

# Chapter 2

## Cranking up Direct3D

- get acquainted with Direct3D
- learn about its main objects
- analyze the graphics adapter
- get Direct3D rocking

*"I know Kung-Fu"  
(Neo, The Matrix)*

### 2.1 Warming up Exercises

Okay, there is only one rule: Everybody fights, no one quits. You might have heard a lot about DirectX in general or about Direct3D in particular. But the very first thing the so called experts will eagerly tell you as a *newbie* in 3D programming is that it is so complicated it will scare the hell out of you. My advice about such self made experts is that you better stop listening to them right now.

Sure, there had been times where programming 3D graphics was a pain in the ass and not that much fun as it is today. Talking about Direct3D there had also been times where using Direct3D was a rather tricky thing. Up to version 7 DirectX required you to work with a 2d graphics interface and couple it to some kind of 3D add-on interface. Once you got it the whole initialization thing was easy but ever since people tend to blame DirectX even up to now to be very complicated.

In this chapter I will show you that this is no longer true. First, I will explain you some things about the architecture of Direct3D and how it relates to

the whole DirectX package. I will also show you how to install and setup everything you need to get started. But then I will just dump you into the cold water. You are here to learn about Direct3D, aren't you?

I don't want to waste page after page talking chitchat. Keep your helmet ready as I will just throw some code at you to look at and study it. You will learn that initializing Direct3D and make it run takes no more than roughly 300 lines of source code in normal formatting. Those lines will already include all the Windows things like the message loop and the WinMain() function. You can even shrink this application down to half the size but then you have to make unhealthy assumptions about the capabilities of the hardware the program runs on.

Most of the lines of code needed for the initialization of Direct3D are spent for enumerating such capabilities. For example, you need to check whether the physical runtime environment (the hardware present) supports a certain screen resolution and color bit depth for example.

Am I talking too much already? Damn, I knew it. Okay, to put it straight the sample of this chapter is a stable real world Windows application that initializes and uses Direct3D in a smart, flexible, and stable way. I will only be able to take over the screen or the client area of a window and clear that area to an arbitrary color. But this is what we came for, right?

## **2.1.1 What's at the stock for you?**

Don't expect bombastic 3D graphics to appear on your screen too soon in this course. There are way too many websites and even whole books out there showing you how to crank up Direct3D the dirty way and start rendering triangles in a nutshell. But later on that comes back to you as a deadly boomerang chopping of your virtual head. Did you ever try to run such samples on hardware not supporting the needed resolutions, bit depth, and hardware features? Or did you try to hit ALT+TAB while running those cut down straight to the point applications in full-screen?

In the first part of this book which contains four chapters you will not find cool 3D graphics rendered to your screen. Only in the fourth chapter you

will start to learn how to render things to the screen. But don't mistake the first three chapters for being boring or even worse unnecessary. If you want to learn Direct3D then just do it. And don't just copy source from websites while not exactly understanding what it is doing or why it is doing it (possibly) in a very bad way only suited for briefly demonstration purposes.

My intended goal in this book is not only to teach you Direct3D. I also want to teach you *applied* Direct3D meaning utilizing this valuable tool in a way that is meant for a real world tool or application. Not only for a tutorial website. This chapter is straight to the point from the implementation point of view. But this is the last chapter in this book [:-)] where you will get a freaky one file project doing each and everything with plain function calls. I only give you this free ride in order to enable you to see what is actually going on during the initialization of Direct3D without the class encapsulation overhead typically encountered in real world applications.

But in the next chapter I will also show you (along with other Direct3D related topics, of course) how to build a nice design around a Direct3D encapsulation. In fact we will be developing a small but smart 3D engine over the course of this book. This engine can be used for a lot more than just showing off or writing tutorials about Direct3D. It will be professional enough to serve you in a real project. And you will have learned enough in this book to integrate even more features and functionalities from Direct3D. But don't let me scare you away. Once you get a grip on object orientation and modern C++ design you will love it.

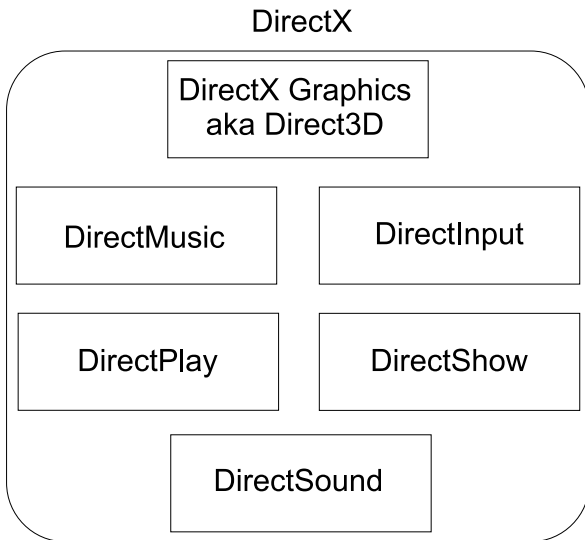
But for now, just concentrate on Direct3D.

## 2.1.2 How Direct3D is organized

Lots of people use DirectX and Direct3D as a part of it every day. Whether they are running some professional application or a video game. Lots of Windows software applications use DirectX without you ever noticing except for the installing requirements or an application's or a game's

DirectX auto-installer. But what exactly is installed when you put DirectX on your system?

Take a look at Figure 1 now. It shows you the main components that make up DirectX. There are some more but only small ones. So actually DirectX is not a single library of functions letting you access certain hardware components by standardized interfaces. It is a collection of libraries to do so.



*Figure 1: DirectX's main components*

Each component of DirectX is realized as a dynamic link library (DLL) of its own. If you want to use a certain component of DirectX you just have to link to the accompanying static library of the dynamic link library and include its header.

With regard to Direct3D, which is now officially called DirectX Graphics even if the interfaces still use the name Direct3D, there is also a huge helper library called D3DX for Direct3D Extensions. Naturally you also would have to link that one too as well as include its headers.

## 2.1.3 Setting up your Development Environment

Now that you know what DirectX is and what it contains the next question is how you can use it. You need to do several steps to be able to really start programming an application using DirectX. Those steps are the following ones:

1. Download and install the free DirectX SDK from Microsoft.
2. Add all needed DirectX directory paths to your IDE.
3. Include certain headers and link certain libraries in each new project.

I will now tell you a bit more about each of the steps involved. I suppose here that you are using a Microsoft IDE like Visual Studio or Visual C++ version 6.0, 7.0, 7.1 for example. All of those can deal with DirectX 9 but as far as I know the support for older versions of the Visual Studio are cut off since DirectX version 9.

