Game Programming Genesis
Part IV : Introduction to DirectX

by Joseph "Ironblayde" Farrell

Introduction

Ahh, it's a good day. Know why? Because today we'll be taking a first look at the awesome creation that is DirectX. It's several times faster than Windows GDI, usable from different languages and different platforms, and supports everything from plotting pixels to advanced 3D effects, from playing simple sound effects to streaming digital music, from handling the keyboard to using Force Feedback controllers. It even gives you a hand with networking, and installing the run-time libraries on the user's machine. Of course, it's a huge subject and it would take several books to cover all of it, not to mention about a hundred times more knowledge than I have to give you, but I can get you started. :)

For this article you need only the knowledge of Windows programming covered in my first few articles, and a working knowledge of C. Since I'll be talking a bit about the Component Object Model on which DirectX is based, some knowledge of C++ will definitely help out this time around. If you don't have it, don't worry about a thing. I'll give you a brief explanation of the C++ elements of the article as we get to them. Rest assured that you don't need to know too much about C++ to use DirectX. All right? Let's do it.

What is DirectX?

DirectX is a game developer's API, or Application Development Interface. It is a group of software libraries that lets you make non-hardware-specific function calls, and handles all the details of the machine it happens to be running on. If the hardware supports the functions you're using, then hardware acceleration does the job, which means it's FAST. If not, DirectX has software emulation that takes over. It will be slower -- much slower in some cases -- but the point is that you don't have to worry about the low-level details. You can just focus on the game. The layer of DirectX that takes care of hardware-supported features is called the hardware acceleration layer (HAL), and the software emulation layer that kicks in if the hardware doesn't support a feature is called the hardware emulation layer (HEL).

DirectX has several components, each one with a specific purpose. DirectDraw is probably the most important one, beacuse in the end, all graphics end up going through it. Used by itself, it's basically a bitmap engine used for 2D graphics, but Direct3D -- another component of DirectX -- also works through DirectDraw when it comes time to display each frame. DirectDraw is what we'll be concentrating our attention on today. Other components of DirectX include DirectInput, which handles every input device you can think of, and probably a lot of devices neither of us has ever heard of; DirectSound, which is used for digital sound and music; DirectMusic, which is a more recent addition to the DirectX package and is used for MIDI music; and DirectPlay, for networking.

DirectX is based on a standard called the Component Object Model, which can be a bit confusing. Creating your own COM objects can be a tricky subject -- don't ask me how to do it -- but understanding the way it works isn't too bad. I'm only going to cover the properties of COM, not the details about how you would create a COM object yourself. So if this next section confuses you a little, don't worry. You don't have to recreate the functionality I'm describing. Using COM objects is a lot easier than creating them, as we'll see a bit later.

The Component Object Model (COM)

COM is a specification that defines rules by which reusable software components, called COM objects, are created. Each object is a collection of interfaces, which themselves are collections of functions -- basically, C++ classes. Using Microsoft's notation, the name of an interface always begins with an "I," much like the way regular class names begin with a "C."

Every COM interface is derived from a special base class called `IUnknown`, which has only three methods: `QueryInterface()`, `AddRef()`, and `Release()`. For you C programmers, a method is simply a member function of a class, and a class is nothing more than a `struct` whose member functions or variables may not be directly accessible to the user. I know that's not a great definition, but I don't want to get into a lengthy discussion of OOP here! Anyway, back to the three methods of `IUnknown`:

`QueryInterface()`: This is probably the most important function of a COM object. It is used to retrieve a pointer to the interface you want to work with. In order to get such a pointer, you must know the GUID of the interface. GUID stands for "globally unique identifier" and is a 128-bit value used to identify the interface. GUIDs used in this way are often called interface IDs, or IIDs. When creating a COM object, you can't just make up a GUID. You must use a special program to create one; most Win32 compilers come with such a tool. These programs apply an algorithm that guarantees that no GUID will ever be generated twice.

It sounds impossible, but it's not. There are enough combinations that these programs can contain rules by which they generate GUIDs such that no combination will ever be repeated. Let me put it this way. There are six billion people living on this planet. If every one of those people had five million kids, and every one of those kids had five million dogs, and men, women, children and dogs alike sat around all day, every day, generating one GUID per second, it would take them about seventy million years to use up all the combinations. That's probably not going to happen. :) I know of only one dog that sits around generating GUIDs, and it takes him at least ten seconds for each one.

`AddRef()`: COM objects use a variable called a reference count to track their usage. This function simply adds one to the reference count for that particular COM object, and must be called when retrieving a pointer to one of the interfaces contained within that COM object. You almost never do this manually. There are several ways you can obtain an interface pointer, and the ones you'll be using automatically call `AddRef()`.

