Navigation

Home
DirectX for Windows 8
  DirectX 11.2
  DirectX 11.1
DirectX for Desktop
  DirectX 11
  DirectX 9
Useful Resources
About DirectXTutorial
DirectXTutorial Premium
Testimonials

Contact

Contact me here:
Twitter: @dastopher
Email: chris@directxtutorial.com

Or leave me feedback:
Quick Feedback

Lesson 2: Initializing Direct3D

Previous Next

Lesson Overview

Now that you've studied the concepts on which Direct3D operates, let's start diving into the practical end of things by building a simple Direct3D program. In this program we'll just initialize Direct3D, close it down again. It isn't much, it's not even "Hello World", but it's a good start.

COM

And COM is what?

COM stands for Component Object Model. COM is a method of creating very advanced objects that, well, they act a lot like Legos actually.

Legos, as you know, can be stuck together to create more advanced shapes. No single Lego actually cares about any other Lego in the set. They are all compatible with each other, and all you have to do is stick them together to get them to work. If you want to change pieces, all you have to do is unplug one piece and put another in its place.

And so it is with COM. COM objects are actually C++ classes or groups of classes from which you can call functions and achieve certain aims. No class requires another to operate, and they don't really need to work together to get things done together, but you can plug them in or unplug them as you desire without changing the rest of the program also.

For example, say you had a game distributed broadly and you wanted to upgrade it. Well, instead of keeping track of and shipping a new copy to every single user who ever bought your game, all you have to do is say "Upgrade! Right Here!". They download the updated COM object, and the new object plugs right in to your program without further hassle. Nice, huh?

I won't get too detailed into COM, because it is far too complex for what we need. It's job is to get all the complex stuff out of the way so that you have an easy time. And if that's its job, what would be the purpose of learning all that complex material?

So why COM? Well, DirectX is actually a series of COM objects, one of which is Direct3D. Direct3D is a COM object that has other COM objects inside it. Ultimately, it contains everything you need to run 2D and 3D graphics using software, hardware, or whateverware.

So because Direct3D is already stored in classes, don't be surprised when you see Direct3D functions being called like this:

```
device->CreateRenderTargetView()

device->Release()
```

We use the indirect member access operator to access the functions CreateRenderTargetView() and Release() from a Direct3D class. We'll get more into this when we see how it is applied in practice. I'm going to try to avoid unneeded theory from here on out.

Even though COM's job is to hide all the complexity, there are four things you need to know about it.

1. A COM object is a class or set of classes controlled by an interface. An interface is a set of functions that, well, controls a COM object. In the example above, "device" is a COM object, and the functions control it.

2. Each type of COM object has a unique ID. For example, the Direct3D object has its own ID, and the DirectSound object has its own ID. Sometimes you need to use this ID in the code.

3. When done using a COM object, you must always call the function Release(). This will tell the object to free its memory and close its threads.

4. COM objects are easy to identify, because they typically start with an 'I', such as 'ID3D10Device'.

Don't worry about the details of these four points. We'll see how they are all applied in a moment.

For now let's get on with the actual code:

Direct3D Headers

Before we get to the actual Direct3D code, let's talk about the header files, library files, and other such fun things. In our demo programs, we'll have these things at the top, giving us global access to Direct3D.

Let's take a look at these and see what they are.

```
// include the basic windows header files and the Direct3D header files
#include <windows.h>
#include <windowsx.h>
#include <d3d11.h>
#include <d3dx11.h>
#include <d3dx10.h>

// include the Direct3D Library file
#pragma comment (lib, "d3d11.lib")
#pragma comment (lib, "d3dx11.lib")
#pragma comment (lib, "d3dx10.lib")

// global declarations
IDXGISwapChain *swapchain;             // the pointer to the swap chain interface
ID3D11Device *dev;                     // the pointer to our Direct3D device interface
ID3D11DeviceContext *devcon;           // the pointer to our Direct3D device context

// function prototypes
void InitD3D(HWND hWnd);     // sets up and initializes Direct3D
void CleanD3D(void);         // closes Direct3D and releases memory

// the WindowProc function prototype
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);
```

#include <d3d11.h>
#include <d3dx11.h>

This includes the Direct3D 11 header files. These files consist of various declarations to the actual methods contained in the Direct3D 11 library.