*The acronym IDE stands for Integrated Development Environment and is describing tools like Visual Studio. An IDE is integrating a smart source code editor, a compiler, a linker, and a debugger for example. But it can contain even more tools and provide a uniform interface for all of them.*

### 2.1.3.1 Installing the DirectX SDK

It is not that difficult to understand that you need to have DirectX installed in order to be able to program with DirectX. But that is not the whole truth and you would now say that you already have DirectX installed because it came with the last video game you installed. But that is not totally right. There are two different types of DirectX. The first type is only needed if you want to run applications that are only using DirectX. It is also called the home user version of DirectX.

The second type of DirectX is the so called DirectX SDK (as opposed to the home user version) where SDK stands Software Developer Kit. As you can see from its name already this one is needed if you want to program application using DirectX. The SDK contains mainly header files and libraries you need to include and link, respectively.

But there is a lot more in this SDK. You will find a huge amount of samples and tutorials about DirectX. There is also a pretty good documentation which includes general introductions to 3D programming, introductions to each DirectX component, references for each component, and some more things. I will come to this documentation in a minute.

The best news for today is that you can download the whole DirectX SDK for free from Microsoft using the following URL:

<http://msdn.microsoft.com/library/default.asp?url=/downloads/list/directx.asp>

But don't be startled if you go on this site. The complete download for the SDK is about 190 megabyte in size. If you don't own a fast connection to the internet or don't have flatrate you should use a download manager like *Get Right* for example ([www.getright.com](http://www.getright.com)).

After you downloaded the SDK just execute the file to kick off the install process. You need to enter a directory to install DirectX to, and I will assume that you install it to the default `c:\dx9sdk\` if I need to refer to the install path later. If the installer wants to know which version you want to install and offers the choice between debug and retail you should select the debug version and continue the installing process.

*If you install and use the debug version of the DirectX SDK your applications will use the debug builds of DirectX. That means you have a greater amount of error reports if something goes wrong. But it also means your programs will run a bit slower than they would using the retail libraries. If you want to make a final release build you should use the retail version of the DirectX SDK.*

Finally, when the installing is nearly complete, the DirectX installer should be smart enough to recognize the IDE on your system and ask you whether it should apply some changes to it to make it ready for DirectX. You should confirm that suggestion and let it do. There will be only the directory paths added to the IDE and a wizard to create DirectX applications if you need that one. We won't be using this wizard here.

Now the DirectX SDK should be successfully installed on your system. Please go to your Windows start menu and check the programs folder. It should now contain a new entry for the DirectX SDK. In this folder you will find the DirectX documentation or help file as well as the sample browser. Now start the sample browser and try to run the executables of some of the Direct3D samples. The Figure 2 shows a screenshot of the sample browser.



Figure 2: Screenshot of the DirectX SDK sample browser

### 2.1.3.2 Using the DirectX SDK Help

As I already mentioned the documentation of the DirectX SDK is rather good. Go to the Windows start menu and navigate to the Direct3D folder. Then click on the question mark icon linking to the documentation file. Now take yourself some time and have a look at its contents. It will be your second best friend for the next twelve weeks besides me. :-)

The documentation contains books for each DirectX component. Inside you will find a lot of introductions, tutorials, and the all important reference. If you are unsure about the name of a function or its parameter list you can take a look at the reference and just read the function's description.

### 2.1.3.3 Adding Directory Paths

Normally you should not be required to adjust your IDE settings manually anymore if you let the installer set the DirectX SDK paths to your IDE beforehand. If you refused to let the installer do it you have to set the paths by hand. The IDE needs to know two paths to be able to compile and link DirectX programs. The first path is the one to all header files you might need (c:\dx9sdk\include) and the second path is the one to the directory with all the libraries of DirectX you might need for your projects (c:\dx9sdk\lib).

But always make sure that the DirectX paths appear on the top most position in the list of path. The reason for that is straightforward. The compiler will look at this list from the top to the bottom. If the DirectX paths appear at the end of the list chances are that the IDE is using other DirectX libraries and include files from other paths because the IDE itself might already contain its own version of the DirectX SDK. But normally that is not the latest one so you have to ensure that your own version is used instead.

*To set the paths manually in Visual Studio .NET open the "Options" dialog from the "Extras" menu and select the folder "Projects" and the option "VC++ Directories". There you can set paths for libraries as well as for include directories. For other IDE's refer to your user manual to see how to add paths manually.*

### 2.1.3.4 Headers and Libraries

Given a correct and working installation of the latest DirectX SDK you are now ready to rock and to build your own programs using DirectX. You can set up a project just like you would normally do. You can choose the good old plain Win32 application project, a dynamic link library project, or a static library project. Just do like you are used to do.

## 2.2 Bumping into Direct3D

There are several unique ways to start introducing a new topic to a reader. For example you could just start showing little concepts about the topic you are going to explain, adding part after part and finally some time in the future you get to the point where all those pieces come together somehow. If the reader has not already forgotten half of it.

It don't think this is a good approach so I would like to take the head start approach to the Direct3D topic. I don't just want to explain this and that but I want to provide you some kind of "Direct3D hello world" sample. That is the smallest possible version of cranking up Direct3D and make it do something. Don't mistake this approach for giving you a cut down version of code that is only able to run stable on some machine. What I'm going to do here is a small but stable version on current graphics adapters.

The next chapter will deal with some design related issues and bring those sources in a better encapsulation. But the code of the functions will remain the same for most parts.

## 2.2.1 Initializing Direct3D

Actually initializing Direct3D is pretty easy and takes no more than filling one structure and calling two functions. The following two headings will show you how to do this in a nutshell. After that we will implement a function that does exactly those three things but we will add more runtime security by checking the capabilities of the graphics adapter. But let us proceed step by step. You take point.

### 2.2.1.1 About D3DPRESENT\_PARAMETERS

In Direct3D you will find a structure called `D3DPRESENT_PARAMETERS`. This is the structure you need to fill with values about how you want to run Direct3D, which display mode to use, which resolution, and the like. During the initialization call of Direct3D you only need to provide this structure and Direct3D knows with which settings you want it to run.

