
Game Programming with DirectX

Menu:

Disclaimer

*This document and all files and code provided with it are provided "as is" and without any warranty at all, not even the implied warranty of fitness of use for any particular purpose. I am **NOT** responsible for any harm that might come from use or misuse of either this document or any of the files available for download from this site. Neither is there any guarantee that the information in this document is correct. If you do not agree to this, leave this site now.*

Introduction

Target audience: C/C++ Windoze programmers who want to learn to write applications that use Microsoft's confusing API, DirectX. The tutorial uses MFC, but most of the stuff is MFC-independent.

Contributing: If anyone out there is interested in contributing to this tutorial (I would love to make it better but am limited by time/energy constraints) then please let me know (feedback.html.)

Bugs:I have been made aware of a couple of bugs in the code sample. One day far in the future when I find time I will fix them.

Please let me know if you find any mistakes in the article.

1999/04/20: I have added a new not-quite-complete sample, ddsamp (no MFC, phew!). This demonstrates the PutPixel method described in 5.1.2. I made it with Visual C++ 6, but if you have an incompatible compiler you should still be able to just the source files in any project you create.

Some words about DirectX:

The next few paragraphs constitute my commentary and critique of DirectX. I am soap-boxing; so you can skip the rest of this section if you don't want to hear my rants.

I wrote the following paragraph last December 1998.

> I know this tutorial is a bit brief, somewhat inadequate and has a few bugs, but at the moment I've practically stopped doing any programming using DirectX, and have thus also for the moment stopped updating this tutorial. I grew to dislike the DirectX API; it is bulky, it is overcomplicated, it is proprietary, it is poorly designed, and there is no technical reason for the existence of large portions of it, such as Direct3D (a much better job could have been done with the existing industry standard 3D API OpenGL for example.) Besides all this, the chief reason I stopped is that I do not find programming with DirectX fun (your mileage may vary); I got into programming because for me it was fun, and it is no good to me to program if I am not having fun. None of this is to say that you should not learn DirectX. Industrially, its existence is extremely useful, and the amount of driver support from hardware vendors is unmatched by anything else. It is also generally the best (more like only) option if you wish to create cutting edge games for the Windows platform, which is probably why many of you are reading this. The number of great games using DirectX technology is also testimony to what is right about it. As things stand in the industry right now, you can not go wrong by learning DirectX. Don't be discouraged just because I personally do not like it.

Well, for a number of reasons, I am back in a job where I will be doing DirectX programming. I still do not like the API. I want newbie game programmers to realise that DirectX is not an example of a decent API design. Don't get me wrong - what

> DirectX does is good, and there is a definite need that DirectX fills. But its design is awful. The way things are in the computer industry right now, Microsoft does not have to worry about coming up with something decent - people will use whatever they push, and they know it. This lack of competition has a negative effect on the quality of code that is produced. My advice to newbie programmers is that you should learn how to be critical of ANY software design - you should learn how to recognize quality and lack of it. Never just accept that something is good just because it is the current standard. Right now, we are stuck with DirectX, and we will be using it for many years to come; but in the future, developers should stand up against bad design, and demand quality. If you want to examine various alternate designs for gaming API's, see my section on X/Linux programming at xprog.html. None of the available ones are, however, as feature-rich as DirectX.
>
> To be fair to Microsoft, they have made some reasonable attempts to improve DirectX - for example, the addition of DirectPrimitive. However, I don't believe in starting with rubbish and then trying to patch it up as best you can. As they say, "Garbage In, Garbage Out". The baggage of execute buffers will still be there with the release of DirectX 3000.

The tutorial

- Chapter 1 Introduction to DirectX
- Chapter 2 A little background on game programming
- Chapter 3 A simple DirectDraw sample
- Chapter 4 A simple Direct3D sample
- Chapter 5 Miscellaneous. About conv3ds.exe; PutPixel with DirectDraw.
- ddsamp.zip ddsamp, 13K: new simple sample (no MFC)

Download

I am sick of visiting sites that make it as difficult as possible to obtain the information they have for offline viewing. In my country telephone calls are billed on a per-time-unit basis, so I'm sure as hell not going to sit for hours clicking on link after link after link just to obtain one silly article while advertising banners get shoved down my throat. This is obfuscation of information.

So in light of this, you can download a zipped copy of my tutorial for offline viewing. No

banners either.

dj.zip (168K) Entire tutorial zipped, including the sample programs.

Resources

- **The DirectX eXperience** http://www.geocities.com/SiliconValley/Way/3390/.
  Numerous articles collected from various sources.

- **The official DirectX homepage** http://www.microsoft.com/directx/.
  Here you can find info on DirectX and also download the latest SDK and/or end-user release of DirectX. ***Please do not ask me for help on how to install or set up DirectX! I am not Microsoft tech support and do not wish to be.***

- **CDX: Free C++ DirectX class library** http://www.maidex.demon.co.uk/.
  An extensive C++ class library wrapping up DirectX functionality. Free, with source code. Includes some nifty extra features such as an AVI player.

- **ClanLib: Cross-platform graphics library** http://www.clanlib.org/.
  This is a C++ games API that makes writing games that will run on both Linux and Windows quite a bit easier. It uses DirectX on the Windows side. It looks quite easy to use, and may be useful even if you aren't interested in making your software cross-platform.

- **The Delphi Games Creator** http://www.ex.ac.uk/~PMBearne/DGC.html.
  Game programming library based on DirectX, for Delphi. A bit like CDX, only for Delphi instead.

- **DirectX with Borland C++** http://www.geocities.com/SiliconValley/Pines/7268/.
  Information on using DirectX under Borland C++. Interesting piece of information there that Microsoft "forgot" to include some stuff for the Borland version of the dxguid lib. (Borland C++ just happens to compete with MS Visual C++.)

- **DirectX with Visual Basic** http://members.xoom.com/vba51/.
  Some information on using DirectX with VisualBasic.

- **Microsoft DirectX Initialization Library**
  http://www.research.microsoft.com/graphics/directxlib/.
  This potentially useful little library handles some of the horribly nitty gritty details that are characteristic of DirectX applications, such as enumerating DirectDraw devices, screen depths

etc.

FAQ (Frequently Asked Questions)

If you email me a question that has already been asked here, I will ignore it. In fact, if you email me any question I will probably ignore it, because that's just the way I am. I am not some sort of tech-support.

- **I get a linker error saying external symbol `main` not found. What's wrong?**

  It's looking for "main" because it thinks you're trying to write a console mode (DOS-style) (command-line) application. Win32 functions have a `WinMain` instead of a `main`. Create a project with the target platform being a Win32 application.

- **What's my opinion on wrapper class libraries for DirectX?**

  If you want to write your own one, fine, but there is already one called CDX. It's free, with source code. As such, it is something that is developed by the software development community, and for the software development community. Lets not fragment software development any more by developing several thousand different class libraries. Use CDX; if its not quite what you need, then don't make a new one, rather add to CDX.

- **How do I load a bitmap using DirectDraw?**

  That'll be coming "soon".

- **How do I get started in game programming?**

  Let me know when you find out.

- **How do I create .x files?**

  The conv3ds.exe program that comes with the DirectX SDK (SDK\BIN directory) converts 3d studio (.3ds) files to .x files. You can get loads of 3ds files from the Internet; a good starting point is http://www.3dcafe.com/.

- **How do I set a 3D scene's background color?**

  SetSceneBackgroundRGB.

- **Your code fails when I try it in 256-color mode. What gives?**

  I was a tad lazy. I'll correct this soon, although I kind of believe 256-color modes should be killed (and made to suffer).

- **What's this `stdafx.h` I see #included in your source?**

  The file `stdafx.h` is a .h file that is created with each MFC AppWizard project. It should be included in every .cpp file for your project, and heres a handy tip: include it *before* any other includes.

- **How do I make the samples (and/or my own stuff) from the command-line, with NMAKE?**

  Blah blah. Coming soon. In the meantime, heres a hint (for Visual C++ users): Check out the batch file VCVARS32.BAT in your VC\BIN directory.

- **I'm out of questions. Tell me something.**

  Never hard-code an absolute directory/filename. It's annoying to download code with a makefile or project workspace that specifies absolute directories, such that you have to change it yourself to get it to compile.

- **Tell me more.**

  You can really mess up your fingers typing, so *be careful*. Don't "race" when you type; take it easy. Get yourself one of those "natural" keyboards, they're pretty good. If your fingers/wrists start to hurt, stop typing and get to a doctor.

- **Tell me even more.**

  Check out Linux and X-Windows. It's great.

- **The tutorial really sux, especially the code, when are you going to improve it?**

  I'm busy working on a "new improved" version, with a decent sample that will be download'able, and probably even usable in your own applications.

- **Your TraceErrorDD function is kind of MFC-specific. That sucks.**

  Yes it does. That was an initial oversight. I'll change that quite soon.

- **What is this D3DVAL stuff?**

  A D3DVALUE is pretty much the same as a float; in fact, D3DVALUE is defined as a float. However, it is incorrect to use float and D3DVALUE interchangably, as there is no guarantee that your code will still work if a future version of DirectX changes D3DVALUE to, say, a double. Therefore you should use D3DVALUE's throughout. There is a macro (#define) to convert floats and doubles to D3DVALUE's, called D3DVAL. Hence: D3DVAL(3.4);

- **Shouldn't you explain a bit more about creating and releasing DirectX objects, as well as the reference count stuff?**

  Yes.

- **Why do I get some sort of palette error when I try create a device from the surface?**

  You're probably in 256-color mode, in which case you need to attach a palette to the surface. I recommend running Windows in 16-bit or higher color modes instead.

- **What is the difference between Direct3D Retained Mode and Immediate Mode?**

  Immediate mode is a lower-level 3D API, in which you have to manage lists of vertices and faces, generally doing sorting yourself etc. Rendering is done via Immediate Mode "execute buffers" which contain lists of rendering instructions to give to the 3D hardware. Rendering can also be done using any of 17 `DrawPrimitive` functions, which let you draw triangles etc without going through the execute buffers. Retained Mode is built on top of Immediate Mode, and wrap up a lot of functionality in classes to manage a 3D scene, 3D objects etc.

- **Why is the area around the window in the DirectDraw sample a mess?**

  Because I didn't bother to solid-color-blit the back buffer black when I created it. It happened to be black on my machine, so I never noticed the bug. Consider it an "exercise".

- **What books can I recommend?**

  None, since I've never really used any. Sorry.

- **When am I going to finish the tutorial?**

  I'm not sure if it'll ever be finished, as such.

- **When am I going to put actual sample code here?**

Quit bugging me, okay? :)

- **How do I get started in game programming/3d programming/3d graphics?**

  Link coming soon as answer to this question.

- **What compiler can I recommend?**

  Whatever you prefer. I like Visual C++; it has a *very* nice IDE, a *great* debugger, and the compiler's pretty good. DJGPP is an excellent option for the poorer amongst you: it's free.

- **Can I use DJGPP for DirectX?**

  I think so (not sure) but when I get time I will find out and place the relevant info here. DJGPP is a free C/C++ compiler; that way you won't have to go buy an expensive compiler.