The files include different things. The d3d11.h file includes the core pieces of Direct3D. The d3dx11.h file includes extensions to Direct3D which aren't necessary to a graphics library, but which can come in very handy when writing games or other graphics programs.

Note: Not all compilers will automatically locate these files. In some cases you will need to set up your project to look in the folders of the DirectX SDK. If you are using Visual Studio, I've written a quick mini-lesson on how to do this here.

#include <d3dx10.h>

Direct3D 11 is actually an extention of Direct3D 10. Because of this, it borrows many macros, functions and classes from Direct3D 10. This header allows us to use these in our program.

#pragma comment (lib, "d3d11.lib")
#pragma comment (lib, "d3dx11.lib")
#pragma comment (lib, "d3dx10.lib")

This includes the Direct3D 11 library files. The "#pragma comment" directive places a certain piece of information in your project's object file. With our first parameter, lib, we indicate that we want to add a library file to the project. We then specify which file, "d3d11.lib", "d3dx11.lib" and "d3dx10.lib".

ID3D11Device *dev;

This variable is a pointer to a device. In Direct3D, a device is an object that is intended to be a virtual representation of your video adapter. What this line of code means is that we will create a COM object called ID3D11Device. When COM makes this object, we will ignore it, and access it only indirectly using this pointer. We'll cover how this happens in a moment.

ID3D11DeviceContext *devcon;

A device context is similar to a device, but it is responsible for managing the GPU and the rendering pipeline (the device mostly handles video memory). This object is used to render graphics and to determine how they will be rendered.

IDXGISwapChain *swapchain;

As we covered last lesson, the swap chain is the series of buffers which take turns being rendered on. This variable is a pointer to such a chain.

Notice that this object does not belong to Direct3D, but is actually part of the DXGI, underlying Direct3D.

Launching Direct3D

The first step to actually coding Direct3D is to create the above three COM objects and initialize them. This is done by using a single function, and a struct containing graphics device information. Let's take a look at this function here, then go over its parts. I didn't bother to bold the new parts, because the entire thing is new.

```
// this function initializes and prepares Direct3D for use
void InitD3D(HWND hWnd)
{
    // create a struct to hold information about the swap chain
    DXGI_SWAP_CHAIN_DESC scd;

    // clear out the struct for use
    ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));

    // fill the swap chain description struct
    scd.BufferCount = 1;                                    // one back buffer
    scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;     // use 32-bit color
    scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;      // how swap chain is to be used
    scd.OutputWindow = hWnd;                                // the window to be used
    scd.SampleDesc.Count = 4;                               // how many multisamples
    scd.Windowed = TRUE;                                    // windowed/full-screen mode

    // create a device, device context and swap chain using the information in the scd struct
    D3D11CreateDeviceAndSwapChain(NULL,
                                  D3D_DRIVER_TYPE_HARDWARE,
                                  NULL,
                                  NULL,
                                  NULL,
                                  NULL,
                                  D3D11_SDK_VERSION,
                                  &scd,
                                  &swapchain,
                                  &dev,
                                  NULL,
                                  &devcon);
}
```

If the comments in the code are good enough for you, excellent. Otherwise, I've described each of these commands below.

DXGI_SWAP_CHAIN_DESC scd;

There are certain factors in both beginning and advanced game programming which require certain information to be fed into Direct3D from the start. There are plenty of these, but we will only go into a few of them here.

For now, DXGI_SWAP_CHAIN_DESC is a struct whose members will contain the description of our swap chain. We will go over the ones we need, and cover new members as they come up throughout the tutorial.

ZeroMemory(&scd, sizeof(DXGI_SWAP_CHAIN_DESC));

We use ZeroMemory() to quickly initialize the entire scd struct to NULL. That way we don't have to go through every member of the struct and initialize them individually.

scd.BufferCount = 1;

This member contains the number of back buffers to use on our swap chain. We'll only be using one back buffer and one front buffer, so we'll set this value to 1. We could use more, but 1 will probably be enough for anything we'll do.

scd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;

This next member we will use to set the format of the colors. On the front and back buffers, each pixel is stored by color. This value determines what format that data is stored in.