`Release()`: When you're finished using an interface pointer, you must release it. This decrements the reference count for the COM object. So when you're writing a program that will be using one or more COM objects, the reference count for each COM object should be at zero when the program ends. Sometimes, if you create an object, and then use that object to create a child object, releasing the parent will also release the child. Since this is not guaranteed, it's usually a good idea to release objects in the reverse order that you created them in.

One of the nice things about COM is that its objects can be used with any language, and on any platform. How is this possible? One of the guidelines defining the COM specification states that every COM object must match the binary image that would be generated by a Microsoft Visual C++ compiler. It seems a bit strange to think of it that way, but consider this: once you compile your code, it just becomes binary data. You can't look at a string of 1s and 0s and divine whether it was C++ or Visual Basic before it was compiled, can you? Neither can I.

Another feature of COM is that you can recreate or update COM objects without changing or recompiling the programs that use them. The reason this is possible is that COM objects are usually dynamic-link libraries (`.DLL` files). They aren't compiled into programs that use them; they are linked at run-time. Thus, if a program you write ships with a few COM objects you created, and later on you want to add or change something in one of those COM objects, you simply need to supply users with the new `.DLL` file. No recompiling the main program is necessary.

The new version of the COM object, however, **must** contain all of the old interfaces and functions. This is to preserve backwards-compatibility. It guarantees that any program created to run with old versions of COM objects will function exactly the same if it's using new versions of those objects. Since DirectX follows the COM specification, the same rules apply. A program developed with DirectX 5.0 will work just fine if the user has DirectX 7.0 installed on his computer. Cool, hey?

Setting Up

There are three main things you need to develop programs with DirectX. First are the COM objects themselves, which are inside `.DLL` files. These `.DLL`s need to be registered with Windows, which is what happens when you install DirectX. Inside these objects are interfaces like `IDirectDraw` that we'll be using to create DirectX applications. But this isn't enough, because handling DirectX on the COM level would be tedious and frustrating. There are times when you'll make a few direct COM calls, but for the most part, we want to use something easier.

That's where the static libaries (`.LIB` files) come into play. These are the main part of the DirectX SDK, which is available free from Microsoft. They contain "wrapper" functions to make using DirectX a whole lot easier. To use the various components of DirectX, you need to link the appropriate libraries to your project. For DirectDraw, this would be `ddraw.lib`.

Finally, you need the DirectX header files (`.H` files) that contain the function prototypes, macros, constants, and variable types that are used for each component of DirectX. For DirectDraw, this would be `ddraw.h`.

To make sure you're using the correct versions of these files, you must set your compiler to include the directories where you have installed the SDK. There are several lists of directories to search for library and header files, and the SDK directories must be first on the list. This is important, because most Win32 compilers ship with some version of these files already included. If the SDK directories are not first on the list, the older versions that came with your compiler will be included. The two directories you need to set are those for library and header files. For example, if you installed the DirectX SDK in your `c:\mssdk` directory, the first item on the include file directory list should be `c:\mssdk\include` and the first item on the library directory list should be `c:\mssdk\lib`. In Visual C++, you can find these settings by choosing "Options..." from the Tools menu.

For this and any remaining articles dealing with DirectX, we will be using DirectX 7.0 (or higher). If you don't have the DirectX SDK, you'll need to get it from Microsoft. You don't need the whole thing, just the essentials. The entire SDK has tons of examples packed in with it, which makes for a whopping 128MB download. If you're like me and you're not lucky enough to have high-speed Internet access, you probably don't want to wait 12 hours for that sucker to download. Scroll down on the SDK download page, and you'll find the links for only the libaries, headers, and docs. It's around 10MB that way, which is much more reasonable.

DirectX Version Numbers

You wouldn't think something as simple as a version number would require an explanation, but remember who we're dealing with here. :) Microsoft may have created an incredible technology in DirectX, but they're still out to confuse everyone. With each release of DirectX, not all of the interfaces are updated. Therefore, even though there have been seven versions of DirectX, there have not been seven versions of DirectDraw. When DirectX 6 was the most recent version, the most recent interface for DirectDraw was `IDirectDraw4`, not `IDirectDraw6`. The most recent release which updated DirectDraw was DirectX 7, and so we'll be dealing with `IDirectDraw7`. Strange, hey? I thought I'd bring that up now, so you know what's going on when you see it later.

One last thing. When I wrote this article, DirectX 7 was the most recent API available, but you've probably heard that now we have DirectX 8. You have probably also heard that as of now, DirectDraw is no longer being updated. Instead, there's DirectX Graphics, which is one big catch-all graphics API. Now, just because DirectDraw isn't being updated doesn't mean we can't still use it. This is COM, after all. If you want to write 2D games with DirectX 8 interfaces, you need to employ 3D methods to create a 2D view. It sounds cool, and it is, since using 3D gives you access to more hardware-supported features like alpha-blending, but it has a problem. It's going to be slow if the user's machine doesn't have hardware acceleration.