- **What are all those strange little letters in front of your variable names, like `bAnimating`?**

  Hungarian Notation. The idea is to try make code more readable by including the type of the variable in the name, such as a b for boolean or a dw for a DWORD. Thus a boolean variable indicating whether or not the program is busy animating would be bAnimating. "m_" in front of a variable means 'member'. It takes only an hour or two at most to get used to, and it really helps make code more readable. I recommend it.

- **Collision detection?**

  There is a rather lengthy document on collision detection available elsewhere. Pretty soon I'll even put the URL here, as soon as I find it.

- **Video/AVI player with DirectX?**

  Ack. Ask me again later. I'm still getting nightmares.

- **So what the hell is a UINT anyway?**

  It's an `unsigned int`.

- **What does the Afx in AfxMessageBox mean?**

  I think it means "application framework" its part of MFC-specific stuff.

- **Isn't it silly that you've named the function that creates both primary and back surfaces `CreatePrimarySurface`?**

  Yes. I'll change that soon, with the new sample that's in development.

- **What's your email address? Can I email you?**

  To be perfectly honest, I'm *not* fond of getting email, especially lots of annoying questions. My RSI is bad enough as it is, without that extra typing work to do. Also, I'm really bad at answering email; I sometimes take up to 6 months to answer email, if I answer it at all. I suck at email.

Todo:Coding style, Overlays, Lock, enumerating drivers, enumerating display modes, Cleaning up, Palette, COM Objects, Handling WM_ACTIVATEAPP, WM_MOVE, WM_PAINT, BltFast, Using GDI on DD surface, Showing frame rate, simple bitmap animation techniques & tile-based game techniques in DD, conv3ds and .x files, camera animation in 3d, 3d math and 3d programming basics, setting the windows size in MFC, meshbuilder stuff; explain ramp/RGB/other drivers, as well as caps, in enough detail; sprites (decals), Thanks to section; other compilers like Borland and Watcom and DJGPP, OpenGL.

- 4 April 1999: Tidied up front page a bit.
- 2 September 1997: Changed error-checking to use the SUCCEEDED and FAILED macros; this is the correct way to check for errors with COM objects.
- 17 June 1997: Fixed error in chapter 4 (Direct3D RM App) in CreateSurface - caps for the offscreen surface should have included DDSCAPS_3DDEVICE. (thanks go to Eetu Mänttäri)
- 8 June 1997: Fixed error in SetMode: hr undeclared. Fixed error that SetMode wasn't called in InitInstance. (thanks go to Rostislav V. Shabalin)
- 11 May 97: Fixed: lpDD vs. pDD problem; useless DDERR macro reference removed; 640x480 "scrolling seasickness" problem fixed Added: Clipper code, restoring lost surfaces, most of Direct3D sample
- 5 May 98: Gosh, has it been that long? Anyway, I'm adding a D3D sample program.

URL of this document: http://www.geocities.com/SoHo/Lofts/2018/djdirectxtut.html

---

[Top] [Home]

David Joffe's Guide to Programming Games with DirectX

Chapter 1: Introduction to DirectX

1.1 A few quick words about these articles

Disclaimer

*This document and all files and code provided with it are provided "as is" and without any warranty at all, not even the implied warranty of fitness of use for any particular purpose. I am **NOT** responsible for any harm that might come from use or misuse of either this document or any of the files available for download from this site. Neither is there any guarantee that the information in this document is correct. If you do not agree to this, leave this site now.*

The samples in this document are developed for Visual C/C++ MFC applications, but most of the material covered is pretty much MFC-independent; so don't worry too much if you don't know MFC.

The top-level page to this tutorial contains a list of Frequently Asked Questions. See this first if you have any problems; your question may be there.

1.2 What is DirectX?

Before the release of Windows 95, most games were released for the DOS platform, usually using something like DOS4GW or some other 32-bit DOS extender to obtain access to 32-bit protected mode. Windows 95, however, seemed to signal the beginning of the end of the DOS prompt. Games developers began to wonder how they were going to write games optimally that would run under Windows 95 - games typically need to run in full-screen mode, and need to get as close as possible to your hardware. Windows 95 seemed to be "getting in the way" of this. DOS had allowed them to program as "close to the metal" as possible, that is, get straight to the hardware, without going through layers of abstraction and encapsulation. In those days, the extra overhead of a generic API would have made games too slow.

So Microsoft's answer to this problem was a Software Development Kit (SDK) called DirectX. DirectX is a horrible, clunky poorly-designed, bloated, ugly, confusing beast (*) of an API (Application Programming Interface) that has driven many a programmer to drink. It was originally purchased from a London company called RenderMorphics, and quietly released more or less as is as DirectX 2. DirectX 3 was probably the first "serious" release by Microsoft, who had now begun to actively push it as the games programming API of the future. Being the biggest software company on the planet, and being the developers of the Operating System that some 90% of desktop users were using, they succeeded. Hardware vendors quickly realised that following the Microsoft lead was the prudent thing to do, and everyone began to produce DirectX drivers for their hardware. In many ways this was a good thing for game developers.

The current version (at time of writing this paragraph) is DirectX 6.1. A lot of improvements have been made to the original DirectX. For example, the documentation, which originally sucked, doesn't suck as much anymore.

One of the main purposes of DirectX is to provide a standard way of accessing many different proprietary hardware devices. For example, Direct3D provides a "standard" programming interface that can be used to access the 3D hardware acceleration features of almost all 3D cards on the market which have Direct3D drivers written for them. In theory this is supposed to make it possible for one application to transparently run as it is supposed to across a wide variety of different hardware configurations. In practice, it usually isn't this simple.

The DirectX API is designed primarily for writing games, but can be used in other types of applications as well. The API at the moment has five main sections:

1.2.1 DirectX Components

DirectDraw	2 dimensional graphics capabilities, surfaces, double buffering, etc
Direct3D	A relatively extensively functional 3D graphics programming API.
DirectSound	Sound; 3D sound
DirectPlay	Simplifies network game development
DirectInput	Handles input from various peripherals

Additionally, DirectX 6(?check this) introduces something called DirectMusic, which is supposed to make it easier for game developers to include music in their games so that the mood of the music changes depending on what type of action is going on in the game.

1.3 DirectX performance and hardware acceleration

Although the performance of Direct3D in software only is not too shabby, it doesn't quite cut it for serious games. DirectX is designed with hardware acceleration in mind. It tries to provide the lowest possible level access to hardware, while still remaining a generic interface. Allowing functions such as 3D triangle drawing to be performed on the graphics card frees the CPU (Central Processing Unit) to do other things. Typical Direct3D hardware accelerators would also have at least 4 or preferably 16 or more Megabytes of onboard RAM to store texture maps (bitmapped images made up of small dots called "pixels"), textures, sprites, overlays and more.

DirectDraw and Direct3D are built as a relatively thin layer above the hardware, using what is called the DirectDraw "hardware abstraction layer" (HAL). For functionality not provided by a certain card, an equivalent software implementation would be provided through the "hardware emulation layer" (HEL).

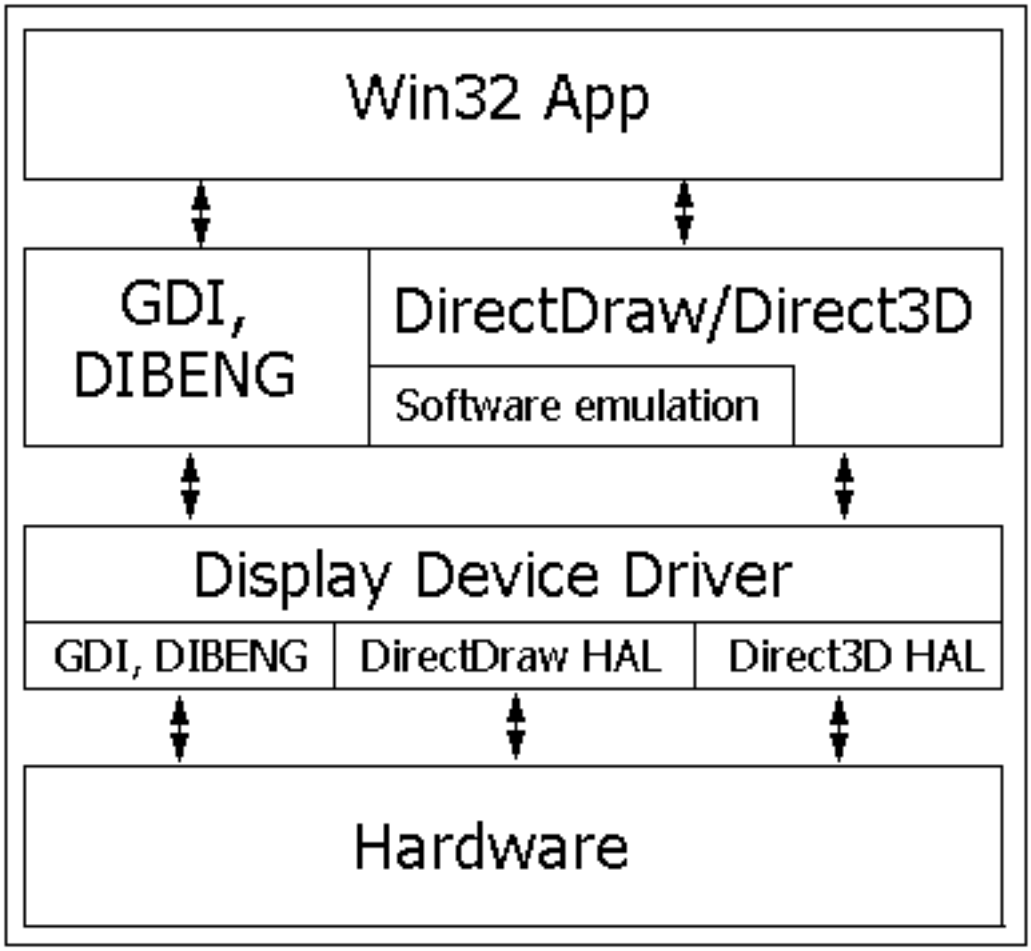


Diagram illustrating where the DirectDraw/Direct3D architecture fits in, hopefully reasonably accurately.

1.4 DirectX and COM

The set of DirectX modules are built as COM (Component Object Model) objects. COM is yet another ugly broken interface from Microsoft - although newer versions of COM don't suck as much as the earlier incarnations. Don't get me wrong, I'm not against the existence of something that does what COM does - but the implementation leaves much to be desired. Anyway, a COM object is a bit like a C++ class, in that it encapsulates a set of methods and attributes in a single module, and in that it provides a kludgy sort of inheritance model, whereby one COM object can be built to support all the methods of it's parent object, and then add some more.

You don't need to know much about COM to use DirectX, so don't worry too much about it. You do a little bit of COM stuff when initializing objects and cleaning them up, and when checking return values of function calls, but that's more or less it.

(*) My own personal opinion. Yours may differ.

Article by David Joffe (http://www.geocities.com/SoHo/Lofts/2018/); Last updated: 17 April 1999

David Joffe's Guide to Programming Games with DirectX

Chapter 2: Palettes, Gaming concepts, double buffering etc

2.1 Video Modes

Screen modes come in several flavours, based on how many bits are used to store the color of each pixel on the screen. Naturally, the more bits you use per pixel, the more colours you can display at once; but there is more data to move into graphics memory to update the screen.