Here, we set Format to be DXGI_FORMAT_R8G8B8A8_UNORM. This is a coded flag which indicates the format. You can learn more about formats here, but you won't need to change this for now.

scd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;

These names just keep getting longer and longer! This member tells how we intend to use our swap chain. This table has two commonly-used values for this. Like many flag-values, these flags can be ORed together.

Value	Description
DXGI_USAGE_RENDER_TARGET_OUTPUT	This value is used when you wish to draw graphics into the back buffer.
DXGI_USAGE_SHARED	Typically, when a device creates a buffer, only that device can work with it. This value allows it to be shared across multiple device objects.

scd.OutputWindow = hWnd;

This value sets the handle of the window Direct3D should draw in. We'll just use the same hWnd we've always been using.

scd.SampleDesc.Count = 1;

This member is used to tell Direct3D how to perform multisample anti-aliased (MSAA) rendering. If you are a next-gen game enthusiast, you have probably heard of this. Basically, anti-aliasing renders the edges of shapes smoothly by blending each pixel slightly with the surounding pixels.

Multisample Anti-Alias (MSAA)

The effect is shown in the image above. On the left, you can see the line has created a stair-like effect. On the right, the image is a little bit smoothed because Direct3D blended the pixels.

This value tells Direct3D how much detail should be put into anti-aliasing, the higher the number the better. Direct3D 11 video cards are guaranteed to support up to 4 here, but the minimum is 1.

scd.Windowed = TRUE;

When we run Direct3D in a window like we are now, this value is set to TRUE. Otherwise, it is set to FALSE for full-screen mode.

Note: There are other changes you will need to make before going full-screen. Changing this one value will not make your application properly full-screen, unfortunately. Patience for now.

scd.Flags

While there are no flags being used in the sample code yet, It'll be good to cover them here. They are described in this table.

D3D11CreateDeviceAndSwapChain()

This is a big function, but in actual fact, it is quite simple. Most of the parameters will probably stay the same in every game you write.

What this function does is create the device, device context, and swap chain COM objects. Once we've created them, we can use them to perform actual rendering.

Let's start by looking at the prototype of the function.

```
HRESULT D3D11CreateDeviceAndSwapChain(
    IDXGIAdapter *pAdapter,
    D3D_DRIVER_TYPE DriverType,
    HMODULE Software,
    UINT Flags,
    D3D_FEATURE_LEVEL *pFeatureLevels,
    UINT FeatureLevels,
    UINT SDKVersion,
    DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
    IDXGISwapChain **ppSwapChain,
    ID3D11Device **ppDevice,
    D3D_FEATURE_LEVEL *pFeatureLevel,
    ID3D11DeviceContext **ppDeviceContext);
```

Now let's go into the parameters of this function. They are all simple, so this will go fast.

IDXGIAdapter *pAdapter,

This is a value that indicates what graphics adapter Direct3D should use. A graphics adapter typically refers to a GPU and its video memory, digital-to-analog converter, etc.

We could get detailed here and try to find the best graphics card available, but we'll let DXGI take care of that for us (because in most cases there's only one). To tell DXGI that it needs to decide, we use NULL in this parameter, indicating the default adapter.

D3D_DRIVER_TYPE DriverType,

This parameter is used to determine whether Direct3D should use hardware or software for rendering. There are a number of flags you can use to determine this. They are listed in this table.

Flag	Description
D3D_DRIVER_TYPE_HARDWARE	The obviously best choice. This uses the advanced GPU hardware for rendering.
D3D_DRIVER_TYPE_REFERENCE	Not everyone has hardware that runs Direct3D 11. If you are such a person, and wish to run Direct3D 11 features, you will need to use this flag. It is slow (veerry slow), but it will at least run. You can use it until you have a Direct3D 11 graphics card.
D3D_DRIVER_TYPE_SOFTWARE	You can optionally build your own software-based rendering engine. This is usually very slow, but you can do it.
D3D_DRIVER_TYPE_WARP	WARP is a high performance software engine that runs older Direct3D features. You can find out about how to use it here.
D3D_DRIVER_TYPE_NULL	Use this flag if you don't intend to render. Direct3D can be used for other things than rendering, such as accessing the power of the GPU for non-graphics programs.