DirectDraw is easier to learn, and its software layer is nice and fast since the added processor burden of using 3D is not present. So I will be using DirectDraw in this series. This also has to do with the fact that this and the next couple of articles in the series were written before the release of DirectX 8, and also because I still know very little about how to use the DirectX 8 interfaces, so I can hardly go telling you how to do it, can I? :) In any case, DirectDraw still makes a great introduction to the world of DirectX.

Overview of DirectDraw

To set up DirectDraw for use in your program, you need to do a minimum of four things in the initialization section of your program. They are:

1. Create a DirectDraw object.
2. Set the cooperation level.
3. Set the screen resolution and color depth (fullscreen applications).
4. Create at least one DirectDraw surface.

Before looking at how these steps are accomplished, let's go over a brief explanation of what each one means. First, you need to create a DirectDraw object, meaning we want to retrieve a pointer to the `IDirectDraw7` interface. That's pretty simple, right? There are actually three ways to do this. You can make a direct COM call, or use one of two DirectDraw wrapper function. Each has its merits, so we'll be looking at all three of them in a bit.

Second, you need to set the cooperation level. This is probably new to you. Cooperation is a concept that's introduced because Windows is a multitasking operating system. The idea is that all programs running at any given time have to inform Windows what resources they're going to be using, and in what way. This is to make sure Windows doesn't try to take resources essential to your program and allocate them to something else, and so it has the details of how your program will be operating. Don't worry, it's just a simple function call.

The third item on the list is familiar, right? If you're writing a fullscreen application, like a game will usually be, you need to set the screen resolution and color depth to where you want them. Doing this in a windowed application is usually not a good idea, because it can cause problems with other programs that are running at the same time. You'd also have to make certain to restore it when you are finished. In fullscreen mode, setting the resolution is just a single call to DirectDraw, and when your program ends, the Windows desktop settings are restored.

Lastly, and most importantly, is the concept of a DirectDraw surface. Manipulating surfaces is what DirectDraw is all about. Basically, a surface is an area in memory reserved for graphics and graphical operations. The size of a DirectDraw surface is defined by its width and height, in pixels, so you can think of it as a rectangular area for drawing graphics. It has its own interface, called `IDirectDrawSurface7`. There are three main kinds of surfaces, each of which we'll be using between this article and the next.

Primary surfaces: Every DirectDraw application must have a primary surface in order to accomplish anything. The primary surface is the surface that represents the user's display. Its contents are always visible. As such, a primary surface is automatically set to the width and height of the screen mode.

Back buffers: Back buffers are surfaces that are attached to the primary surface, but not visible. This is how animation is created without flicker. Normally, you draw each frame on a back buffer, and then copy the contents of the back buffer to the primary urface, causing it to appear instantaneously. Since they are attached to the primary surface, they are also the same size as the display.

Offscreen buffers: These are very much like back buffers, only they are not attached to the primary surface. They are most often used for storing bitmaps, although you can do anything you want with them. Offscreen buffers can be any size you want. The only limitation is the amount of memory the system has.

DirectDraw surfaces can either be created in system memory, or directly on the video card in VRAM. If you have all of your surfaces in video memory, the speed is going to be fantastic. System memory is slower. Also, if you have one surface stored in video memory, and one stored in system memory, performance will suffer quite a bit, especially if the video card in question has lousy memory bandwidth. Anyway, the moral is that if you can get away with creating all of your surfaces in video memory, it's probably worth doing so.

All right, now that we have some idea for what we have to do, let's see how we go about doing it. Here's the plan. We're going to create a fullscreen DirectX application that runs in 640x480x16bpp. I'll go through all the DirectX stuff you need to do that, but before you can go setting up DirectX, you need to have a window to operate on. That much is up to you! We went through it in my first article, so you should be pretty familiar with creating windows by now. Since this is going to be a fullscreen application, you're going to want a window that doesn't have any Windows controls on it, so for the window style, use `WS_POUP | WS_VISIBLE`. Got it? All right, here we go.

Creating a DirectDraw Object

As I said before, there are three ways to do this. We could either use one of two DirectDraw wrapper functions, or make a call directly to the COM object. Let's look at all of them, just to get ourselves used to this stuff. The last method I'll show you is by far the easiest, so you'll probably want to use that. As for the other two, they give you a first look at a few different things you'll come across again. First, have a look at `DirectDrawCreate()`:

```
HRESULT WINAPI DirectDrawCreate(
    GUID FAR *lpGUID,
    LPDIRECTDRAW FAR *lplpDD,
    IUnknown FAR *pUnkOuter
);
```

Looks a bit strange, doesn't it? The `HRESULT` return type is standard for DirectDraw functions. If successful, the return value is `DD_OK`. Each function has several constants it can return describing various errors, but in the interest of keeping the length down a bit, I won't list the values for every function. You can always look them up. One thing I will mention is that there are two useful macros you can use to determine the result of a DirectDraw function call: `SUCCEEDED()` and `FAILED()`. Pretty self-explanatory, wouldn't you say? Just put the function call inside the macro to see what's happening. Anyway, the parameters for the function are:

`GUID FAR *lpGUID`: This is the address of the GUID identifying the driver to use. For the default, simply pass `NULL`. Otherwise, there are two values you can use here for debugging purposes:

`DDCREATE_EMULATIONONLY`	Creating a DirectDraw object with this flag means that no hardware support will be utilized, even if it is available. All operations are handled in the HEL.
`DDCREATE_HARDWAREONLY`	Creating a DirectDraw object with this flag means that only the HAL will be used; if the hardware doesn't support an operation you call, the function calling that operation will return an error rather than invoking the HEL.