- 1,2,4 and 8 bit "indexed" modes (8 bit is the most popular and is better known as "256-color mode").
- 16-bit (64K colors) "high-color" modes
- 24-bit (16.7M colors) "true-color" modes
- 32-bit RGBA modes. The first 3 bytes are used the same as in 24-bit modes; the A byte is for an "alpha-channel", which provides information about the opacity (transparency) of the pixel.

These modes are available, typically, in the following resolutions:

- 320x200
- 320x240
- 640x400
- 640x480
- 800x600
- 1024x768
- 1280x1024
- 1600x1200 (drool)

with 640x480 being probably the most common mode for running games in at the moment.

Monitor's generally have a width that is 4/3 times their height (called the **aspect ratio**); so with modes where the number of pixels along the width is 4/3 times the number of pixels along the height, the pixels will have an aspect ratio of 1, and thus be physically square. That is to say, 100 pixels in one direction should then be the same physical length as 100 pixels in a perpendicular direction. Note that 320/200 does not have this property; so in 320x200 pixels are actually stretched to be taller than they are wide.

2.2 Color theory

There are a number of different ways that colors can be represented, known as "color models". The most common one is probably RGB (Red,Green,Blue). Nearly all possible visible colors can be produced by combining, in various proportions, the three primary colors red, green and blue. These are commonly stored as

three bytes - each byte represents the relative intensity of each primary color as a value from 0 to 255 inclusive. Pure bright red, for example, would be RGB(255,0,0). Purple would be RGB(255,0,255), grey would be RGB(150,150,150), and so on.

Here is an example of some C code that you might use for representing RGB colors.

```
struct SColor
{
    int r;
    int g;
    int b;
};

SColor make_rgb( int r, int g, int b )
{
    SColor ret;
    ret.r = r;
    ret.g = g;
    ret.b = b;
    return ret;
}
```

This is quite ugly. Alternatively you may want to store an RGB color in an unsigned integer:

```
typedef unsigned int rgb_color;

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

Anyway, I'm rambling now.

There are other color models, such as HSV (Hue, Saturation, Luminance), but I won't be going into them here. The book "Computer Graphics, principles and practise" by Foley & van Dam (often referred to as The Computer Graphics Bible) explains color modes in some detail, and how to convert between color modes.

2.2.1 High-color and true-color modes

In high-color and true-color modes, the pixels on the screen are stored in video memory as their corresponding RGB make-up values. For example, if the top left pixel on the screen was green, then (in true-color mode) the first three bytes in video memory would be 0, 255 and 0.

In high-color modes the RGB values are specified using (if I remember correctly) 5, 6 and 5 bits for red, green and blue respectively, so in the above example the first two bytes in video memory would be, in binary: **00000111 11100000.**

2.2.2 Palette-based, or "indexed" modes

Urgh.

The most common indexed mode is 8-bit, better known as 256-color mode. In this mode, the programmer has a choice of $2^8 = 256$ different colors that can be displayed on the screen at once. The RGB make-up values for each of these 256 colors are stored in a table of RGB records, each 3 bytes long. Thus, each pixel takes one byte in video memory, and the value of that byte specifies an index into the table (called the "palette") which is used to look up the RGB values to generate the pixel on the screen.

Creating an application around this palette is a pain. But using a palette offers several advantages, which include:

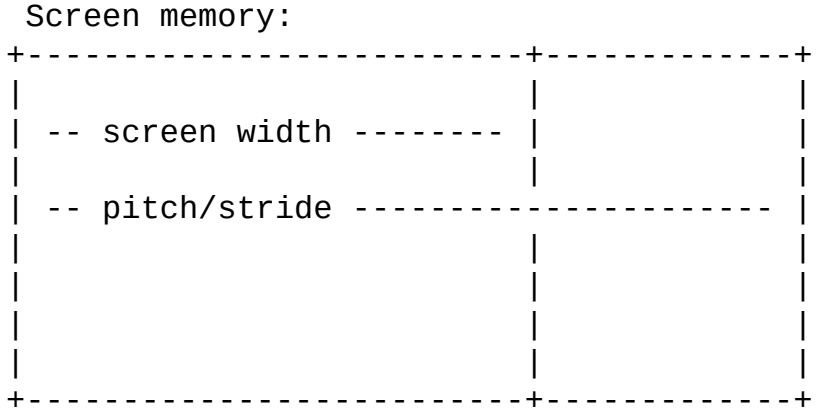
- the contents of the screen can be stored using less video memory
- routines that draw into video memory, and thusly onto the screen, can be made faster, since there are fewer bytes to transfer
- various other interesting tricks; for example, the palette can be "animated" - that is, the RGB values in the look-up table are changed without changing the bytes in video memory. This will change the colors on the screen. One possible effect for this technique might be a fade-out, where the colors in the look-up table are faded to black.

2.2.3 ModeX

ModeX is a special type of mode in which the contents of graphics memory (i.e. what appears on the screen) is stored in a somewhat complex planar format. DirectDraw knows how to write to ModeX surfaces, but the Windows GDI doesn't, so be careful when trying to mix GDI and DirectDraw ModeX surfaces. When setting the DirectDraw fullscreen mode, it is possible to choose whether or not DirectDraw is allowed to create ModeX surfaces. These days you want to avoid ModeX.

2.2.4 Pitch

Even though the screen resolution might be, say, 640x480x8, this does not mean that each row of pixels take up 640 bytes in memory. To optimize the code that goes into a graphics card, video modes are often stored in memory much wider than the screen resolution. A 640x480 mode, for example, might be stored in memory as 1024x480 - then only the leftmost 640 pixels of each row are mapped to the screen. The "real" width that a mode is taking up is known as the **pitch** or **stride** of the surface, and must be obtained from DirectDraw. This is important to know when you write to the memory representing a surface.



2.3 A few gaming concepts you'll need to know to write games

2.3.1 Bitmaps and sprites

A **bitmap** is an image on the computer that is stored as an array of pixel values. That's a pretty crappy description. Basically, a bitmap is any picture on the computer, normally a rectangular block of 'pixels'. A **sprite** is the same thing as a bitmap, except normally it refers to a bitmap that has transparent areas. Sprites are an extremely important component of games. They have a million and one uses. For example, your mouse cursor qualifies as a sprite. The monsters in DOOM are also sprites. They are flat images with transparent areas that are programmed to always face you. Note that the *sprite* always faces you - this doesn't mean the *monster* is facing you. Anyway, enough said about bitmaps and sprites, I think.

2.3.2 Double buffering and page flipping

Double-buffering is a rather nifty technique that is very important in achieving fast, smooth animation in your application. It's not the only thing you'll need to achieve this, but it's definitely one of the more important ones. Basically, it works like this. Your game draws what the user is meant to see on the screen onto an **off-screen** surface. The upshot of this is that the user doesn't have to *watch* the elements of the screen being drawn, as this causes all sorts of icky artefacts, like flickering. Now, only when the image is ready for viewing (or "finished rendering", if you want to speak a bit more technically) is it shown to the user. There are generally two ways of accomplishing this:

- Double-buffering: The off-screen surface is simply copied over onto the main screen surface that is visible. With this setup, the off-screen surface can reside in system memory or in video memory.
- Page-flipping: With this technique, the "off-screen" surface must be in video memory. The graphics hardware is then instructed to use the off-screen surface memory as the memory to update the screen. Now the are in memory that was being shown on the screen becomes the "off-screen" surface. Each alternate frame is drawn to the other surface.

A problem that can arise from this technique is "tearing". Your monitor redraws the image on the screen fairly frequently, normally at around 70 times per second (or 70 Hertz). It normally draws from top to bottom. Now, it can happen that the screen has only drawn half of its image, when you decide to instruct it to start drawing something else, using any one of the two techniques described above. When you do this, the bottom half of the screen is drawn using the new image, while the top half still had the old image. The visual effect this produces is called tearing, or shearing. A solution exists, however. It is possible to time your page flipping to co-incide with the end of a screen refresh. I'll stop here though, having let you know that it is possible. (fixme: i think DirectDraw handles this for you, check this)

2.4 Clipping and DirectDraw clippers

Clipping is the name given to the technique of preventing drawing routines from drawing off the edge of the screen or other rectangular bounding area such as a window. If not performed, the general result could best be described as a mess. In DirectDraw, for example, when using windowed mode; Windows basically gives DirectDraw the right to draw anywhere on the screen that it wants to. However, a well-behaved DirectDraw application would normally only draw into it's own window. DirectX has an object called a "clipper" that can be attached to a DirectDraw surface to prevent it drawing outside of the window.

2.5 DirectDraw surfaces

DirectDraw uses "surfaces" to access any section of memory, either video memory or system memory, that is used to store (normally) bitmaps, texture maps, sprites, and the current contents of the screen or a window.

DirectDraw also provides support for "overlays"; a special type of sprite. An overlay is normally a surface containing a bitmap with transparent sections that will be "overlaid" on the entire screen. For example, a racing car game might use an overlay for the image of the cockpit controls and window frame.

The memory a DirectDraw surface uses can be lost in some circumstances, because DirectDraw has to share resources with the GDI. It is necessary for you application to check regularly that this hasn't happened, and to restore the surfaces if it has.

2.6 DirectX return values and error-checking

All DirectX functions return an HRESULT as an error-code. Since DirectX objects are based on the COM architecture, the correct way to check if a DirectX function has failed is to use the macros **SUCCEEDED()** and **FAILED()**, with the HRESULT as the parameter. It is not merely sufficient to check if, for example, your DirectDraw HRESULT is equal to DD_OK, since it is possible for COM objects to have multiple return values as success values. Your code will probably still work, but technically it is the wrong thing to do.

2.6 DirectX debugging

When you install the DirectX SDK you get a choice of whether to install the retail version of the libraries, or the debug version. The debug version will actually write diagnostic *OutputDebugString* messages to your debugger. This can be very useful. However, it slows things down a LOT - if you have anything less than a Pentium 166, rather choose the release libraries. Also, if you want to mainly play DirectX games, install the retail version. If you want to do mainly DirectX development, and your computer is quite fast, install the debug version. If you want to do both, then you should probably use the release libraries, unless you have a very fast computer that can handle it.

Article by David Joffe (http://www.geocities.com/SoHo/Lofts/2018/); Last updated: 17 April 1999

David Joffe's Guide to Programming Games with DirectX

Chapter 3: A simple DirectDraw sample

3.1 Setting up DirectX under Visual C/C++

I most likely won't be doing DirectX development under Watcom or Borland C/C++ or Delphi or VisualBASIC etc; so if you want such info included, please send as much info as you can on generally getting DirectX programs to compile in these environments + troubleshooting etc.

Firstly, the directories must be set up so that Visual C/C++ can find the DirectX include files and libraries.

Access the **Tools/Options/Directories** tabbed dialog.

Select "library directories" from the drop-down list, and add the directory of the DX SDK libraries, e.g. "d:\dxsdk\sdk\lib"

Select "include directories" from the drop-down list, and add the directory of the DX SDK header files, e.g. "d:\dxsdk\sdk\inc".

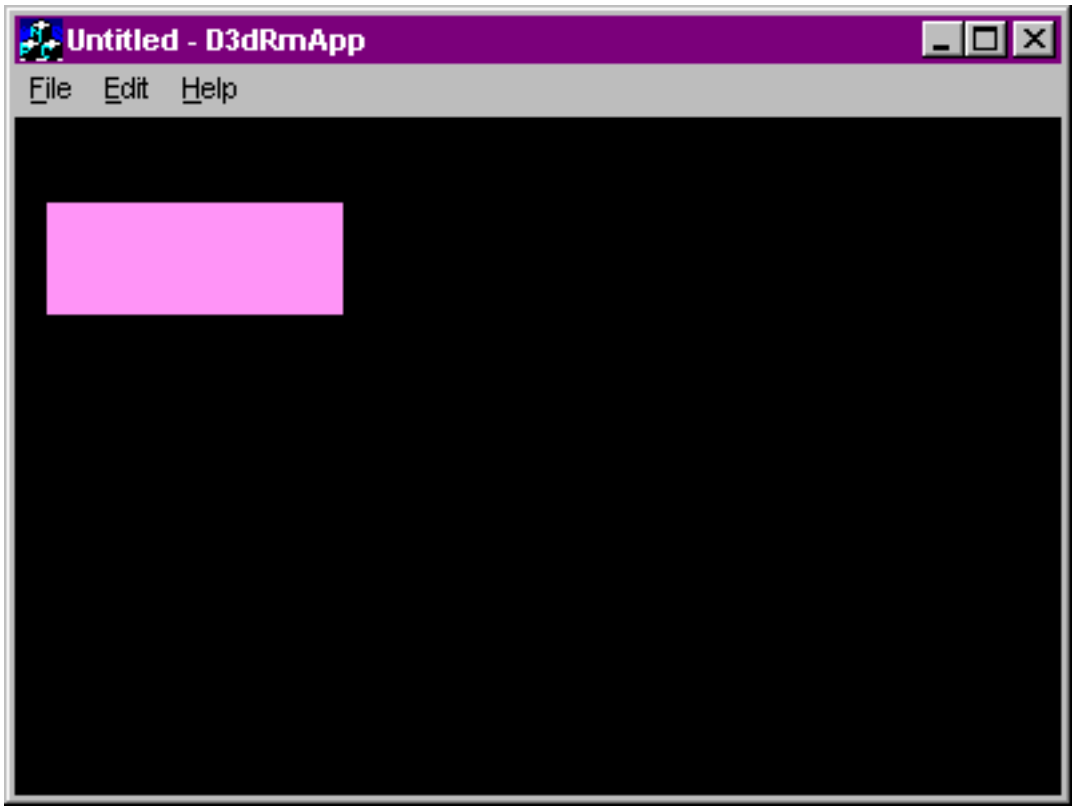
If you are going to be using some of the DX utility headers used in the samples, then also add the samples\misc directory, e.g. "d:\dxsdk\sdk\samples\misc".

Note: In Visual C/C++ 4.2, the header file directory for the DirectX SDK must be located *before* the default VC++ development include directories, so that the compiler searches the DXSDK directories first. As far as I can tell, the reason for this is that with VC++ 4.2 some of the DirectX was included, but those header files seem to be out of date or something because they don't work.

To use DirectDraw you must tell it which library file the DirectDraw routines are in. Ditto for Direct3D. Go to **Build/Settings/Link** and in the *Object/Library modules* box add **ddraw.lib** to use DirectDraw, and **d3drm.lib** if you want to use Direct3D Retained mode. Separate these with spaces.

3.2 The DirectDraw sample

Firstly, heres a screenshot of the small ***simple*** sample application we're putting together here.



The general routine for starting a DirectDraw application is the following:

1. Set up our global variables
2. Initialize a DirectDraw object
3. Set the "cooperative level" and display modes as necessary (explained later)
4. Create front and back flipping surfaces, and a clipper if necessary
5. Attach the clipper if you're in windowed mode
6. Perform flipping. If in full-screen mode, just flip. If in windowed mode, you need to blit from the back surface to the primary surface each frame.

3.3 Setting up global variables: Globals.h and Globals.cpp

We are going to need a number of global variables for our DirectDraw application (these dont strictly need to be

global variables, you may want to encapsulate them in a class for example. This was just a design decision I made, for simplicity). Firstly, we need a variable for accessing the DirectDraw object. Also, we'll need a boolean variable to track whether or not we are in full-screen mode, called **bFullScreen**. Then we need two DirectDrawSurface variables for the primary surface and the back surface which form our double-buffered surface-flipping structure. For windowed mode, we'll need a **clipper** object to attach to the primary surface to prevent DirectDraw from drawing outside of the window's edges. If using a 256 color mode, we'll probably want a structure to store the palette we are going to use. Then, we need an **HWND** object to let the clipper know which window's boundaries it should clip to. Finally, we'll need a boolean variable to start and stop animation of the DirectDraw application, once it is initialized. Create a global variable for purposes of accessing the DirectDraw object, and a boolean variable to track whether or not we are in full-screen mode.

These global variables, along with the global functions, I place in a separate file called **Globals.cpp** which I add to my project in Visual C/C++. They could all have been members of a class, but I have chosen to make them global for easier access across all modules, to conceptually separate the DX stuff from the rest of the program code, and to make it simpler to use these functions in non-MFC applications.