#### ***Excursion: Back Buffers and Render Targets***

*In the good old days of ancient graphic adapters you would normally set the color of each pixel directly in the so called front buffer. This front buffer is nothing else then just a memory area in the VRAM which is more or less directly connected to the output on the screen.*

*But that meant you have to set all the pixel of one frame and then wait for the frame to be rendered and sit idle before you can set the next pixels. This lead to the idea to use two of such memory areas. While the graphics adapter is setting the pixels from frame you can already render the next frame into the second memory area. If both operations are done you just switch the pointers to both areas with each other.*

*The memory area that is presented to the screen is called front buffer while the second memory area is called back buffer. Note that the a swap between the buffers will also swap the names so the back buffer is always referring to the area you can render into. Such a combination of at least two buffers is called a swap chain.*

*With the advance in graphics hardware it was then not only possible to render to certain memory areas in the VRAM only. Nowadays you can also render to textures for example. So finally there is an abstract object called **render target** which contains a back buffer as attribute. Actually you can now only render to a render target and the render target will provide the back buffer for the graphics adapter.*

*The swap chain is therefore also a type of a render target and you can access the back buffer of the swap chain and activate this for rendering. In fact if you start Direct3D there is an implicit swap chain created that doesn't require you to create it by yourself. Later on I will show you how to access that one.*

*Note that nowadays you should not access the content of the front buffer. Direct3D will typically prevent you from doing this anyway.*

Now I will show you this structure and explain each field of it briefly. Unfortunately you will run into some buzz words you don't know about yet. But don't worry if you don't fully understand the meaning of everything. It all comes together one by one when we start initializing Direct3D with our own function. Then I will explain all those things and fill in your gaps. Here we go:

```
typedef struct _D3DPRESENT_PARAMETERS
{
    UINT                BackBufferWidth, BackBufferHeight;
    D3DFORMAT           BackBufferFormat;
    UINT                BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD               MultiSampleQuality;
    D3DSWAPEFFECT       SwapEffect;
    HWND                hDeviceWindow;
    BOOL                Windowed;
    BOOL                EnableAutoDepthStencil;
    D3DFORMAT           AutoDepthStencilFormat;
    DWORD               Flags;
    UINT                FullScreen_RefreshRateInHz;
    UINT                PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

The structure's fields in detail:

- **BackBufferWidth, BackBufferHeight**  
These values represent the width and the height of the render target (that is the image that is rendered). In full-screen mode they have to correspond to a valid format the graphics adapter supports. In windowed mode you can supply 0 as value which would let Direct3D take the size of the client area from the window it is couple to.
- **BackBufferFormat**  
This is the pixel format used for the render target, see the enumerated Direct3D type `D3DFORMAT`. In windowed mode you can set the format to `D3DFMT_UNKNOWN` to use the current desktop display mode.
- **BackBufferCount**  
This is the number of back buffers. You can use the values 0, 1, 2, and 3 here. Other values would make the initialization fail. The value of 0 is treated as 1 because there has to be one back buffer at least. The back buffers created on the initialization call can be accessed as the so called swap chain later on.
- **MultiSampleType**  
In Direct3D full-screen antialiasing is done by multisampling. Antialiasing means blurring the edges of polygons rendered to avoid the effect of noticeably hard-shaped pixel steps. Multisampling works by merging neighboring pixels together to blur their colors. Use a member of the Direct3D enumerated type `D3DMULTISAMPLE_TYPE` here. Note that the swap effect must be set to `D3DSWAPEFFECT_DISCARD` to enable multisampling at all.
- **MultiSampleQuality**  
The quality level of multisampling. Please refer to the Direct3D reference to see how to check for valid multisampling type and quality on your graphics adapter.
- **SwapEffect**  
Here you can control the type of switch between the front buffer and the back buffer. Use a member of the `D3DSWAPEFFECT`

enumerated type here. We will only use the value `D3DSWAPEFFECT_DISCARD` here. That is the fastest one but it does not guarantee that the old contents of a back buffer are maintained when the buffers are swapped. Using any other setting here would lead to internal overheads and copying back buffers that would slow you down.

- `hDeviceWindow`  
This is the handle to the window that should be used to render into. Note that a change of size in the window is not automatically taken care of. We have to handle the resizing event by ourselves as you will see later.
- `Windowed`  
Set to `TRUE` if you want to stay in windowed mode. Otherwise set to `FALSE`.
- `EnableAutoDepthStencil`  
If this is set to `TRUE` Direct3D will automatically create a depth stencil surface and recreate it as well if you reset Direct3D.
- `AutoDepthStencilFormat`  
If you enabled the automatic creation of a depth stencil surface you need to set this to a valid bit depth format of the type `D3DFORMAT`. I will explain this to you in detail in a minute.
- `Flags`  
We don't need to use any flags here so just ignore that one.
- `FullScreen_RefreshRateInHz`  
This is the refresh rate you want to run the monitor with. For windowed mode this must be set to 0. In full-screen mode you can select a valid value. Again, hang on a minute or two and I will show you.
- `PresentationInterval`  
Here you can set how Direct3D should take care of the monitor's vertical synchronization rate. The default value which equals to 0 but

in Direct3D is called `D3DPRESENT_INTERVAL_DEFAULT` will stay in sync with the monitor refresh rate. So even if your graphics adapter could be faster it will wait for the vertical retrace period to begin before rendering. This would prevent you from suffering tearing effects.

If you want to render without regarding the sync rate use the value called `D3DPRESENT_INTERVAL_IMMEDIATE` here. This would enable you to render an image immediately no matter where the vertical beam of the monitor is located in this very moment. But you could suffer optical artifacts.

Well, that looks like a whole bunch of things you need to know about, now does it? But it is not as bad as it sounds. Just look at this structure and think of it as a collection of settings to adjust Direct3D to your needs. You will see how to make typical settings serving all your needs most of the time. But before we start talking about the initialization I want to talk to you about the objects you need to create to make Direct3D run.

### 2.2.1.2 The Direct3D Object and its Device

The first object coming across your way in Direct3D is its main object. The interface of this object is called `IDirect3D9`. To create pointers to that kind of object without using the `*` sign Microsoft defined the following typedefs for the main object:

```
typedef struct IDirect3D9 *LPDIRECT3D9, *PDIRECT3D9;
```

// thus writing "**IDirect3D9\* pD3D**;" equals "**LPDIRECT3D9 pD3D**;" e.g.

Interestingly you don't need the main object very often. You can query a few capabilities of the graphics adapter using this object and you can create the Direct3D device object. That is the real work horse doing all the things related to rendering graphics. Its interface is called `IDirect3DDevice9` and again Microsoft added the following type definitions:

```
typedef struct IDirect3DDevice9 *LPDIRECT3DDEVICE9, *PDIRECT3DDEVICE9;
```

But first you might want to know how you can create Direct3D's main object. In Direct3D you will normally only deal with objects. That means all its functionality can be accessed by calling member functions of certain objects. But here is one of the rare situations where you have to call a stand alone function because you have no single object yet.

*The object oriented design of Direct3D is opposed to the functional programming in OpenGL where you don't have objects at all.*

It is not possible to create a Direct3D object by just calling the new operator for example. You have to use the function that is meant to create such an object for you and that is this one here:

```
IDirect3D9 *WINAPI Direct3DCreate9( UINT SDKVersion );
```

As parameter you have to supply D3D\_SDK\_VERSION there is no other choice than that. This value is used as check if you are using the same header files where this value is defined as that ones used to build the Direct3D library. If the function succeeds which is very likely by the way then you have a pointer to the main object of Direct3D.

You can then use this pointer to enumerate capabilities of the current graphics hardware in the system. And of course you can use this object to create the workhorse of Direct3D – namely its device:

```
HRESULT IDirect3D9::CreateDevice(
    UINT                Adapter,
    D3DDEVTYPE          DeviceType,
    HWND                hFocusWindow,
    DWORD               BehaviorFlags,
    D3DPRESENT_PARAMETERS * pPresentationParameters,
    IDirect3DDevice9** ppReturnedDeviceInterface );
```

The parameters of the function in detail:

- Adapter  
If you have several graphic adapters installed on your system you need to select one of them by its ordinal number. I suppose that you have only one installed or that the best one is used as primary

adapter in Windows. The value of `D3DADAPTER_DEFAULT` will select the primary adapter.

- DeviceType

Direct3D knows three types of devices as enumerated in `D3DDEVTYPE`. The most important is `D3DDEVTYPE_HAL` which basically means that as much as possible is done in hardware. If you hardware cannot comply certain functions will fail if they are not supported.

The second type is `D3DDEVTYPE_REF` which is the reference rasterizer. This device is a complete software implementation so you can use all Direct3D features even if you hardware does not support them. But be careful. Its awesomely slow so should only use this as a testing environment and not in a final release product.

Finally you could write your own software device and plug it into Direct3D but I don't consider this an option here. After all you came here to see how the Direct3D interfaces work.

- hFocusWindow

The window who's focus is connected to the Direct3D device.

- BehaviorFlags

Here you can provide several behavior flags enumerated in the `D3DCREATE` type. The most important will control how the vertex data is processed:

- `D3DCREATE_HARDWARE_VERTEXPROCESSING`
- `D3DCREATE_SOFTWARE_VERTEXPROCESSING`
- `D3DCREATE_MIXED_VERTEXPROCESSING`

The first one states that the hardware should transform the data. This is the fastest one but it just fails if the hardware does not support several operations like vertex tweening like we will see in a later chapter. The second one lets Direct3D do the transformations in software. This is slow but guaranteed to work. Finally, the third state is actually both cases. You can switch during runtime if you want to use hardware or software processing, thus combining speed where available with stability where needed. I will show you how to switch the processing type when we are talking about vertex tweening.

- `pPresentationParameters`  
We just talked about the presentation parameters in detail. Here you hand over the readily filled structure telling Direct3D the settings you want to have.
- `ppReturnedDeviceInterface`  
This is a reference to a pointer where the new object should be saved.

Okay, now you know all you need to know to actually initialize Direct3D. But hang on for a moment. Before we come to that we will add some comfort to our upcoming initialization routine by ...

*Most Direct3D function use HRESULT as return value to indicate a success or failure of the call just like WinAPI functions do. You can check such a value using the following WinAPI defines:*

```
if (FAILED( hr )) { /*react to error*/ }
```

```
if (SUCCEEDED( hr )) { /*proceed as intended*/ }
```

*In case of failure you can check for certain error values. Refer to the DirectX reference to check for the errors a specific function can generate.*

### 2.2.1.3 Defining an Enumeration for the Resolution

It is always a good idea to keep several standard settings handy and don't let the user freely decide for example about the values for the width and the height of a full-screen mode. Most graphics adapters don't like weird custom formats at all. So here is what we define as resolutions the user can request from our initialization.

```
enum SCREENRESOLUTION
{
    RES_CURRENT,
    RES_640x480,
    RES_800x600,
    RES_1024x768,
```

```
RES_1280x960,  
RES_1280x1024,  
RES_1600x1200  
};
```

Note that choosing a certain resolution does not mean the user will get it. If the graphics hardware does not support a certain mode e.g. the most biggest ones then we need whether to fail the initialization what would suck or to switch the next available one. The latter is what we are going to do.

### 2.2.1.4 Tying up the Function

And here we are, finally getting down to code. The following function is doing a smart initialization of Direct3D. You have come along way and learned a lot of things before reaching this point. So just have a look at the function, first. Then let us talk about it later. I just want to add here that the functions takes three parameters. First of all the handle of the window used, second the decision if you want to crank up in windowed mode, and finally the number of bits you want to have for the stencil buffer.

The function does also hardwire the pixel format of the back buffer to the Direct3D format D3DFMT\_X8R8G8B8. That means it is using color depth of 32 bits with eight bits for the red, the green, and the blue component each thus leaving 8 bits unused. This is what the X is standing for. In formats for textures you can use those eight bits to store alpha information about transparency effects. But alpha blending is a topic of its own later. There are also a lot of other formats defined by Direct3D like different 16 bit color depth formats and the like. But nowadays each graphics adapter in use by video game players should be able to do 32 bits of color – even older models.

But I'm talking too much, right? Here you are:

```
bool Initialize( HWND hWnd, bool bWindowed, unsigned int StencilBits )  
{  
    D3DPRESENT_PARAMETERS PP;  
  
    memset( &PP, 0, sizeof( D3DPRESENT_PARAMETERS ) );  
  
    if ( !(g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)) )
```

```

    {
        return false;
    }

    // find best fitting depth stencil format
    D3DFORMAT fmtDepthStencil = GetBestDepthStencil( StencilBits);

    if ( fmtDepthStencil == D3DFMT_UNKNOWN )
    {
        return false;
    }

    // for fullscreen get max resolution available
    if ( !bWindowed )
    {
        SetBestResolution(RES_1600x1200,&(PP.FullScreen_RefreshRateInHz),PP);
    }
    // for windowed mode get current desktop resolution
    else
    {
        SetBestResolution( RES_CURRENT, NULL, PP );
    }

    // misc parameters
    PP.Windowed           = bWindowed;
    PP.hDeviceWindow      = hWnd;
    PP.SwapEffect         = D3DSWAPEFFECT_DISCARD;
    PP.MultiSampleType    = D3DMULTISAMPLE_NONE;
    PP.BackBufferCount    = 1;