`LPDIRECTDRAW FAR *lplpDD`: This is the address of a pointer to a DirectDraw object; it will be initialized as such if the function succeeds. You must declare a variable of type `LPDIRECTDRAW` that will hold your interface pointer, then pass the address to that pointer.

`IUnknown FAR *pUnkOuter`: This parameter is reserved for advanced COM features that are not yet implemented. Always pass `NULL`.

That wasn't so bad, but there is a problem. This function gives you a pointer to an IDirectDraw interface, but we want a pointer to the IDirectDraw7 interface! What do we do? The answer is to call the QueryInterface() method of the DirectDraw object we have obtained, and request the IDirectDraw7 interface. Here's what the call looks like:

```
HRESULT QueryInterface(
    REFIID iid,         // Identifier of the requested interface
    void **ppvObject    // Address of output variable that receives the
);                      // interface pointer requested in iid
```

The parameters are pretty straightforward:

REFIID iid: Remember when I said GUIDs are sometimes called IIDs for interfaces? This is what I was talking about. The IID for IDirectDraw7 is IID_IDirectDraw7. To use this constant, you must link the dxguid.lib library to your project.

void **ppvObject: Similar to what we did with DirectDrawCreate(), except here you should declare a pointer of type LPDIRECTDRAW7, and pass its address. If you're using Visual C++ 6.0, you'll probably need a typecast here.

Now we have two interface pointers: one to an IDirectDraw interface, and one to IDirectDraw7. The latter is what we want; the former is useless. Remember that when you're done using an interface, you must call its Release() method to decrement the reference count for its COM object. The prototype for Release() is simple:

```
ULONG Release(void);
```

The return value is the resulting reference count, which you would only need to worry about for testing or debugging purposes. Also, it's recommended that for safety, you should set a pointer to a released interface to NULL. It's also usually a good idea to set such pointers to NULL when you declare them in the first place. Are you following me? It can be a bit much to remember in the beginning, but it'll become second nature to you in no time. Let's bring it all together and look at an example of getting an IDirectDraw7 interface pointer:

```
LPDIRECTDRAW lpdd = NULL;   // pointer to IDirectDraw (temporary)
LPDIRECTDRAW7 lpdd7 = NULL; // pointer to IDirectDraw7 (what we want)

// get the IDirectDraw interface pointer
if (FAILED(DirectDrawCreate(NULL, &lpdd, NULL)))
{
  // error-handling code here
}

// query for IDirectDraw7 pointer
if (FAILED(lpdd->QueryInterface(IID_IDirectDraw7, (void**)&lpdd7)))

{
  // error-handling code here
}
else
```

```
{
  // success! release IDirectDraw since we don't need it anymore
  lpdd->Release();
  lpdd = NULL;
}
```

Now, if you're a C programmer, you may be a little confused by the way I called QueryInterface() and Release(). You've probably seen the -> operator before. It's used to dereference members of a struct when using a pointer to the struct rather than the variable itself. It's the same thing with objects, except that in this case, the member you're making reference to is a function instead of a variable. While we're on the topic, I'll introduce another bit of C++ notation as well, the scope resolution operator (::). It's used to show the class (or interface, in our case) of which a certain function or variable is a member. In this case, QueryInterface() is a method of the base interface IUnknown, so we would refer to it as IUnknown::QueryInterface(). I'll be using this often in the future, so remember it!

To be honest, that's only useful for demonstrating how to use the QueryInterface() method, which is a part of all DirectX interfaces, so let's move on. Next, just to show you what it looks like, let's use the COM method. The nice thing about doing it this way is that you can get an IDirectDraw7 interface pointer immediately, instead of getting the older pointer and querying for the new one. First, you have to initialize COM, like this:

```
HRESULT CoInitialize(LPVOID pvReserved);
```

It doesn't get much easier. The parameter is reserved as must be set to NULL. When you're finished with COM calls, you need to uninitialize it, with an even easier call:

```
void CoUninitialize(void);
```