```
LPDIRECTDRAW pDD;                // DirectDraw object
LPDIRECTDRAWSURFACE pDDSPrimary; // DirectDraw primary surface
LPDIRECTDRAWSURFACE pDDSBack;    // DirectDraw back surface
LPDIRECTDRAWPALETTE pDDPal;      // Palette (coming sometime soonish)
LPDIRECTDRAWCLIPPER pClipper;    // Clipper for windowed mode
HWND ddWnd;                      // Handle of window
BOOL bFullScreen;                // are we in fullscreen mode?
BOOL bAnimating;                 // are we animating?
```

I also make a function to initialize all of these variables:

```
void InitDirectXGlobals()
{
    pDD = NULL;
    pDDSPrimary = NULL;
    pDDSBack = NULL;
    pDDPal = NULL;
    pClipper = NULL;
    bFullScreen = FALSE;
    bAnimating = FALSE;
}
```

Here is the general layout of my **Globals.h** and **Globals.cpp** files:

Globals.h

```
#ifndef _GLOBALS_H_
#define _GLOBALS_H_

include <d3drm.h>

extern LPDIRECTDRAW pDD;
```

```
...

extern void InitDirectXGlobals();
...

#endif
```

Globals.cpp

```
#include "stdafx.h"

#include <ddraw.h>
#include <d3drm.h>

#include "Globals.h"

LPDIRECTDRAW pDD;
...

void InitDirectXGlobals()
{
        pDD = NULL;
...
}
```

3.4 Initializing the DirectDraw system

Now we need a routine to initialize the directdraw system. The DirectDraw system must be initialized every time you switch between full-screen and windowed mode (I think :) - gimme a bit of time). The **DirectDrawCreate** function call is used to create a DirectDraw object.

```
BOOL InitDDraw()
{
    HRESULT hr;

    // Create the DirectDraw object
    // The first NULL means use the active display driver
    // The last parameter must be NULL
    hr = DirectDrawCreate(NULL, &pDD, NULL);
        if (FAILED(hr)) {
        TRACE("Unable to create DDraw object\n");
        return FALSE;
    }

    // The DirectDraw object initialized successfully
    return TRUE;
}
```

3.5 Setting the screen mode

The function **SetCooperativeLevel** is used to tell the system whether or not we want to use full-screen mode or windowed. In full-screen mode, we have to get exclusive access to the DirectDraw device, and then set the display mode. For windowed mode, we set the cooperative level to normal.

```
BOOL SetMode() {
    HRESULT hr;
    if (bFullScreen) {
        // Set the "cooperative level" so we can use full-screen mode
        hr = pDD->SetCooperativeLevel(AfxGetMainWnd()->GetSafeHwnd(),
            DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN | DDSCL_NOWINDOWCHANGES);
        if (FAILED(hr)) {
            pDD->Release();
            return FALSE;
        }
        // Set 640x480x256 full-screen mode
        hr = pDD->SetDisplayMode(640, 480, 8);
        if (FAILED(hr)) {
            TRACE("Error setting display mode: %d\n", int(LOWORD(hr)));
            pDD->Release();
            return FALSE;
        }
    } else {
        // Set DDSCL_NORMAL to use windowed mode
        hr = pDD->SetCooperativeLevel(AfxGetMainWnd()->GetSafeHwnd(),
            DDSCL_NORMAL);
    }

    // Success
    return TRUE;
}
```

3.6 Creating surfaces

OK ... now that we've got that bit of initialization out of the way, we need to create a flipping structure. No, I'm not cursing the structure .. "flipping" as in screen page-flipping :).

Anyway, we need to create one main surface that everyone will see, and a "back" surface. All drawing is done to the back surface. When we are finished drawing we need to make what we've drawn visible. In full-screen mode, we just need to call a routine called **Flip**, which will turn the current back surface into the primary surface and vice versa. In windowed mode, we don't actually flip the surfaces - we copy the contents of the back buffer onto the primary buffer, which is what's inside the window. In other words, we **"blit"** the back surface onto the primary surface.

Anyway, here is the bit of code to create the surfaces. Right now the code is ignoring full-screen mode and only catering for windowed mode, but that'll change. Also, if there are errors in this code, consider them "exercises" ... :). Naah, just kidding. If there are errors, mail me.

```
UINT CreatePrimarySurface()
```

```
{
    DDSURFACEDESC ddsd; // A structure to describe the surface we want
    HRESULT hr;         // Holds return values for function calls

    // Screw the full-screen mode (for now)

    // This clears all fields of the DDSURFACEDESC structure to 0
    memset(&ddsd, 0, sizeof(ddsd));
    // The first parameter of the structure must contain the
    // size of the structure.
    ddsd.dwSize = sizeof(ddsd);

    // The dwFlags paramater tell DirectDraw which DDSURFACEDESC
    // fields will contain valid values
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

    // Create the primary surface
    hr = pDD->CreateSurface(&ddsd, &pDDSPrimary, NULL);
    if (FAILED(hr))
    {
        TRACE("Error: pDD->CreateSurface (primary)\n");
        TraceErrorDD(hr);
        pDD->Release();
        pDD = NULL;
        return 1;
    }

    // Now to create the back buffer
    ddsd.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
    // Make our off-screen surface 320x240
    ddsd.dwWidth = 320;
    ddsd.dwHeight = 240;

    // Ignore this: I'm trying stuff still
    /*memset(&ddsd.ddpfPixelFormat, 0, sizeof(ddsd.ddpfPixelFormat));
        ddsd.ddpfPixelFormat.dwSize = sizeof(ddsd.ddpfPixelFormat);
        ddsd.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8;
//      ddsd.ddpfPixelFormat.dwRGBBitCount = 8;*/

    // Create an offscreen surface
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;

    hr = pDD->CreateSurface(&ddsd, &pDDSBack, NULL);
    if (FAILED(hr)) {
        TRACE("Error: CreateSurface (back)\n");
        TraceErrorDD(hr);
        return 2;
    }

    // success
    return 0;
```

```
}
```

3.7 Creating the Clipper

Now that we've created the surfaces, we need to create a clipper (if we're running in windowed mode), and attach the clipper to the primary surface. This prevents DirectDraw from drawing outside the windows client area.