    // depth- and stencil- buffer
    PP.EnableAutoDepthStencil = true;
    PP.AutoDepthStencilFormat = fmtDepthStencil;
    PP.BackBufferFormat       = D3DFMT_X8R8G8B8;

    if ( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT,
                                     D3DDEVTYPE_HAL, hWnd,
                                     D3DCREATE_MIXED_VERTEXPROCESSING,
                                     &PP, &g_pDeviceD3D ) ) )
    {
        return false;
    }

    return true;
} // Initialize
/*-----*/

```

Actually there is nothing going on you don't know what it is all about, right?  
 Except maybe for two cute, little helper functions named

GetBestDepthStencil() and SetBestResolution(), respectively. Now let's have a look at these helper functions, shall we?

### ***Excursion: The Depth Buffer and the Stencil Buffer***

*While rendering polygons to the screen they get projected from 3D to 2d and become pixels. The problem is that you would have to render the polygon sorted by their depth in 3D space starting with the most distant ones. But you simply cannot sort arbitrary polygons perfectly if they are overlapping each other.*

*The solution is to save the depth value of each pixel on a separate image using the same resolution as the back buffer and which is called depth buffer. Prior to rendering a pixel the hardware will check its depth buffer if the new pixel is closer to the viewer than the pixel that has already potentially been rendered to the same position on screen.*

*The stencil buffer is another addition buffer that can take additional values per pixel to create special effects. You could for example mask a part of the screen by setting the mask into the stencil buffer.*

*Form a technical point of view the stencil buffer is realized as part of the depth buffer. Hence you refer to both of them as depth stencil surface normally.*

## **2.2.2 Helper Functions**

In the initialization function you have already seen the call of two helper functions named GetBestDepthStencil() and SetBestResolution(). But there is a third one called CheckResolutionAvailability() which is used in one of the other functions. So we will start looking at this one.

### **2.2.2.1 About D3DDISPLAYMODE**

But beforehand I want you to show the following structure defined by Direct3D. It is used to describe a so called display mode. A display mode

is a combination of the resolution, the color depth format, and the refresh rate compatible with the other settings. It looks like this:

```
typedef struct _D3DDISPLAYMODE
{
    UINT          Width;
    UINT          Height;
    UINT          RefreshRate;
    D3DFORMAT     Format;
} D3DDISPLAYMODE;
```

The structure's fields in detail:

- Width, Height  
The values for the width and height of this display mode, what else? :-)
- RefreshRate  
The refresh rate for which this resolution and color depth is available.  
.
- Format  
A member of the enumerated type D3DFORMAT. While examining the graphics adapter you will normally find several display modes using the very same resolution and refresh rate but differ in their formats only. This could be a 32 bit or a 16 bit format for example, stating how many bits are available for the components red, green, and blue each.

That is not too difficult to get but I guess you wonder how you can ask your graphics adapter what display modes he has at the stock for you? No problem, just read on.

### 2.2.2.2 Checking Resolution Availability

Well, above we created an own enumerated type for the screen resolutions the user can request e.g. RES\_1024x768. But there is one catch. Not all graphics adapters support all possible screen resolutions. Only the latest

ones would be able to cope with the biggest resolutions like 1280x960 and greater.

So we need to check at run time if the graphics adapter supports a certain resolution the user asked for. You could for sure hardwire the resolution as well. But if you would use a too small resolution you would laugh at the guys with better hardware. And if you would use a too big resolution you would screw with the guys using older hardware. So it is better to keep that flexibility and don't suppose everybody on the world uses exactly the same graphics adapter as you do and your program will run on their machines because it does on yours.

Luckily the Direct3D main object supports such queries to the graphics hardware before you are actually creating a Direct3D device object. The following member function of the Direct3D main object lets you ask the graphics adapter how many different display modes it supports and answers you by returning the number of available display mode formats:

```
UINT IDirect3D9::GetAdapterModeCount( UINT Adapter, D3DFORMAT Format );
```

The parameters of the function in detail:

- Adapter  
The ordinal number of the adapter. We only use D3DADAPTER\_DEFAULT.
- Format  
Supply the format you want to use (e.g. D3DFMT\_X8R8G8B8 in our case). If you want to use different formats you need to make a single call to this function for each format.

Now you only know how many different display modes the graphics adapter supports for the color depth format you are using. But you don't know how those display modes are looking at all. Actually we are here to ask the adapter if it supports a certain resolution together with the format D3DFMT\_X8R8G8B8. So the following function lets you extract the information about one of the available display modes for a given format:

```
HRESULT IDirect3D9::EnumAdapterModes( UINT Adapter, D3DFORMAT Format,
```

```
UINT Mode,  
D3DDISPLAYMODE* pMode);
```

The parameters of the function in detail:

- Adapter  
The ordinal number of the adapter. We only use `D3DADAPTER_DEFAULT`.
- Format  
Supply the format you want to use (e.g. `D3DFMT_X8R8G8B8` in our case).
- Mode  
This is the number of the display mode you want to have information about. This number can be any value from 0 to n-1 where n is the number of modes available as returned by `IDirect3D9::GetAdapterModeCount`.
- pMode  
A reference to the Direct3D structure `D3DDISPLAYMODE` that is filled with the information about the mode.

The combination of the last two functions I just showed you will enable you to write a loop to examine each display mode that is available for the color format we are using. This way we can build a list of all display modes available for the format we are using. And this is what we're going to do in a moment. But then there is also the question of the refresh rate. Personally I don't want to let the user decide about the refresh rate he wants to use. Flexibility is a nice thing but don't bother the user by forcing him to decide this and that and a lot of other things while he is simply not interested in those things normally.

So I decided to go the easy way and just select display formats into our list of suitable modes that are using the same refresh rate as the current Windows desktop mode. Another positive aspect of this method is that the monitor as well as the graphics adapter are already known to support that

refresh rate. So here is the function that lets you get the current desktop settings as display mode structure:

```
HRESULT IDirect3D9::GetAdapterDisplayMode( UINT Adapter,
                                           D3DDISPLAYMODE* pMode);
```

The parameters of the function in detail:

- Adapter  
The ordinal number of the adapter. We only use D3DADAPTER\_DEFAULT.
- pMode  
A reference to the Direct3D structure D3DDISPLAYMODE that is filled with the information about the mode.

Now you are set and ready to go. Here is the plan what the next function has to do:

1. If running the first time build a list of display modes for D3DFMT\_X8R8G8B8
2. Check if the supplied resolution is valid for one display mode in the list
3. Return true if the resolution can be done, false otherwise.

Well, its no magic as you can see. Just take a look at the function. Now that you know what it has to do and which functions Direct3D offers to help us out it should be no problem for you to understand the code.