I usually call CoInitialize() in the very beginning of my DirectX programs, and call CoUninitialize() at the very end, after I've released all my DirectX objects. Once COM is initialized, you can get the pointer you're after by calling CoCreateInstance(), which can get a little ugly:

```
STDAPI CoCreateInstance(
    REFCLSID rclsid,     // Class identifier (CLSID) of the object
    LPUNKNOWN pUnkOuter, // Pointer to whether object is or isn't part
                         //  of an aggregate
    DWORD dwClsContext,  // Context for running executable code
    REFIID riid,         // Reference to the identifier of the interface
    LPVOID *ppv          // Address of output variable that receives
);                       //  the interface pointer requested in riid
```

If the call succeeds, the return value is S_OK. The parameters require a bit of explanation, so here's the list:

REFCLSID rclsid: This is a class identifier (not to be confused with the IID), which also has constants defined for it. For IDirectDraw7, use CLSID_DirectDraw. Notice the version number is not specified because this is a class identifier, not an interface identifier.

LPUNKNOWN pUnkOuter: This is the same thing we saw in DirectDrawCreate(). Set it to NULL.

`DWORD dwClsContext`: The value required here is called an execution context, and it defines the way that the code that manages the newly created object will run. The values it can take are defined in an enumeration called `CLSCTX`. In this case, use `CLSCTX_ALL`, which includes all the possible values.

`REFIID riid`: We saw this in our call to `QueryInterface()`. The IID is `IID_DirectDraw7`.

`LPVOID *ppv`: This was also in `DirectDrawCreate()`, and is the address of the pointer to our interface.

That one call takes the place of our calls to `DirectDrawCreate()`, `QueryInterface()`, and `Release()` in the previous example, so it's a bit less code, but it really doesn't matter which way you decide to use. The COM call is a little less hassle than the first method we saw, since it doesn't require getting that extra interface pointer. Once you create an object using `CoCreateInstance()`, though, you have to initialize it by calling the `Initialize()` method of the object. In C++ this would be written as `IDirectDraw7::Initialize()`. Here's the prototype:

```
HRESULT Initialize(GUID FAR *lpGUID);
```

Use the same GUID you would have used in the call to `DirectDrawCreate()`. In other words, `NULL`. So before moving on, let me show you the example of using COM to create a DirectDraw object:

```
LPDIRECTDRAW7 lpdd7; // interface pointer

// initialize COM
CoInitialize(NULL);

// create the object
CoCreateInstance(CLSID_DirectDraw, NULL, CLSCTX_ALL, IID_IDirectDraw7, (void**)&lpdd7);

// initialize the object
lpdd7->Initialize(NULL);
```

It's good to see an example of calling COM directly, since there may come a time when you want to -- or have to -- do this instead of calling one of the high-level wrapper functions. Finally, now that you've seen the hard ways to do this, let's use the easy way. There is another wrapper function that takes care of everything we want to do, in a single call. No querying for higher interfaces, no setting up COM, no nothing. Here's the function:

```
DirectDrawCreateEx(
    GUID FAR *lpGuid,
    LPVOID *lplpDD,
    REFIID iid,
    IUnknown FAR *pUnkOuter
);
```

All these parameters should look familiar, because we've just seen them. The first, second, and fourth parameters are the same ones we passed to `DirectDrawCreate()`, only in this case we need to cast the address of our interface pointer to `void**` -- don't ask me why; it wasn't my idea. The third parameter, `riid`, is the interface ID that we passed to `CoCreateInstance`, so just use `IID_IDirectDraw7` and we're good to go.

That's it! Now that we've got our DirectDraw object, we can start doing things with it. The first two things we want to do are setting the cooperation level and screen resolution, so let's take a look.

Setting Cooperation and Resolution

We don't need anything fancy here; setting the cooperation level with Windows is as easy as calling `IDirectDraw7::SetCooperativeLevel()`, and setting the resolution is a simple call to `IDirectDraw7::SetDisplayMode()`. Is that cool or what? First up is the cooperation level. Here's the function you use:

```
HRESULT SetCooperativeLevel(
    HWND hWnd,
    DWORD dwFlags
);
```

The return type is the `HRESULT` you should already be getting used to seeing from these DirectX functions. On all such calls, you should be using the `SUCCEEDED()` or `FAILED()` macros to test the result of the call. Here are the parameters:

`HWND hWnd`: Something familiar! Pass the handle to your main window, so Windows knows who's going to be hording all of its resources. :)

`DWORD dwFlags`: This should look familiar too. Every time we've seen a `dwFlags` parameter, it almost always results in a big list of constants that can be combined with the `|` operator. I'd hate to disappoint you!

DDSCL_ALLOWMODEX	Allows the use of Mode X display modes (like 320x200, 320x240, or 320x400). This must be combined with DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN.
DDSCL_ALLOWREBOOT	Allows the use of the Ctrl+Alt+Del Windows shortcut while running.
DDSCL_EXCLUSIVE	Indicates the application will have exclusive access to the display screen; must be combined with DDSCL_FULLSCREEN.
DDSCL_FULLSCREEN	Microsoft's effort to create redundancy; it must be combined with DDSCL_EXCLUSIVE. :)
DDSCL_NORMAL	Indicates a normal Windows application. Use this for windowed DirectX programs. Cannot be combined with DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN.
DDSCL_NOWINDOWCHANGES	Indicates that DirectDraw may not minimze or restore the application window on activation.