```
UINT CreateClipper()
{
    HRESULT hr;

    // Create the clipper using the DirectDraw object
    hr = pDD->CreateClipper(0, &pClipper, NULL);
    if (FAILED(hr)) {
        TRACE("Error: CreateClipper\n");
        TraceErrorDD(hr);
        return 1;
    }

    // Assign your window's HWND to the clipper
    ddWnd = AfxGetMainWnd()->GetSafeHwnd();
    hr = pClipper->SetHWnd(0, ddWnd);
    if (FAILED(hr)) {
        TRACE("Error: SetHWnd\n");
        TraceErrorDD(hr);
        return 2;
    }

    // Attach the clipper to the primary surface
    hr = pDDSPrimary->SetClipper(pClipper);
    if (FAILED(hr)) {
        TRACE("Error: SetClipper\n");
        TraceErrorDD(hr);
        return 3;
    }
    // success
    return 0;
}
```

3.8 Putting it all together

Now that we have all these initialization routines, we need to actually call them, so the question is, where to call them? In an MFC application, a logical place to do this is in the application's **InitInstance** routine.

```
BOOL CDirectDrawApp::InitInstance()
{
#ifdef _AFXDLL
    Enable3dControls();
```

```
#else
    Enable3dControlsStatic();
#endif

    LoadStdProfileSettings(0);

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CDirectDrawDoc),
        RUNTIME_CLASS(CMainFrame),
        RUNTIME_CLASS(CDirectDrawView));
    AddDocTemplate(pDocTemplate);

    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    InitDirectXGlobals();
    InitDDraw();
    SetMode();
// These will be coming soonish
//    TRACE("Calling LoadJascPalette\n");
//    LoadJascPalette("inspect.pal", 10, 240);
    TRACE("Calling CreatePrimarySurface\n");
    CreatePrimarySurface();
    TRACE("Calling CreateClipper\n");
    CreateClipper();
// Also coming soonish
//    TRACE("Calling AttachPalette\n");
//    AttachPalette(pDDPal);

    // Start the animation
    bAnimating = TRUE;

    return TRUE;
}
```

3.9 Restoring lost surfaces

As if all this initialization wasn't enough, we also have to make sure our DirectDraw surfaces are not getting "lost". The memory associated with DirectDraw surfaces can be released under certain circumstances, because it has to share resources with the Windows GDI. So each time we render, we first have to check if our surfaces have been lost and **Restore** them if they have. This is accomplished with the **IsLost** function.

```
// Checks if the memory associated with surfaces
// is lost and restores if necessary.
// Not sure about fullscreen/windowed;
// I'll get back to you people on this :)
```

```
BOOL CheckSurfaces()
{
    // Check the primary surface
    if (pDDSPrimary) {
        if (pDDSPrimary->IsLost() == DDERR_SURFACELOST) {
            pDDSPrimary->Restore();
            return FALSE;
        }
    }
    return TRUE;
}
```

3.10 The rendering loop

Now that we've got most of the general initialization out of the way, we need to set up a rendering loop. This is basically the main loop of the game, the so-called **HeartBeat** function. So we're going to call it just that.

The **HeartBeat** function gets called during your applications idle-time processing; so we need to override the application's **OnIdle** function. Use ClassWizard or the toolbar wizard thingy to create a handler for idle-time processing for your main application class. Here I am assuming you know a bit about MFC. Sorry about that; that'll probably change though with time as my knowledge improves and as I have more time to work on this.

Note that your applications **OnIdle** handler gets called even when your program does not have the current focus, and we'd rather not have your application do anything unless it has the current focus. Trust me on this one :). This is what we use **bAnimating** for; we set it to **FALSE** when we receive the **WM_ACTIVATE(APP?)** message with a parameter of FALSE. We set this variable to TRUE just after we've set up DirectDraw and the surfaces etc., and we are ready to begin.

```
BOOL CDirectDrawApp::OnIdle(LONG lCount)
{
    CWinApp::OnIdle(lCount); // Call the previous default handler

    if (bAnimating) {
        // Our game's heartbeat function
        HeartBeat();
        // prevent's too much flicker. We'll need a smarter system
        // soon, this one is temporary
        Sleep(50);
    }
    // request more idle-time, so that we can render the next loop!
    return TRUE;
}
```

3.11 The HeartBeat function

Now let's look at the heartbeat function. At the moment mine just draws a silly block in the top left of the window that changes color each frame.

```
BOOL CDirectDrawApp::HeartBeat()
{
    // Variables for the blocks color, in RGB format
    static r = 0;
    static g = 100;
    static b = 150;
    r++;
    g += 3;
    b += 2;
    if (r > 255) r = 0;
    if (g > 255) g = 0;
    if (b > 255) b = 0;

    // The destination rectangle on my 320x240 off-screen surface
    CRect rc(10,30,100,70);

    // We draw the rectangle using a blit.
    // This structure describes how to do the blit.
    DDBLTFX fx;
    // Zero out all fields of the blit effects structure
    memset(&fx, 0, sizeof(fx));
    // The dwSize field must contain the size of the structure
    fx.dwSize = sizeof(DDBLTFX);

    // Set the color we want to draw the rectangle
    fx.dwFillColor = RGB(r,g,b);

    // Blit. Note that we blit to the back buffer
    HRESULT hr;
    hr = pDDSBack->Blt(&rc, NULL, NULL, DDBLT_COLORFILL | DDBLT_WAIT,
        &fx);
    if (FAILED(hr)) {
        TRACE("Error blitting: ");
        TraceErrorDD(hr);
        return FALSE;
    }

    // Call our routine for flipping the surfaces
    FlipSurfaces();

    // No major errors
    return TRUE;
}
```

Note that Blt parameter DDBLT_WAIT. If a surface is busy then DirectDraw will return an error, without performing the blit. Passing the DDBLT_WAIT option will instruct DirectDraw to wait until the surface becomes available and then perform the blit.

3.12 Flipping surfaces

Now let's look at the function that performs the surface flipping. This time I *do* take full-screen mode into account ... I'm inconsistent, eh? :-).

```
UINT FlipSurfaces()
{
    HRESULT hr;
    // source and destination rectangles
    RECT rcSrc;
    RECT rcDest;
    POINT p;

    // Make sure no surfaces have been lost
    CheckSurfaces();

    // if we're windowed do the blit, else just Flip
    if (!bFullScreen)
    {
        // find out where on the primary surface our window lives
        p.x = 0; p.y = 0;
        ::ClientToScreen(ddWnd, &p);
        ::GetClientRect(ddWnd, &rcDest);
        OffsetRect(&rcDest, p.x, p.y);
        SetRect(&rcSrc, 0, 0, 320, 240);
        hr = pDDSPrimary->Blt(&rcDest, pDDSBack, &rcSrc, DDBLT_WAIT,
            NULL);
    } else {
        hr = pDDSPrimary->Flip(NULL, DDFLIP_WAIT);
    }

    // success!
    return 0;
}
```

3.13 Tracing DirectDraw errors

Here is a pretty useful debugging function for DirectX apps, called **TraceErrorDD**:

```
void TraceErrorDD(HRESULT hErr)
{
    switch (hErr)
    {
        case DDERR_ALREADYINITIALIZED:
                TRACE("DDERR_ALREADYINITIALIZED"); break;
        case DDERR_CANNOTATTACHSURFACE:
                TRACE("DDERR_CANNOTATTACHSURFACE"); break;
        case DDERR_CANNOTDETACHSURFACE:
                TRACE("DDERR_CANNOTDETACHSURFACE"); break;
        case DDERR_CURRENTLYNOTAVAIL:
                TRACE("DDERR_CURRENTLYNOTAVAIL"); break;
        case DDERR_EXCEPTION:
```

```
        TRACE("DDERR_EXCEPTION"); break;
case DDERR_GENERIC:
        TRACE("DDERR_GENERIC"); break;
case DDERR_HEIGHTALIGN:
        TRACE("DDERR_HEIGHTALIGN"); break;
case DDERR_INCOMPATIBLEPRIMARY:
        TRACE("DDERR_INCOMPATIBLEPRIMARY"); break;
case DDERR_INVALIDCAPS:
        TRACE("DDERR_INVALIDCAPS"); break;
case DDERR_INVALIDCLIPLIST:
        TRACE("DDERR_INVALIDCLIPLIST"); break;
case DDERR_INVALIDMODE:
        TRACE("DDERR_INVALIDMODE"); break;
case DDERR_INVALIDOBJECT:
        TRACE("DDERR_INVALIDOBJECT"); break;
case DDERR_INVALIDPARAMS:
        TRACE("DDERR_INVALIDPARAMS"); break;
case DDERR_INVALIDPIXELFORMAT:
        TRACE("DDERR_INVALIDPIXELFORMAT"); break;
case DDERR_INVALIDRECT:
        TRACE("DDERR_INVALIDRECT"); break;
case DDERR_LOCKEDSURFACES:
        TRACE("DDERR_LOCKEDSURFACES"); break;
case DDERR_NO3D:
        TRACE("DDERR_NO3D"); break;
case DDERR_NOALPHAHW:
        TRACE("DDERR_NOALPHAHW"); break;
case DDERR_NOCLIPLIST:
        TRACE("DDERR_NOCLIPLIST"); break;
case DDERR_NOCOLORCONVHW:
        TRACE("DDERR_NOCOLORCONVHW"); break;
case DDERR_NOCOOPERATIVELEVELSET:
        TRACE("DDERR_NOCOOPERATIVELEVELSET"); break;
case DDERR_NOCOLORKEY:
        TRACE("DDERR_NOCOLORKEY"); break;
case DDERR_NOCOLORKEYHW:
        TRACE("DDERR_NOCOLORKEYHW"); break;
case DDERR_NODIRECTDRAWSUPPORT:
        TRACE("DDERR_NODIRECTDRAWSUPPORT"); break;
case DDERR_NOEXCLUSIVEMODE:
        TRACE("DDERR_NOEXCLUSIVEMODE"); break;
case DDERR_NOFLIPHW:
        TRACE("DDERR_NOFLIPHW"); break;
case DDERR_NOGDI:
        TRACE("DDERR_NOGDI"); break;
case DDERR_NOMIRRORHW:
        TRACE("DDERR_NOMIRRORHW"); break;
case DDERR_NOTFOUND:
        TRACE("DDERR_NOTFOUND"); break;
case DDERR_NOOVERLAYHW:
```