```
bool CheckResolutionAvailability( unsigned int uiWidth, unsigned int uiHeight )
{
    static D3DDISPLAYMODE sDispModes[1024];
    static unsigned int   suiNum   = 0;
    static unsigned int   suiNumKept = 0;

    if ( suiNumKept == 0 )
    {
        D3DDISPLAYMODE dspmd;

        // get current refresh rate
        g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &dspmd );

        suiNum = g_pD3D->GetAdapterModeCount( D3DADAPTER_DEFAULT,
```

```
D3DFMT_X8R8G8B8 );
```

```
memset( sDispModes, 0, sizeof( D3DDISPLAYMODE ) * 1024 );

// enumerate all available displaymodes
for ( unsigned int i=0; i<suiNum; i++ )
{
    g_pD3D->EnumAdapterModes( D3DADAPTER_DEFAULT,
                             D3DFMT_X8R8G8B8,
                             i, &sDispModes[ suiNumKept ] );

    // keep modes using current refresh rate only
    if ( sDispModes[ suiNumKept ].RefreshRate == dspmd.RefreshRate )
    {
        suiNumKept++;
    }
}

// now check if there is a suited one
for ( unsigned int j=0; j<suiNumKept; j++ )
{
    if ( ( sDispModes[ j ].Width == uiWidth ) &&
         ( sDispModes[ j ].Height == uiHeight ) )
    {
        return true;
    }
}

// requested resolution not found
return false;
}
/*-----*/
```

Not too bad, isn't it? You can now use this function to check whether the graphics adapter supports a resolution you request along with the currently active monitor refresh rate and the 32 bit color format D3DFMT\_X8R8G8B8. You can also enrich the enumerations to check for different refresh rates and different color formats as well. But I don't want to make this an enumeration hell for you. This is enough flexibility to work with.

### 2.2.2.3 Choosing the best Resolution

That out of the way you can go on and write the function SetBestResolution() now. As input the function, that was already used in the

initialization function above, wants to have a resolution from our own enumerated type SCREENRESOLUTION, optionally a pointer to a parameter to store the currently active refresh rate, and finally a reference to a D3DPRESENT\_PARAMETERS structure where it should store the finally selected resolution into.

That is because this function will not only check if the requested resolution is available. If that is not the case it will examine the next lower resolution until a suited one is found. The function CheckResolutionAvailability() is used as helper routine to decided whether the requested resolution is possible or not.

```
void SetBestResolution( SCREENRESOLUTION res, unsigned int *pRefreshRate,
                      D3DPRESENT_PARAMETERS &PP )
{
    // get current refresh rates well
    if ( pRefreshRate )
    {
        D3DDISPLAYMODE dspmd;
        g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &dspmd );
        (*pRefreshRate) = dspmd.RefreshRate;
    }

    // check resolutions
    if ( res == RES_1600x1200 )
    {
        if ( CheckResolutionAvailability( 1600, 1200 ) )
        {
            PP.BackBufferWidth = 1600;
            PP.BackBufferHeight = 1200;
            return;
        }
        else res = RES_1280x1024;
    }

    if ( res == RES_1280x1024 )
    {
        if ( CheckResolutionAvailability( 1280, 1024 ) )
        {
            PP.BackBufferWidth = 1280;
            PP.BackBufferHeight = 1024;
            return;
        }
        else res = RES_1280x960;
    }
}
```

```

if ( res == RES_1280x960 )
{
    if ( CheckResolutionAvailability( 1280, 960 ) )
    {
        PP.BackBufferWidth = 1280;
        PP.BackBufferHeight = 960;
        return;
    }
    else res = RES_1024x768;
}

if ( res == RES_1024x768 )
{
    if ( CheckResolutionAvailability( 1024, 768 ) )
    {
        PP.BackBufferWidth = 1024;
        PP.BackBufferHeight = 768;
        return;
    }
    else res = RES_800x600;
}

if ( res == RES_800x600 )
{
    if ( CheckResolutionAvailability( 800, 600 ) )
    {
        PP.BackBufferWidth = 800;
        PP.BackBufferHeight = 600;
        return;
    }
    else res = RES_640x480;
}

if ( res == RES_640x480 )
{
    if ( CheckResolutionAvailability( 640, 480 ) )
    {
        PP.BackBufferWidth = 640;
        PP.BackBufferHeight = 480;
        return;
    }
}

// take current resolution
PP.BackBufferWidth = GetSystemMetrics(SM_CXSCREEN);
PP.BackBufferHeight = GetSystemMetrics(SM_CYSCREEN);
return;
}
/*-----*/

```

As you can see this function is just a big chain of if's. If the requested format is not available the next lower one is examined. Finally, if nothing seems to be possible that current desktop resolution is taken. That will work for sure as it apparently does at the moment for Windows. The WinAPI function `GetSystemMetrics()` can deliver you all kind of values. In this case the X and Y resolution of the screen but you could of course also use `IDirect3D9::GetAdapterDisplayMode` to get those values.

And that is all one can say about checking for graphics adapter for available modes and selecting the one that is fitting your needs the best.

### 2.2.2.4 Choosing the best Depth/Stencil Format

The last helper function is the one doing again some checking of availabilities. But in this case we are not in search for a good back buffer format but for a format we can use for the depth stencil surface. Because depth and stencil buffers are nowadays implemented in hardware we have to ask the hardware what kinds of formats it supports for depth and stencil surfaces.

Direct3D lets you evaluate the available formats by the following function:

```
HRESULT IDirect3D9::CheckDeviceFormat( UINT Adapter,
                                       D3DDEVTYPE DeviceType,
                                       D3DFORMAT AdapterFormat,
                                       DWORD Usage,
                                       D3DRESOURCETYPE RType,
                                       D3DFORMAT CheckFormat );
```

The parameters of the function in detail:

- Adapter  
The ordinal number of the adapter. We only use `D3DADAPTER_DEFAULT`.
- DeviceType  
`D3DDEVTYPE_HAL`, `D3DDEVTYPE_REF`, or `D3DDEVTYPE_SW`.

- AdapterFormat  
Set this to the display mode's color format you are using (e.g. D3DFMT\_X8R8G8B8 in our case)
- Usage  
Set this to a valid member of the D3DUSAGE enumerated type. But be cautious as not all members are valid for a use with this call. We will stick to use it with this value D3DUSAGE\_DEPTHSTENCIL indicating that we want to check a format and want to know whether it can be used on this adapter for a depth stencil surface or not.
- RType  
Here you have to set the type of resource that you want to create with the format in question. Check the D3DRESOURCETYPE enumerated type for possible values. We will use D3DRTYPE\_SURFACE here because a depth stencil surface is of the surface type in Direct3D. A surface is basically nothing more than a memory area for a 2d image.
- CheckFormat  
Finally, supply the function with the format you want to check for availability.

If this functions succeeds and returns a value of D3D\_OK the format is valid and available for the intended usage on the given graphics adapter. Because we need to call this function now several times with mostly the same settings I wrote this macro to make it easier to use and to read in the code:

```
#define defCheckFormat(f) g_pD3D->CheckDeviceFormat(
    D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL,
    D3DFMT_X8R8G8B8,
    D3DUSAGE_DEPTHSTENCIL,
    D3DRTYPE_SURFACE, f)
```

But before we can utilize this macro there is one thing left you should know. I was talking about the enumerated type D3DFORMAT on and on in several places. Actually the values from this type can be used for different

things like back buffer format, texture format, and depth stencil format. So you can already see that all of these objects contain a surface object internally.

But most of the formats are intended to use them with certain objects only. As well as there are formats meant for back buffers only there are formats to use with textures as well as formats to use for a depth stencil surface. So the following formats are the ones you could check the hardware support for because they are the only ones you can create a depth stencil surface with:

- D3DFMT\_D24S8
- D3DFMT\_D24X4S4
- D3DFMT\_D15S1
- D3DFMT\_D32
- D3DFMT\_D24X8
- D3DFMT\_D16

The number following the letter D stands for the bits reserved for depth values in the format, the number following the X means unused bits, and the number following the S stands for the bits used for the stencil buffer. As you can see the depth stencil buffer is always using whether 32 bit or 16 bit. The more bits you have at your disposal the better the accuracy of the buffer.

Depending on the number of stencil bits requested by the user the following function will choose the depth stencil format most appropriate:

```
D3DFORMAT GetBestDepthStencil( unsigned int StencilBits )
{
    if ( ( StencilBits > 1 ) && ( StencilBits < 4 ) ) StencilBits = 1;
    if ( ( StencilBits > 4 ) && ( StencilBits < 8 ) ) StencilBits = 4;
    if ( StencilBits > 8 ) StencilBits = 8;

    if ( StencilBits == 8 )
```

```

    {
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D24S8 ) ) )
            return D3DFMT_D24S8;
        else StencilBits = 4;
    }

    if ( StencilBits == 4 )
    {
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D24X4S4 ) ) )
            return D3DFMT_D24X4S4;
        else StencilBits = 1;
    }

    if ( StencilBits == 1 )
    {
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D15S1 ) ) )
            return D3DFMT_D15S1;
        else StencilBits = 0;
    }

    if ( StencilBits == 0 )
    {
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D32 ) ) )
            return D3DFMT_D32;
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D24X8 ) ) )
            return D3DFMT_D24X8;
        if ( SUCCEEDED( defCheckFormat( D3DFMT_D16 ) ) )
            return D3DFMT_D16;
    }

    return D3DFMT_UNKNOWN;
}
/*-----*/

```

And now for the good news. That is all you need to know and to do in order to initialize Direct3D in a rather secure method. You would not let the program crash by hardwiring resolutions and bit formats a given hardware cannot support.

## 2.3 Writing a Demo Application

With these four elegant functions at our stock there is nothing keeping us away from writing a simple Windows application that is cranking up Direct3D and lets us decided whether to use windowed mode or full-screen

mode. While running the application will do nothing else than just clearing the screen or the client area of the render window with a cycling red color.

## 2.3.1 Create a Window

Beaming a Windows window to your desktop is a task that I suppose you are already comfortable with. So there is no need to comment the next few lines of code that will simply do this. Note that the function returns the handle of the newly created window to the caller.

```
HWND CreateWindowsStuff( HINSTANCE hInst )
{
    WNDCLASSEX wndclass;
    wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC | CS_DBLCLKS;

    wndclass.cbSize           = sizeof(wndclass);
    wndclass.lpfnWndProc      = MsgProc;
    wndclass.cbClsExtra       = 0;
    wndclass.cbWndExtra       = 0;
    wndclass.hInstance        = hInst;
    wndclass.hIcon            = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor          = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground    = (HBRUSH)(COLOR_WINDOW);
    wndclass.lpszMenuName     = NULL;
    wndclass.lpszClassName    = TEXT("SampleClass");
    wndclass.hIconSm          = LoadIcon(NULL, IDI_APPLICATION);

    if ( RegisterClassEx( &wndclass ) == 0 ) return 0;

    return CreateWindowEx( NULL, TEXT("SampleClass"), "Week 1 Demo",
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        GetSystemMetrics(SM_CXSCREEN)/2 - 250,

        GetSystemMetrics(SM_CYSCREEN)/2-187, 500, 375, NULL, NULL, hInst, NULL );
}
/*-----*/
```

## 2.3.2 Proceeding Windows Messages

For the window class used for our window the above code defines the `MsgProc()` function as callback procedure to handle incoming Windows messages. Again, I suppose you know how to handle the Windows

message stuff. In case you need a refresh of how to deal with the incoming messages refer to the MSDN documentation or just take a look at this function:

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam,
                      LPARAM lParam )
{
    switch( msg )
    {
        // key was pressed
        case WM_KEYDOWN:
        {
            switch (wParam)
            {
                case VK_ESCAPE:
                {
                    PostMessage(hWnd, WM_CLOSE, 0, 0);
                    return 0;
                }
                default: break;
            }
        } break;

        case WM_DESTROY:
        {
            PostQuitMessage( 0 );
            g_bDone = true;
            return 1;
        }
        default: break;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
/*-----*/
```

From the incoming messages this function only proceeds the key hit event for the Escape key as well as the event of the window being destroyed.

### 2.3.3 The WinMain() Function

And here it comes, the well known famous entry point to the Windows application, the WinMain() function. I want to keep it easy to use so it pops up with a message box on startup asking you if you would like it to run in windowed mode or in full-screen mode. After that it will just create the

window needed for Direct3D, call the Direct3D initialization function we wrote over the course of this chapter, and then enter a main loop using the Direct3D device to clear the window's color.