There are also a few flags specific to Windows 98 and NT 5.0 that deal with device windows, but I've left them off the list because we'll only be using these simple flags. Since we want to create a fullscreen application in 640x480x16 resolution, we would use the following call:

```
lpdd7->SetCooperativeLevel(hwnd, DDSCL_ALLOWREBOOT | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
```

So now that the cooperation level is all taken care of, let's look at the function used to change the screen resolution:

```
HRESULT SetDisplayMode(
    DWORD dwWidth,
    DWORD dwHeight,
    DWORD dwBPP,
    DWORD dwRefreshRate,
    DWORD dwFlags
);
```

Remember to test for success or failure! Most of the parameters are just what you'd expect:

DWORD dwWidth, dwHeight: The dimensions of the new display mode, in pixels.

DWORD dwBPP: The color depth of the new display mode, in bits per pixel. This can be set to 8, 16, 24, or 32. be warned, most video cards do not support 24-bit color.

DWORD dwRefreshRate: This is used to set the refresh rate for the screen, but you should probably just set it to 0, which indicates that the default for your display mode should be used.

DWORD dwFlags: Sorry, I haven't got a big list for you this time. The only valid flag is DDSDM_STANDARDVGAMODE, which sets screen mode 0x13 (the DOS coder's best friend!) instead of Mode X 320x200x8 mode. If you're using something other than 320x200x8, which you almost certainly will be, set this to 0.

That's all it takes to set the resolution to what you want it to be. Keep in mind that the video card may not support every resolution you want to create. 640x480, 800x600, 1024x876, etc. are pretty standard, but if you ask for something like 542x336, you'll almost certainly get an error. Some devices will build some odd display modes, but the idea is to be compatible with as many machines as possible, right? Let's move on.

Creating Surfaces

It's time for something that requires a little more than just a function call! Creating surfaces isn't too tough. Actually, it is accomplished with a single function call, but first you need to fill out a structure that describes the surface you want to create. Before I show you this, I just want to say that you don't have to fill out the whole thing, so never fear. :) Here it is, the DDSURFACEDESC2:

```
typedef struct _DDSURFACEDESC2 {
    DWORD          dwSize;
    DWORD          dwFlags;
    DWORD          dwHeight;
    DWORD          dwWidth;
    union
    {
        LONG       lPitch;
        DWORD      dwLinearSize;
    } DUMMYUNIONNAMEN(1);
    DWORD          dwBackBufferCount;
    union
    {
        DWORD      dwMipMapCount;
        DWORD      dwRefreshRate;
    } DUMMYUNIONNAMEN(2);
    DWORD          dwAlphaBitDepth;
```

```
        DWORD           dwReserved;
        LPVOID          lpSurface;
        DDCOLORKEY      ddckCKDestOverlay;
        DDCOLORKEY      ddckCKDestBlt;
        DDCOLORKEY      ddckCKSrcOverlay;
        DDCOLORKEY      ddckCKSrcBlt;
        DDPIXELFORMAT   ddpfPixelFormat;
        DDSCAPS2        ddsCaps;
        DWORD           dwTextureStage;
    } DDSURFACEDESC2, FAR *LPDDSURFACEDESC2;
```

DirectX has lots of big structures. This one is pretty big, and what's worse, it's got all sorts of structures nested inside it! Anyway, let's take a look at this monster and see what it's got to offer. I'm only going to cover the important ones, so I don't grow old writing this.

`DWORD dwSize`: Any DirectX structure has a dwSize member, which must be set to the size of the structure. It's there so that functions receiving a pointer to one of these structures can determine its size.

`DWORD dwFlags`: Great, more flags. :) These flags are used to tell a receiving function which fields are important. Any field containing valid information must have its corresponding flag specified in `dwFlags`, and they can all be combined with `|` as usual. Here's the list:

DDSD_ALL	Indicates that all input fields are valid.
DDSD_ALPHABITDEPTH	Indicates that the `dwAlphaBitDepth` member is valid.
DDSD_BACKBUFFERCOUNT	Indicates that the `dwBackBufferCount` member is valid.
DDSD_CAPS	Indicates that the `ddsCaps` member is valid.
DDSD_CKDESTBLT	Indicates that the `ddckDestBlt` member is valid.
DDSD_CKDESTOVERLAY	Indicates that the `ddckDestOverlay` member is valid.
DDSD_CKSRCBLT	Indicates that the `ddckSrcBlt` member is valid.
DDSD_CKSRCOVERLAY	Indicates that the `ddckSrcOverlay` member is valid.
DDSD_HEIGHT	Indicates that the `dwHeight` member is valid.
DDSD_LINEARSIZE	Indicates that the `dwLinearSize` member is valid.
DDSD_LPSURFACE	Indicates that the `lpSurface` member is valid.
DDSD_MIPMAPCOUNT	Indicates that the `dwMipMapCount` member is valid.
DDSD_PITCH	Indicates that the `lPitch` member is valid.
DDSD_PIXELFORMAT	Indicates that the `ddpfPixelFormat` member is valid.
DDSD_REFRESHRATE	Indicates that the `dwRefreshRate` member is valid.
DDSD_TEXTURESTAGE	Indicates that the `dwTextureStage` member is valid.
DDSD_WIDTH	Indicates that the `dwWidth` member is valid.

`DWORD dwHeight, dwWidth`: These are the dimensions of the surface in pixels.

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

LPVOID lpSurface: This is a pointer to the memory represented by the surface. No matter what display mode you're using, DirectDraw creates a linear addressing mode you can use to manipulate the pixels of the surface. In order to do this, you must lock the surface... but we're getting ahead of ourselves!

DWORD dwBackBufferCount: This is the number of back buffers that will be chained to the primary surface, either for double- or triple-buffering, or for a page flipping chain. Again, more on this later.

DWORD ddckDestBlt, ddckSrcBlt: These are color keys, which control which color pixels can be written to, or written, respectively. This is used to set transparent colors in bitmaps, as we'll see in a future article.

DDPIXELFORMAT ddpfPixelFormat: This structure contains a number of descriptors for the pixel format of the display mode. We'll be covering this next time as well, so I won't show you this giant structure right now.

DDSCAPS2 ddsCaps: This is the last important one, and it's another structure full of control flags. Thankfully, the structure a small one, and only one of its members is important right now. Take a look:

```
typedef struct _DDSCAPS2{
    DWORD dwCaps;
    DWORD dwCaps2;
    DWORD dwCaps3;
    DWORD dwCaps4;
} DDSCAPS2, FAR* LPDDSCAPS2;
```

The only important one is dwCaps. The third and fourth members aren't even used at all; they're just there for future expansion. Anyway, dwCaps can take the following values, logically combined with the | operator. This is only a partial list, leaving out some that are advanced or not currently implemented.

DDSCAPS_BACKBUFFER	Indicates that this surface is a back buffer on a surface flipping structure.
DDSCAPS_COMPLEX	Indicates a complex surface, which consists of a primary surfaces and one or more attached surfaces, usually for page flipping.
DDSCAPS_FLIP	Indicates that this surface is part of a surface flipping structure. The result is that a front buffer along with one or more back buffers are created.
DDSCAPS_FRONTBUFFER	Indicates that this surface is the front buffer on a surface flipping structure.
DDSCAPS_LOCALVIDMEM	Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.
DDSCAPS_MODEX	Indicates that this surface is a 320×200 or 320×240 Mode X surface.
DDSCAPS_NONLOCALVIDMEM	Indicates that this surface exists in nonlocal video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

DDSCAPS_OFFSCREENPLAIN	Indicates that this is any plain offscreen surface that is not a texture, back buffer, etc.
DDSCAPS_OWNDC	Indicates that this surface will have a device context association for a long period.
DDSCAPS_PRIMARYSURFACE	Indicates that this is the primary surface.
DDSCAPS_STANDARDVGAMODE	Indicates that this surface is a standard VGA surface rather than a Mode X surface. Cannot be used with the DDSCAPS_MODEX flag.
DDSCAPS_SYSTEMMEMORY	Indicates that this surface exists in system memory.
DDSCAPS_TEXTURE	Indicates that this surface can be used as a 3D texture, though it is not necessary to use it that way.
DDSCAPS_VIDEOMEMORY	Indicates that this surface exists in display memory.

All right, we're finally done looking at structures. Now we're just about ready to create a surface. The first step is to fill out a DDSURFACEDESC2 structure. Microsoft recommends that before using a structure which you won't be filling out completely, you should zero it out to initialize it. To this end, I usually use a macro something like this:

```
#define INIT_DXSTRUCT(dxs) { ZeroMemory(&dxs, sizeof(dxs)); dds.dwSize = sizeof(dxs); }
```

This can be used for any DirectX structure, since they all have the dwSize member, so it's pretty convenient. If you've never seen ZeroMemory() before, it's just a macro that expands into a memset() call. It's #defined in the Windows headers, so you don't need to #include anything new to use it.