```
        TRACE("DDERR_NOOVERLAYHW"); break;
case DDERR_NORASTEROPHW:
        TRACE("DDERR_NORASTEROPHW"); break;
case DDERR_NOROTATIONHW:
        TRACE("DDERR_NOROTATIONHW"); break;
case DDERR_NOSTRETCHHW:
        TRACE("DDERR_NOSTRETCHHW"); break;
case DDERR_NOT4BITCOLOR:
        TRACE("DDERR_NOT4BITCOLOR"); break;
case DDERR_NOT4BITCOLORINDEX:
        TRACE("DDERR_NOT4BITCOLORINDEX"); break;
case DDERR_NOT8BITCOLOR:
        TRACE("DDERR_NOT8BITCOLOR"); break;
case DDERR_NOTEXTUREHW:
        TRACE("DDERR_NOTEXTUREHW"); break;
case DDERR_NOVSYNCHW:
        TRACE("DDERR_NOVSYNCHW"); break;
case DDERR_NOZBUFFERHW:
        TRACE("DDERR_NOZBUFFERHW"); break;
case DDERR_NOZOVERLAYHW:
        TRACE("DDERR_NOZOVERLAYHW"); break;
case DDERR_OUTOFCAPS:
        TRACE("DDERR_OUTOFCAPS"); break;
case DDERR_OUTOFMEMORY:
        TRACE("DDERR_OUTOFMEMORY"); break;
case DDERR_OUTOFVIDEOMEMORY:
        TRACE("DDERR_OUTOFVIDEOMEMORY"); break;
case DDERR_OVERLAYCANTCLIP:
        TRACE("DDERR_OVERLAYCANTCLIP"); break;
case DDERR_OVERLAYCOLORKEYONLYONEACTIVE:
        TRACE("DDERR_OVERLAYCOLORKEYONLYONEACTIVE"); break;
case DDERR_PALETTEBUSY:
        TRACE("DDERR_PALETTEBUSY"); break;
case DDERR_COLORKEYNOTSET:
        TRACE("DDERR_COLORKEYNOTSET"); break;
case DDERR_SURFACEALREADYATTACHED:
        TRACE("DDERR_SURFACEALREADYATTACHED"); break;
case DDERR_SURFACEALREADYDEPENDENT:
        TRACE("DDERR_SURFACEALREADYDEPENDENT"); break;
case DDERR_SURFACEBUSY:
        TRACE("DDERR_SURFACEBUSY"); break;
case DDERR_CANTLOCKSURFACE:
        TRACE("DDERR_CANTLOCKSURFACE"); break;
case DDERR_SURFACEISOBSCURED:
        TRACE("DDERR_SURFACEISOBSCURED"); break;
case DDERR_SURFACELOST:
        TRACE("DDERR_SURFACELOST"); break;
case DDERR_SURFACENOTATTACHED:
        TRACE("DDERR_SURFACENOTATTACHED"); break;
case DDERR_TOOBIGHEIGHT:
```

```
        TRACE("DDERR_TOOBIGHEIGHT"); break;
case DDERR_TOOBIGSIZE:
        TRACE("DDERR_TOOBIGSIZE"); break;
case DDERR_TOOBIGWIDTH:
        TRACE("DDERR_TOOBIGWIDTH"); break;
case DDERR_UNSUPPORTED:
        TRACE("DDERR_UNSUPPORTED"); break;
case DDERR_UNSUPPORTEDFORMAT:
        TRACE("DDERR_UNSUPPORTEDFORMAT"); break;
case DDERR_UNSUPPORTEDMASK:
        TRACE("DDERR_UNSUPPORTEDMASK"); break;
case DDERR_VERTICALBLANKINPROGRESS:
        TRACE("DDERR_VERTICALBLANKINPROGRESS"); break;
case DDERR_WASSTILLDRAWING:
        TRACE("DDERR_WASSTILLDRAWING"); break;
case DDERR_XALIGN:
        TRACE("DDERR_XALIGN"); break;
case DDERR_INVALIDDIRECTDRAWGUID:
        TRACE("DDERR_INVALIDDIRECTDRAWGUID"); break;
case DDERR_DIRECTDRAWALREADYCREATED:
        TRACE("DDERR_DIRECTDRAWALREADYCREATED"); break;
case DDERR_NODIRECTDRAWHW:
        TRACE("DDERR_NODIRECTDRAWHW"); break;
case DDERR_PRIMARYSURFACEALREADYEXISTS:
        TRACE("DDERR_PRIMARYSURFACEALREADYEXISTS"); break;
case DDERR_NOEMULATION:
        TRACE("DDERR_NOEMULATION"); break;
case DDERR_REGIONTOOSMALL:
        TRACE("DDERR_REGIONTOOSMALL"); break;
case DDERR_CLIPPERISUSINGHWND:
        TRACE("DDERR_CLIPPERISUSINGHWND"); break;
case DDERR_NOCLIPPERATTACHED:
        TRACE("DDERR_NOCLIPPERATTACHED"); break;
case DDERR_NOHWND:
        TRACE("DDERR_NOHWND"); break;
case DDERR_HWNDSUBCLASSED:
        TRACE("DDERR_HWNDSUBCLASSED"); break;
case DDERR_HWNDALREADYSET:
        TRACE("DDERR_HWNDALREADYSET"); break;
case DDERR_NOPALETTEATTACHED:
        TRACE("DDERR_NOPALETTEATTACHED"); break;
case DDERR_NOPALETTEHW:
        TRACE("DDERR_NOPALETTEHW"); break;
case DDERR_BLTFASTCANTCLIP:
        TRACE("DDERR_BLTFASTCANTCLIP"); break;
case DDERR_NOBLTHW:
        TRACE("DDERR_NOBLTHW"); break;
case DDERR_NODDROPSHW:
        TRACE("DDERR_NODDROPSHW"); break;
case DDERR_OVERLAYNOTVISIBLE:
```

```
            TRACE("DDERR_OVERLAYNOTVISIBLE"); break;
        case DDERR_NOOVERLAYDEST:
            TRACE("DDERR_NOOVERLAYDEST"); break;
        case DDERR_INVALIDPOSITION:
            TRACE("DDERR_INVALIDPOSITION"); break;
        case DDERR_NOTAOVERLAYSURFACE:
            TRACE("DDERR_NOTAOVERLAYSURFACE"); break;
        case DDERR_EXCLUSIVEMODEALREADYSET:
            TRACE("DDERR_EXCLUSIVEMODEALREADYSET"); break;
        case DDERR_NOTFLIPPABLE:
            TRACE("DDERR_NOTFLIPPABLE"); break;
        case DDERR_CANTDUPLICATE:
            TRACE("DDERR_CANTDUPLICATE"); break;
        case DDERR_NOTLOCKED:
            TRACE("DDERR_NOTLOCKED"); break;
        case DDERR_CANTCREATEDC:
            TRACE("DDERR_CANTCREATEDC"); break;
        case DDERR_NODC:
            TRACE("DDERR_NODC"); break;
        case DDERR_WRONGMODE:
            TRACE("DDERR_WRONGMODE"); break;
        case DDERR_IMPLICITLYCREATED:
            TRACE("DDERR_IMPLICITLYCREATED"); break;
        case DDERR_NOTPALETTIZED:
            TRACE("DDERR_NOTPALETTIZED"); break;
        case DDERR_UNSUPPORTEDMODE:
            TRACE("DDERR_UNSUPPORTEDMODE"); break;
        case DDERR_NOMIPMAPHW:
            TRACE("DDERR_NOMIPMAPHW"); break;
        case DDERR_INVALIDSURFACETYPE:
            TRACE("DDERR_INVALIDSURFACETYPE"); break;
        case DDERR_DCALREADYCREATED:
            TRACE("DDERR_DCALREADYCREATED"); break;
        case DDERR_CANTPAGELOCK:
            TRACE("DDERR_CANTPAGELOCK"); break;
        case DDERR_CANTPAGEUNLOCK:
            TRACE("DDERR_CANTPAGEUNLOCK"); break;
        case DDERR_NOTPAGELOCKED:
            TRACE("DDERR_NOTPAGELOCKED"); break;
        case DDERR_NOTINITIALIZED:
            TRACE("DDERR_NOTINITIALIZED"); break;
        default:
            TRACE("Unknown Error"); break;
        }
        TRACE("\n");
}
```

3.14 Cleaning up

Coming soon.

Article updated: 19 June 1998

Article by David Joffe
http://www.geocities.com/SoHo/Lofts/2018/

David Joffe's Guide to Programming Games with DirectX

Chapter 4: A simple Direct3D Retained mode sample

4.1 Direct3D: An Overview

Over here I'll shove in some basics, like coordinate systems, world and object coordinate systems, etc. For now I'll assume you're at least a little familiar with 3D programming. Blah blah blah, differences between immediate and retained mode, etc etc.

4.1.1 Devices

Direct3D interfaces with the surface it is rendering to (e.g. screen memory, system memory) using an **IDirect3DRMDevice** object. More than one type of rendering device can exist and a specific rendering device must be chosen for a scene. For example, there is normally a device for RGB rendering and a device for Mono rendering (these names refer to the lighting model used for rendering. Mono means that only white lights can exist in the scene, while RGB supports colored lights, and is thus slower). Additional devices may be installed that make use of 3D hardware acceleration. It is possible to iterate through the installed D3D devices by enumarating through them (**EnumDevices**). It is possible to have two different devices rendering to the same

surface.

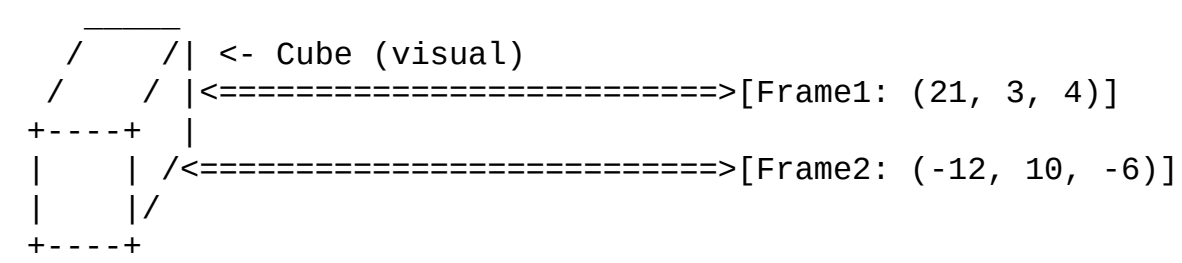
4.1.2 Viewports

The **IDirect3DRMViewport** object is used to keep track of how our 3D scene is rendered onto the device. It is possible to have multiple viewports per device, and it is also possible to have a viewport rendering to more than one device. The viewport object keeps track of the camera, front and back clipping fields, field of view etc.

4.1.3 Frames

A **frame** in Direct3D is basically used to store an object's position and orientation information, relative to a given *frame of reference*, which is where the term *frame* comes from. Frames are positioned relative to other frames, or to the world coordinates. Frames are used to store the positions of objects in the scene as well as other things like lights. OK, so I'm explaining it badly. It's late, I'm tired, I'll revise it soon. To add an object to the scene we have to attach the object to a frame. The object is called a *visual* in Direct3D, since it represents what the user *sees*. So, a visual has no meaningful position or orientation information itself, but when attached to a frame, it is transformed when rendered according to the transformation information in the frame. Multiple frames may use the same visual. This can save a lot of time and memory in a situation like, for example, a forest or a small fleet of spacecraft, where you have a bunch of objects that look exactly the same but all exist in different positions and orientations.

Here is a crummy ASCII diagram of a single visual attached to two frames which are at different positions:



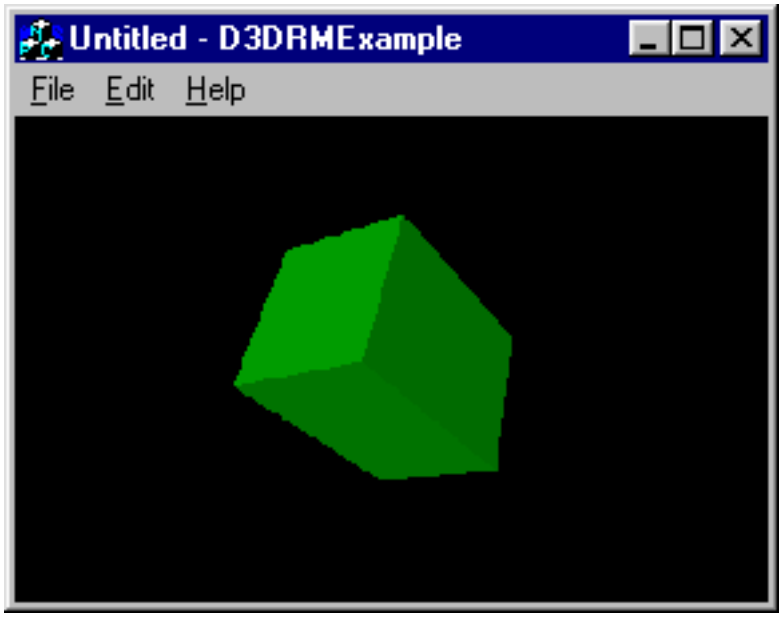
If both of these frames were attached to the scene frame, then our scene would have **2** cubes in it; one at (21, 3, 4) and the other at (-12, 10, -6).

4.1.4 Materials

Coming sometime.

4.2 The Direct3D RM Sample

Firstly, heres a screenshot of the small ***simple*** sample application we're putting together here.



4.3 Setting up global variables

Before we start we'll need a few global variables.

