GDI
    ddrval = lpDD->SetCooperativeLevel( hwnd, DDSCL_NORMAL );
    if( ddrval != DD_OK )
    {
        lpDD->Release();
        return(false);
    }

    memset( &ddsd, 0, sizeof(ddsd) );
    ddsd.dwSize = sizeof( ddsd );
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

    // The primary surface is not a page flipping surface this time
    ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
    if( ddrval != DD_OK )
    {
        lpDD->Release();
        return(false);
    }

    // Create a clipper to ensure that our drawing stays inside our window
    ddrval = lpDD->CreateClipper( 0, &lpClipper, NULL );
    if( ddrval != DD_OK )
    {
        lpDDSPrimary->Release();
```

```
        lpDD->Release();
        return(false);
    }

    // setting it to our hwnd gives the clipper the coordinates from our window
    ddrval = lpClipper->SetHWnd( 0, hwnd );
    if( ddrval != DD_OK )
    {
        lpClipper-> Release();
        lpDDSPrimary->Release();
        lpDD->Release();
        return(false);
    }

    // attach the clipper to the primary surface
    ddrval = lpDDSPrimary->SetClipper( lpClipper );
    if( ddrval != DD_OK )
    {
        lpClipper-> Release();
        lpDDSPrimary->Release();
        lpDD->Release();
        return(false);
    }

    memset( &ddsd, 0, sizeof(ddsd) );
    ddsd.dwSize = sizeof( ddsd );
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = 640;
    ddsd.dwHeight = 480;

    // create the backbuffer separately
    ddrval = lpDD->CreateSurface( &ddsd, &lpDDSBack, NULL );
    if( ddrval != DD_OK )
    {
        lpClipper-> Release();
        lpDDSPrimary->Release();
        lpDD->Release();
        return(false);
    }
    return(true);
}
```

The first difference between this code and the previous code for initializing DirectDraw is that you call SetCooperativeLevel() with the flag DDSCL_NORMAL. This means that we are not taking complete control of the display and therefore not removing GDI from the picture. When we create the Primary surface, we don't create a complex surface, because we are not preparing for page flipping.

Next we create a DirectDraw clipper for our window and attach it to the primary surface. The DirectDraw clipper is an object used to maintain a list of rectangles for clipping. By setting the clipper to our window handle, we are implicitly telling the clipper to clip using the client area of the window as the rectangle to clip. We then set this to our primary surface, so that blits to this surface are clipped per the client area of our window.

We create the backbuffer separately, and this time we must specify the size.

Now we are set to go. The last thing we need to do differently is handle the code where we would normally call Flip(), and instead do a blit. You can brew your own Flip that does the right thing like this:

```
bool MyFlip()
{
    HRESULT ddrval;
    RECT rcRectSrc;
    RECT rcRectDest;
    POINT p;

    // if we're windowed do the blit, else just Flip
    if (IsWindowed)
    {
        // first we need to figure out where on the primary surface our window lives
        p.x = 0; p.y = 0;
        ClientToScreen(ddWnd, &p);
        GetClientRect(ddWnd, &rcRectDest);
        OffsetRect(&rcRectDest, p.x, p.y);
        SetRect(&rcRectSrc, 0, 0, 640, 480);
        ddrval = lpDDSPrimary->Blt( &rcRectDest, lpDDSBack, &rcRectSrc, DDBLT_WAIT, NULL);
    } else {
        ddrval = lpDDSPrimary->Flip( NULL, DDFLIP_WAIT);
    }

    return (ddrval == DD_OK);
}
```

3. Losing Surfaces

One very important area of robustness that the code samples above ignored is what happens if the user switches away from a fullscreen DirectDraw app to some other app running in the system. First of all, your code needs to handle the WM_ACTIVATEAPP message and set a flag when your app goes inactive. In the idle section of your PeekMessage loop, check this flag, and don't try to update your display when you're not active.

But the real issue is that once you switch to another app, you have left the DirectDraw environment and allowed GDI to become active. GDI is oblivious to DirectDraw, and will overwrite the areas of display memory in use by your DirectDraw surfaces. This means that your bitmaps get wiped out and will need to be reloaded. Fortunately, when you call a function that operates on a surface, such as Flip() or BltFast(), DirectDraw will return an error of DDERR_SURFACELOST to let you know if the surface was lost. So, your code has to check for this error and reload the bitmaps into the surfaces if this error occurs.

```
ddrval = lpDDSPrimary->Blt( &rcRectDest, lpDDSBack, &rcRectSrc, DDBLT_WAIT, NULL);
if( ddrval == DDERR_SURFACELOST )
{
    ddrval = restoreAll();
}

ddrval = lpDDSPrimary->Flip( NULL, DDFLIP_WAIT);

if( ddrval == DDERR_SURFACELOST )
{
    ddrval = restoreAll();
}
:
:

void restoreAll()
{
    // for each surface your app has created you should do the following:
    ddrval = lpMyDDSurface->Restore(); // this reattaches the video memory to the surface
    if( ddrval == DD_OK )
    {
        lpTempDDS ->DDReLoadBitmap(); // this will be the same as the function above
                                      // that was originally used to load the bitmap
                                      // with the exception that the surface is already
                                      // created and ready to go
    }
}
```

4. Further Reading

The best reading I can recommend is the sample code that comes with the DirectX SDK. These samples are worth looking at as they cover some areas of robustness that this paper ignored. I have yet to find a book on DirectX that exposes anything advanced in the API. MSJ has published a couple of very good introductory DirectDraw and DirectSound articles, but nothing very advanced.

Discuss this article in the forums

Date this article was posted to GameDev.net: **8/7/1999**
(Note that this date does not necessarily correspond to the date the article was written)

See Also:
DirectDraw