After initializing the structure, the minimum you'll need to fill out depends on the surface. For primary surfaces, you'll just need ddsCaps and dwBackBufferCount. For offscreen buffers, you'll also need dwHeight and dwWidth, but not dwBackBufferCount. For some surfaces you may also want to use the color keys, but we're not that far yet. After you've filled out the structure, you make a call to IDirectDraw7::CreateSurface(), which looks like this:

```
HRESULT CreateSurface(
    LPDDSURFACEDESC2 lpDDSurfaceDesc,
    LPDIRECTDRAWSURFACE7 FAR *lplpDDSurface,
    IUnknown FAR *pUnkOuter
);
```

The parameters, you can probably figure out. But here they are, since we're just getting used to all this crazy DirectX stuff:

LPDDSURFACEDESC2 lpDDSurfaceDesc: This is a pointer to the DDSURFACEDESC2 you'll need to fill out to describe the surface.

LPDIRECTDRAWSURFACE7 FAR *lplpDDSurface: Since a surface is defined by an interface pointer, you need to create a variable of type LPDIRECTDRAWSURFACE7 to hold the pointer, and then pass its address here.

IUnknown FAR *pUnkOuter: Starting to see a pattern? Whenever you see something called pUnkOuter, it's for advanced COM stuff we don't want to mess around with. :) Set it to NULL.

Let's run through an example. Suppose that for our program, we want a primary surface with one back buffer attached, and one offscreen buffer to use for bitmap storage. Assuming we already have our IDirectDraw7 interface pointer, the following code would create the primary surface:

```
DDSURFACEDESC2 ddsd;  // surface description structure
LPDIRECTDRAWSURFACE7 lpddsPrimary = NULL;  // primary surface

// set up primary drawing surface
INIT_DXSTRUCT(ddsd);                               // initialize ddsd
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;  // valid flags
ddsd.dwBackBufferCount = 1;                        // one back buffer
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |    // primary surface
                      DDSCAPS_COMPLEX |            // back buffer is chained
                      DDSCAPS_FLIP |               // allow page flipping
                      DDSCAPS_VIDEOMEMORY;         // create in video memory

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

Now, when you specify the DDSCAPS_COMPLEX flag, the call to CreateSurface() actually creates the front buffer along with any back buffers it may have. Since we specified that there was to be one back buffer, all we need to do is get the pointer to it. This is done by calling IDirectDrawSurface7::GetAttachedSurface():

```
HRESULT GetAttachedSurface(
    LPDDSCAPS2 lpDDSCaps,
    LPDIRECTDRAWSURFACE7 FAR *lplpDDAttachedSurface
);
```

The parameters are pretty easy to deal with:

LPDDSCAPS2 lpDDSCaps: A pointer to a DDSCAPS2 structure which describes the attached surface. You can use the corresponding member of our DDSURFACEDESC2 structure.

LPDIRECTDRAWSURFACE7 FAR *lplpDDAttachedSurface: This is the address of the interface pointer that will represent the attached surface. Simply declare a pointer and pass its address.

With that relatively painless function, we can get the back buffer which was created along with the primary surface. The following code does the job:

```
LPDIRECTDRAWSURFACE7 lpddsBack = NULL;  // back buffer

// get the attached surface
ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
if (FAILED(lpddsPrimary->GetAttachedSurface(&ddsd.ddsCaps, &lpddsBack)))

{
  // error-handling code here
}
```

Are you starting to get the hang of this stuff? If you're still having trouble remembering all these steps, just give it time. And nobody remembers every member of every structure there is, so don't worry about those huge lists of members and flags. That's why you have your MSDN Library CD. :) Anyway, the last step is to create the offscreen buffer. Suppose we wanted it to be 400 pixels wide and 300 pixels high. We would go about creating it like this:

```
LPDIRECTDRAWSURFACE7 lpddsOffscreen = NULL;  // offscreen buffer

// set up offscreen surface
INIT_DXSTRUCT(ddsd);                                 // initialize ddsd
ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;  // valid flags
ddsd.dwWidth = 400;                                  // set width
ddsd.dwHeight = 300;                                 // set height
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |       // offscreen buffer
                      DDSCAPS_VIDEOMEMORY;            // video memory

// create offscreen buffer
if (FAILED(lpdd7->CreateSurface(&ddsd, &lpddsOffscreen, NULL)))
{
  // error-handling code here
}
```

And we're all set! Now we've got all of our surfaces created and we're ready to do graphics. Of course, the only problem is that this article has gone pretty long already, so we're going to have to stop here. You can now create a working DirectX program, even though it won't do anything except set up your surfaces. Remember that when you're done with the DirectDraw interface, and with all your surfaces, you must release them all, and it's best to release in the opposite order of creation.

Closing

Sorry to cut this off before we got to any graphics, but if I were to start talking about graphics, the plain text version of this article would go over 100KB, to say nothing of the HTML-ified version. :) There's so much to explain when it comes to working with graphics, so I think I'm going to split it up over the next two articles. The next installment will cover pixel-based graphics and setting up palettized modes, and the article after that will discuss bitmaps and clipping. Until then, if you have any questions, send me an E-mail, or catch me on ICQ at UIN #53210499.

See you next time!

Discuss this article in the forums