```
LPDIRECTDRAW pDD;                // A DirectDraw object
LPDIRECT3DRM pD3DRM;             // A Direct3D RM object
LPDIRECTDRAWSURFACE pDDSPrimary; // DirectDraw primary surface
LPDIRECTDRAWSURFACE pDDSBack;    // DirectDraw back surface
LPDIRECTDRAWPALETTE pDDPal;      // Palette for primary surface
LPDIRECTDRAWCLIPPER pClipper;    // Clipper for windowed mode
LPDIRECT3DRMDEVICE pD3DRMDevice; // A device
LPDIRECT3DRMVIEWPORT pViewport;  // A viewport
LPDIRECT3DRMFRAME pCamera;       // A camera
LPDIRECT3DRMFRAME pScene;        // The scene
LPDIRECT3DRMFRAME pCube;         // The one and only object in
                                 // our scene
BOOL bFullScreen;                // Are we in full-screen mode?
BOOL bAnimating;                 // Has our animating begun?
HWND ddWnd;                      // HWND of the DDraw window
```

Note that we need both a DirectDraw object and a Direct3D object to create a Direct3D application. This is because Direct3D works in conjunction with DirectDraw. As before, we need a primary and a back surface for our double-buffering, and a clipper to handle window-clipping in windowed mode. The palette object is still not discusses in this tutorial (yet). We have objects for the device and viewport, and we have **frame** objects to keep track of the scene and the scene's camera. Also, we have a frame that is used for the object we'll have in this scene.

Here is a routine just to initially flatten these globals:

```
void InitDirectXGlobals()
{
```

```
    pDD = NULL;
    pD3DRM = NULL;
    pDDSPrimary = NULL;
    pDDSBack = NULL;
    pDDPal = NULL;
    pClipper = NULL;
    pD3DRMDevice = NULL;
    pViewport = NULL;
    pCamera = NULL;
    pScene = NULL;
    pCube = NULL;

    bFullScreen = FALSE;
    bAnimating = FALSE;
}
```

4.4 From 'Initializing the DirectDraw system' to 'Creating the clipper'

These steps all proceed exactly as in the DirectDraw sample, with the exception of the CreateSurface function, where the back surface has to created with the DDSCAPS_3DDEVICE, since it will be used for 3d rendering:

```
UINT CreatePrimarySurface()
{
    .
    .
    .
    // Create an offscreen surface, specifying 3d device
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_3DDEVICE;
    .
    .
    .
}
```

4.5 Creating the Direct3D Retained Mode object

Now we need to create an **IDirect3DRM** object. This is achieved, quite simply, by calling the **Direct3DRMCreate** function.

```
UINT CreateDirect3DRM()
{
    HRESULT hr;
    // Create the IDirect3DRM object.
    hr = Direct3DRMCreate(&pD3DRM);
    if (FAILED(hr)) {
        TRACE("Error creating Direct3d RM object\n");
        return 1;
    }
    return 0;
}
```

4.6 Creating the device for rendering

We create the device object from the back surface, since this surface is the one we will render to.

```
UINT CreateDevice()
{
    HRESULT hr;
    hr = pD3DRM->CreateDeviceFromSurface(
        NULL, pDD, pDDSBack, &pD3DRMDevice);
    if (FAILED(hr)) {
        TRACE("Error %d creating d3drm device\n", int(LOWORD(hr)));
        return 1;
    }
    // success
    return 0;
}
```

4.7 Creating the viewport

We do a bit more than just create the viewport here. We create the scene object and the camera object, as well as set the ambient light for the scene, and create a directional light.

```
UINT CreateViewport()
{
    HRESULT hr;

    // First create the scene frame
    hr = pD3DRM->CreateFrame(NULL, &pScene);
    if (FAILED(hr)) {
        TRACE("Error creating the scene frame\n");
        return 1;
    }

    // Next, create the camera as a child of the scene
    hr = pD3DRM->CreateFrame(pScene, &pCamera);
    if (FAILED(hr)) {
        TRACE("Error creating the scene frame\n");
        return 2;
    }
    // Set the camera to lie somewhere on the negative z-axis, and
    // point towards the origin
    pCamera->SetPosition(
        pScene, D3DVAL(0.0), D3DVAL(0.0), D3DVAL(-300.0));
    pCamera->SetOrientation(
        pScene,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
        D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0));
```

```
    // create lights
    LPDIRECT3DRMLIGHT pLightAmbient = NULL;
    LPDIRECT3DRMLIGHT pLightDirectional = NULL;
    LPDIRECT3DRMFRAME pLights = NULL;

    // Create two lights and a frame to attach them to
    // I haven't quite figured out the CreateLight's second
    // parameter yet.
    pD3DRM->CreateFrame(pScene, &pLights);
    pD3DRM->CreateLight(D3DRMLIGHT_AMBIENT, pD3DRMCreateColorRGB(
        D3DVALUE(0.3), D3DVALUE(0.3), D3DVALUE(0.3)),
        &pLightAmbient);
    pD3DRM->CreateLight(D3DRMLIGHT_DIRECTIONAL, D3DRMCreateColorRGB(
        D3DVALUE(0.8), D3DVALUE(0.8), D3DVALUE(0.8)),
        &pLightDirectional);

    // Orient the directional light
    pLights->SetOrientation(pScene,
        D3DVALUE(30.0), D3DVALUE(-20.0), D3DVALUE(50.0),
        D3DVALUE(0.0), D3DVALUE(1.0), D3DVALUE(0.0));

    // Add ambient light to the scene, and the directional light
    // to the pLights frame
    pScene->AddLight(pLightAmbient);
    pLights->AddLight(pLightDirectional);

    // Create the viewport on the device
    hr = pD3DRM->CreateViewport(pD3DRMDevice,
        pCamera, 10, 10, 300, 220, &pViewport);
    if (FAILED(hr)) {
        TRACE("Error creating viewport\n");
        return 3;
    }
    // set the back clipping field
    hr = pViewport->SetBack(D3DVAL(5000.0));

    // Release the temporary lights created. It seems
    // they will have been copied for the scene during AddLight
    pLightAmbient->Release();
    pLightDirectional->Release();

    // success
    return 0;
}
```

4.8 Creating your scene

We need some objects for the scene. Let's create a cube! Wheee! Here is the **cube.x** file, zipped. Unzip it into the directory from which your app runs.

SetRotation assigns values in radians to a frame that it must rotate around it's local axis each time the frame is rendered using **Tick**. Direct3D automatically performs this rotation when **Tick** is called.

```
// Put stuff into the scene
UINT CreateDefaultScene()
{
    HRESULT hr;

    // Create the frame for the cube
    hr = pD3DRM->CreateFrame(pScene, &pCube);
    if (FAILED(hr)) {
        TRACE("Error creating frame pCube\n");
        return 1;
    }
    // Load the cube
    hr = pCube->Load("CUBE.X", NULL, D3DRMLOAD_FROMFILE, NULL, NULL);
    if (FAILED(hr)) {
        TRACE("Error [%d] loading cube.x\n", int(LOWORD(hr)));
        return 2;
    }
    // Set cube's position/orientation relative to scene
    pCube->SetPosition(pScene,
        D3DVALUE(0.0), D3DVALUE(0.0), D3DVALUE(0.0));
    pCube->SetRotation(pScene,
        D3DVALUE(1.0), D3DVALUE(1.0), D3DVALUE(0.0), D3DVALUE(0.1));

    // success
    return 0;
}
```

4.9 Putting it all together

Here is the tail-end of the app's **InitInstance** function:

```
    InitDirectXGlobals();
    TRACE("Calling InitDDraw\n");
    InitDDraw();
    SetMode();
//    TRACE("Calling LoadJascPalette\n");
//    LoadJascPalette("inspect.pal", 10, 240);
    TRACE("Calling CreatePrimarySurface\n");
    CreatePrimarySurface();
    TRACE("Calling CreateClipper\n");
    CreateClipper();
//    TRACE("Calling AttachPalette\n");
//    AttachPalette(pDDPal);
    TRACE("Calling CreateDirect3DRM\n");
    CreateDirect3DRM();
    TRACE("Calling CreateDevice\n");
```

```
    CreateDevice();
    TRACE("Calling CreateViewport\n");
    CreateViewport();
    TRACE("Calling CreateDefaultScene\n");
    CreateDefaultScene();

    bAnimating = TRUE;

    return TRUE;
}
```

4.10 Restoring lost surfaces

Same as the DirectDraw sample:

```
BOOL CheckSurfaces()
{
    // Check the primary surface
    if (pDDSPrimary) {
        if (pDDSPrimary->IsLost() == DDERR_SURFACELOST) {
            pDDSPrimary->Restore();
            return FALSE;
        }
    }
    return TRUE;
}
```

4.11 The Rendering loop

Same as the DirectDraw sample:

```
BOOL CD3dRmAppApp::OnIdle(LONG lCount)
{
    CWinApp::OnIdle(lCount);
    if (bAnimating) {
        HeartBeat();
        Sleep(50);
    }
    return TRUE;
}
```

4.12 The HeartBeat function

```
BOOL CD3dRmAppApp::HeartBeat()
{
    HRESULT hr;
//    if (!CheckSurfaces) bForceUpdate = TRUE;
//    if (bForceUpdate) pViewport->ForceUpdate(10,10,300,220);
```

```
    hr = pD3DRM->Tick(D3DVALUE(1.0));
    if (FAILED(hr)) {
        TRACE("Tick error!\n");
        return FALSE;
    }

    // Call our routine for flipping the surfaces
    FlipSurfaces();