But first, I want to introduce the following global variables to you:

```
bool                g_bDone                = false;
IDirect3DDevice9*  g_pDeviceD3D           = NULL;
IDirect3D9*        g_pD3D                = NULL;
```

Don't bother about using nasty global variables. In the next chapter they will become history. But for this simple demonstration purpose they just hit the match. Now dig yourself through the function. We will talk about the things going on in the main loop afterwards.

```
int WINAPI WinMain( HINSTANCE hInst, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    bool bWindowed = ( MessageBox( NULL, "Start in windowed mode?", "Question",
                                   MB_YESNO | MB_ICONQUESTION )==IDYES);

    HWND hWnd = CreateWindowsStuff( hInst );

    if ( !hWnd || !Initialize( hWnd, bWindowed, 0 ) )
    {
        if ( hWnd )
        {
            MessageBox( NULL, "Initialization of Direct3D failed!",
                        "Warning", MB_OK | MB_ICONERROR );
        }
        g_bDone = true;
    }

    MSG msg;
    float fR=1.0f;

    while ( !g_bDone )
    {
        while ( PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) )
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        // waving color value
        fR = sin( float(GetTickCount())/1000.0f);
        if ( fR < 0.0f ) fR *= -1.0f;
    }
}
```

```

        // clear render target and depth buffer
        g_pDeviceD3D->Clear( 0, 0, D3DCLEAR_TARGET |
                            D3DCLEAR_ZBUFFER,
                            D3DCOLOR_COLORVALUE( fR, 0, 0, 1), 1.0f, 0);

        g_pDeviceD3D->Present( 0, 0, 0, 0);
    }

    // release direct3d objects
    if ( g_pDeviceD3D ) g_pDeviceD3D->Release();
    if ( g_pD3D ) g_pD3D->Release();

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

```

Note that you have to clean up the Direct3D objects you have used. In order to do this just call the `IUnknown::Release` method from those objects. This would lead us deeply into the Component Object Model topic (COM) on which Direct3D is based. Basically `IUnknown` is a pretty simple interface each COM object inherits from. So do all the DirectX objects because they are build upon COM. This basic interface is only meant to keep track of instances created from a certain object.

If you create such an object by requesting it from another object (like you did with the Direct3D device from the Direct3D main object) an internal reference counter is increased. By releasing such an object this reference counter is decreased. If it reaches 0 it should mean that there are no more programs referencing to this object and it can be destroyed then.

### 2.3.3.1 Clearing the Direct3D device

A Direct3D device object has three buffers inside of it among some others. The back buffer, the depth buffer and the stencil buffer. Once you rendered something into those buffers they are filled with data. If you need those buffers without any kind of data except for default entries you have to clear them.

Normally that is at least needed each frame for the depth buffer. If you want to render a similar scene again from a slightly moved viewpoint the

depth values in the depth buffer would prevent lots of pixels from being drawn correctly. The render target itself would normally not require to be cleared if you know that you will render each single pixel of the back buffer anyway.

So here is the function to let you clear one or more of those three buffers:

```
HRESULT IDirect3D9::Clear( DWORD Count, const D3DRECT *pRects,  
                          DWORD Flags, D3DCOLOR Color,  
                          float Z, DWORD Stencil );
```

The parameters of the function in detail:

- Count  
Number of rectangles provided in the next parameter.
- pRects  
Array of D3DRECT structures containing rectangles you want to clear on the render target. Provide NULL if you want to clear the whole render target area.
- Flags  
Set this to a logical OR combination of the following flags indicating which of the three possible buffers should be cleared. Naturally you must use at least one flag:
  - D3DCLEAR\_TARGET (clear the render target's back buffer)
  - D3DCLEAR\_ZBUFFER (clear the depth buffer)
  - D3DCLEAR\_STENCIL (clear the stencil buffer)
- Color  
This is the color provided as 32 bit ARGB value which is filled to the render target if that should be cleared. You can use the following Direct3D macro D3DCOLOR\_COLORVALUE( r, g, b, a ) and set its parameters for RGBA to float values ranging from 0.0f to 1.0f.
- Z  
This is the value filled into the depth buffer is you want to clear it. Normally the depth values are calculated in the range [0.0f, 1.0f] where 0 means closest to the viewer and 1 means farthest away

from the view and equals an infinite distance. So normally you would clear the depth buffer to this value.

- Stencil

Similar like the last parameter this is the value to be filled into the stencil buffer if it is cleared. The parameter can range from 0 to  $2^n-1$  where n is the bit depth of the stencil buffer.

## 2.3.3.2 Presenting the Scene

Whatever you have done to the back buffer like rendering polygons or just clearing it has no effect on what you see on the screen as long as you don't swap the front buffer and the back buffer. This is done by using the following function from the Direct3D device object:

```
HRESULT IDirect3DDevice9::Present( CONST RECT *pSourceRect,  
                                  CONST RECT *pDestRect,  
                                  HWND hDestWindowOverride,  
                                  CONST RGNDATA *pDirtyRegion );
```

The parameters of the function in detail:

- `pSourceRect`  
Provide NULL if the swap effect is not `D3DSWAPEFFECT_COPY`. Otherwise this is the source rectangle for the presented scene. If NULL the entire source surface is used.
- `pDestRect`  
Provide NULL if the swap effect is not `D3DSWAPEFFECT_COPY`. Otherwise this is the target rectangle for the presented scene in client coordinates. If NULL the entire target surface is used.
- `hDestWindowOverride`  
Here you can name any window you want to use to display the scene. If not NULL then this overrides the window handle you used during the initialization process of the Direct3D device.
- `pDirtyRegion`  
Provide NULL if the swap effect is not `D3DSWAPEFFECT_COPY`. Otherwise this is a region in back buffer coordinates which must be updated at least.

As you can see a lot of things depend on the swap effect for this function. We will only use the `D3DSWAPEFFECT_DISCARD` value so most of the time you will call this function with a simple 0 for all of its parameters.

## 2.4 Summing Up

Congratulations, you survived the first chapter. Now you can open up your IDE if you don't have that already and take a look at the code that came along with this book. Compile and run it, and see if you can twist around some things. Changing the color of the clear call for example. Or use the debugger to walk through the initialization step by step.

But don't stop here. I advise you open the DirectX documentation as well and take a closer look at each function from Direct3D I used in this chapter. Take a look at all the enumerated types as well and read the descriptions of the various types and what they are meant for.

If you then feel comfortable with all you have seen so far I will see you in the next chapter. Oh that reminds me to add a word on knowing the Direct3D initialization by heart. Most guys new to Direct3D come to me saying then can never ever keep this whole initialization stuff in their memory and ask me if they need to. Now that depends on the eye of the beholder. Personally I don't think you are better in Direct3D programming if you know each function call, its parameter list, and each enumerated type from memory. There is nothing wrong in opening up the DirectX documentation and use its reference. So the answer is no, you don't have to memorize all this stuff.

But you should know where to look it up and you should be able to do it fast. For the next few weeks make it a habit of yours to open the DirectX reference each time you start your IDE to screw around with some DirectX source code. You will soon come to realize that you will automatically memorize all the things you use quite often. Your brain is in fact intelligent enough to do this on its own. :-)