    // No major errors
    return TRUE;
}
```

4.13 Flipping surfaces

Same as for DirectDraw sample.

4.14 Tracing Direct3D errors

Coming soon(?).

4.15 Cleaning up

Coming soonish(?).

Article updated: 19 June 1998

Article by David Joffe
http://www.geocities.com/SoHo/Lofts/2018/

David Joffe's Guide to Programming Games with DirectX

Chapter 5: Miscellaneous unsorted new stuff

conv3ds.exe

conv3ds -X: save templates. Nothing to worry about, part of the nature of x-files. Useful though if you want to see what the various parameters of things (like materials) mean.

conv3ds -m: merge all objects into one single mesh. Useful if you don't need separate meshes, and can speed up loading.

conv3ds -x: Save as text file: Definitely not a bad idea, especially when debugging. It helps to have a look at the files etc

conv3ds -T: Create a top-level frame with all objects as child frames. This is useful since the pFrame->Load loads only the first frame it finds in the file.

conv3ds -v : Verbose mode.

Possible bug (possibly not in v5, only in v3) .. seems to need XForms reset in 3dstudio or strange things happen. Will maybe check this out.

When using a pFrame object to load a mesh from a mesh file, and you want to retain (more coming)

Coming:

Plan to learn OpenGL: Comparison etc, as unbiased as I can muster up.

Enumerating device drivers

Callbacks

RM vs IM

A decent sample with D/L'able code.

Comments on MS samples

How to compile with NMAKE

faq: main: linker error

3D math

Fog + some other frame rate optimization techniques

Loading and displaying a .bmp (or .png :)

Execute buffers vs DrawPrimitive

5.1 Drawing a pixel with DirectDraw

Microsoft did not provide a "PutPixel"-type function with DirectDraw to draw a pixel onto a DirectDraw surface. Granted, no real game is going to draw its graphics using a PutPixel routine, since it would be too slow - but it would have been nice to have such a "convenience function", and it could also be useful for testing out stuff. Some applications that might not necessarily need speed could also benefit from the convenience of a PutPixel routine. But, DirectX was not designed with convenience in mind.

Anyway, there are a number of ways to go about writing a PutPixel routine. One way would be to use GDI functions on the surface, using GetDC(). Another way is to use the IDirectDrawSurface::Blt() function, specifying a solid color operation with a 1x1 rectangle. A third way is to obtain a pointer to the memory representing the surface by calling IDirectDrawSurface::Lock(), and drawing the pixel in memory yourself. Each of these methods demonstrates general techniques whose usefulness goes beyond a simple PutPixel function.

5.1.1 PutPixel by surface locking

The basic technique here is to obtain a pointer to the memory representing the surface. You then calculate the offset into that memory for the coordinates of the pixel you wish to draw, and then you write the bytes representing the color of the pixel you want into that position in memory. There is nothing complex or difficult here, it's just a lot of grunt-work and jumping through hoops.

By default, you may not just write directly into the memory of a surface. You first have to obtain exclusive access to that surface, by "locking" it. This is done with a call to **`Lock()`**.

```
HRESULT IDirectDrawSurface::Lock( LPRECT pRect, LPDDSURFACEDESC pDDSD,
DWORD dwFlags, HANDLE hEvent );

pRect
      Area of surface to lock. NULL locks entire surface.
pDDSD
      Points to struct that describes the surface.
dwFlags
      Control flags.
hEvent
```

```
        Unused. Microsoft loves these unused parameters.
```

The pointer to the surface memory is one of the members of the DDSURFACEDESC structure, and is filled by the call to **`Lock()`** with a valid pointer. You may then write to this memory, and when you are finished with it you must call **`Unlock`** on that surface.

Now, to draw a pixel, you need to understand how pixels are represented in memory, which will depend on what color depth screen mode your surface is (see Chapter 2). Of course, if you create a "generic" PutPixel routine, that will work no matter what mode you are in, you will probably want to specify the color in some human-readable format, like an RGB triple. The PutPixel routine must then convert this, store it somewhere, and dump it to the surface memory. I use an **`unsigned int`** to temporarily store the pixel value. For convenience, I have created a **`CreateRGB`** function that can generate this **`unsigned int`** from supplied RGB values which range from 0 to 255:

```
/*-------------------------------------------------------------------------*/
// Create color from RGB triple
unsigned int CreateRGB( int r, int g, int b )
{
   unsigned int pixel;
   switch (g_iBpp)
   {
   case 8:
      // Here you should do a palette lookup to find the closest match.
      // I'm not going to bother with that. Many modern games no
      // longer support 256-color modes.
      break;
   case 16:
      // Break down r,g,b into 5-6-5 format.
      pixel = ((r/8)<<11) | ((g/4)<<5) | (b/8);
      break;
   case 24:
   case 32:
      pixel = (r<<16) | (g<<8) | (b);
      break;
   default:
      pixel =0;
   }
   return pixel;
}
/*-------------------------------------------------------------------------*/
```

Here is the actual PutPixel routine, taking as input a DirectDraw surface, an (x,y) location, and an RGB triple with r,g and b in the range 0 to 255. The function returns 0 for success. Notice the function, like CreateRGB, assumes the existence of g_iBpp, which I will explain in a bit.

```
/*-------------------------------------------------------------------------*/
int PutPixel( LPDIRECTDRAWSURFACE pDDS, int x, int y, int r, int g, int b )
{
   DDSURFACEDESC   ddsd;       // Surface description
   unsigned int    pixel;      // Store pixel to be dumped to screen
   int             offset;     // Store offset of destination memory location
   unsigned char * szSurface;  // Store pointer to surface
   HRESULT         hr;
```

```
    int             pixelwidth; // The width of a single pixel, in bytes

    // Initialize struct information
    ddsd.dwSize = sizeof(ddsd);

    // Calculate how the pixel is represented in memory
    pixel = CreateRGB( r, g, b );

    // Lock entire surface, wait if it is busy, return surface memory pointer
    hr = pDDS->Lock( NULL, &ddsd, DDLOCK_WAIT | DDLOCK_SURFACEMEMORYPTR, NULL );
    if (FAILED(hr))
       return -1;

    // Get surface memory pointer
    szSurface = (unsigned char*)ddsd.lpSurface;

    // Calculate the width of a pixel in bytes - this is how many bytes
    // we must write to the surface
    switch (g_iBpp)
    {
    case 16: pixelwidth = 2; break;
    case 24: pixelwidth = 3; break;
    case 32: pixelwidth = 4; break;
    default: pixelwidth = 1;
    }

    // Calculate memory offset
    // (Notice I use dwPitch instead of dwWidth - see Chapter 2)
    offset = (y*ddsd.lPitch + x*pixelwidth);

    // Copy pixel onto the surface, need string.h for this
    memcpy( szSurface + offset, &pixel, pixelwidth );

    // Unlock the surface. The parameter may be NULL if you locked the entire
    // surface, otherwise it must be a pointer to the DirectDraw surface memory.
    hr = pDDS->Unlock( NULL );
    if (FAILED(hr))
       return -2;

    return 0;
}
/*-----------------------------------------------------------------------*/
```

There are a few things worth mentioning about this function. Firstly, it can be optimized quite a lot, I know. But in a production game, this is not the function you will be using for all your drawing routines anyway. (One of the rules of optimization - never optimise what you don't need to.) Secondly, it makes some crass assumptions about your architecture - it may not work if your system is big endian. However, the Intel x86 is little endian, and as far as I know DirectX is only really available on the Intel x86. Thirdly, it only knows about a limited number of screen modes, and sometime in the future, when people start using video modes with higher bit depths (like 48), the code may no longer work. (However, as far as I can tell, DirectDraw itself may not be able to handle 48-bit color depths.) Anyway, the point is that to make the above PutPixel routine "correct" requires quite a bit more work still.

The `g_iBpp` in the code above is an integer representing the bits per pixel of the screen mode of the surface. You can get this from the DDSURFACEDESC.

Something else worth mentioning is that if the DirectDraw surface here is the primary surface - that is, it represents the entire screen, and you are in windowed mode and not in full-screen mode - then the pixel offsets here are relative to the top left of the screen, and not to your application window. You will find yourself drawing all over the screen, unless you work the offset of your window into your calculations (eg by using the ClientToScreen Win32 call or something.)

5.1.2 PutPixel by using IDirectDrawSurface::Blt

This routine is demonstrated in the supplied sample DDSamp.

This method also makes use of the **`CreateRGB`** function described in 5.1.1, so go read that part now if you have not done so yet.

This method is quite simple, and probably a bit "safer" than the method described in 5.1.1.

```
/*-----------------------------------------------------------------------*/
// PutPixel routine for a DirectDraw surface
void DDPutPixel( LPDIRECTDRAWSURFACE pDDS, int x, int y, int r, int g, int b )
{
   HRESULT hr;
   DDBLTFX ddbfx;
   RECT    rcDest;
   POINT   p;

   // Safety net
   if (g_pDDS == NULL)
      return;

   // Initialize the DDBLTFX structure with the pixel color
   ddbfx.dwSize = sizeof( ddbfx );
   ddbfx.dwFillColor = (DWORD)CreateRGB( r, g, b );

   // Prepare the destination rectangle as a 1x1 (1 pixel) rectangle
   p.x = x;
   p.y = y;
   ClientToScreen( g_hWnd, &p );
   OffsetRect( &rcDest, p.x, p.y );
   SetRect( &rcDest, p.x, p.y, p.x+1, p.y+1 );

   // Blit 1x1 rectangle using solid color op
   hr = g_pDDS->Blt( &rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL, &ddbfx );
   if (FAILED(hr))
      OutputDebugString( (LPCTSTR)"Royal fuckup\n" );
}
/*-----------------------------------------------------------------------*/
```

All I've done here is:

- Create and set up a destination rectangle 1x1 pixels large
- Prepare the DDBLTFX structure with the solid color required
- Call **`Blt`** on the surface, specifying DDBLT_COLORFILL as the blit operation

That's about it.

5.2 Some words about surface locking

Before you go running off locking every surface you see now, it must first be mentioned that there are some limitations to surface locking (what did you expect?) These are summarized from the DirectX help files.

- Never assume a constant display pitch - this can change for various reasons, so check the pitch before you do anything.
- Never blit to a locked surface. The call will fail, since the surface is "busy".
- Limit what you do while the surface is locked. This is important - Windows has a braindamaged mutex known as the Win16 lock, which lingers from the days of 16-bit Windows. While any thread holds this mutex, Windows is effectively dead, it cannot do anything. Even your debugger stops working while the Win16Lock is held. If an application crashes while holding this mutex, you have to press that ol' well-worn reset button. While a surface is locked, DirectDraw grabs this mutex (because the Win16Lock "serialises access to GDI and USER"). Calls to **`IDirectDrawSurface::GetDC`** and **`IDirectDrawSurface::ReleaseDC`** also implicitly call Lock and Unlock, so use those with caution as well.

Note that you can have multiple rectangles on a surface locked at the same time, so long as the rectangles don't overlap.

The DirectX documentation also makes the following warning which I don't fully understand:

> Copy aligned to display memory. Windows 95 uses a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to DirectDraw. Copying unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.

---

dj5.html; created: 19 June 1998. Updated 21 April 1999.

http://www.geocities.com/SoHo/Lofts/2018/