HMODULE Software,

We won't get into this parameter. It is used with the flag D3D_DRIVER_TYPE_SOFTWARE, to set the software code. It's very slow, so we won't get into it.

UINT Flags,

Flags! Flags are easy! Here we have a few flag values which can alter how Direct3D runs. These flags can be ORed together. Fortunately, we don't need any of these flags to get things going. The flags are listed in this table here.

Flag	Description
D3D11_CREATE_DEVICE_SINGLETHREADED	Multithreaded rendering is enabled by default. To not allow it, use this flag.
D3D11_CREATE_DEVICE_DEBUG	Enables debugging. You may have to enable your application's

	debugging in the DirectX Control Panel as well. Debug messages are displayed in the compiler's output window.
D3D11_CREATE_DEVICE_SWITCH_TO_REF	It can be handy to switch from hardware mode to reference mode during runtime for debugging reasons. This allows your program to do that.
D3D11_CREATE_DEVICE_BGRA_SUPPORT	This allows Direct2D to work with Direct3D. Direct2D is a separate graphics library that renders 2D images only.

D3D_FEATURE_LEVEL *pFeatureLevels,

Each major version of Direct3D has a series of video card features that are required. If you know what version your hardware meets the requirements of, you can more easily understand the capabilities of the hardware (given that your customers will have different video cards).

This parameter allows you to create a list of feature levels. This list tells Direct3D what features you are expecting your program to work with.

For this tutorial you will need a Direct3D 11 video card, and so we will not get into this parameter. We can set it to NULL, and we won't have to worry about it.

UINT FeatureLevels,

This parameter indicates how many feature levels you had in your list. We'll just put NULL.

UINT SDKVersion,

This parameter is always the same: D3D11_SDK_VERSION. Why is this? Well, it really only matters for compatibility on other machines. Each machine will usually have varying minor versions of DirectX. This tells the user's DirectX which version you developed your game for. The user's DirectX can then look back and properly execute your program without implementing the changes that have occurred since then.

In different versions of the SDK, this value returns different numbers. Note that you should not change this value, as it causes confusion and isn't really needed anyway. Just use D3D11_SDK_VERSION, and everything will work out all right.

DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,

This is a pointer to the swap chain description struct. We just fill this with '&scd'.

IDXGISwapChain **ppSwapChain,

This is a pointer to a pointer to the swap chain object. This function will create the object for us, and the address of the object will be stored in this pointer. Easy! All the work is done for us!

All we do here is put the location of the pointer, '&swapchain'.

ID3D11Device **ppDevice,

This is a pointer to a pointer to the device object. We defined the pointer as 'dev', so we'll put '&dev' in this parameter.

Like the swap chain, this function will create the device and store the address in our pointer, 'dev'.

D3D_FEATURE_LEVEL *FeatureLevel,

More about feature levels. This is a pointer to a feature level variable. When the function is completed, the variable will be filled with the flag of the highest feature level that was found. This lets the programmer know what hardware is available for him to use. We'll just set this to NULL.

ID3D11DeviceContext **ppImmediateContext

This is a pointer to a pointer to the device context object. We defined this pointer as 'devcon', so we'll put '&devcon' in this parameter. It will then be filled with the address of the device context object.

Well! That was quite a function! Let's step back for a moment and look at what the whole thing looks like:

```
D3D11CreateDeviceAndSwapChain(NULL,
                              D3D_DRIVER_TYPE_HARDWARE,
                              NULL,
                              NULL,
                              NULL,
                              NULL,
                              D3D11_SDK_VERSION,
                              &scd,
                              &swapchain,
                              &dev,
                              NULL,
                              &devcon);
```

There, that wasn't so hard was it?

Now that we have Direct3D initialized, let's go ahead and close it down.

Closing Direct3D

Whenever Direct3D is created, it must be closed down. This is very simple to do.

We have just three commands in here:

```
// this is the function that cleans up Direct3D and COM
void CleanD3D()
{
    // close and release all existing COM objects
    swapchain->Release();
    dev->Release();
    devcon->Release();
}
```

Here, we call the Release() function from each of the three interfaces we created, dev, devcon and swapchain. No parameters, nothing special. Just cleans everything up.

Why? Well, let's just say it would be a bad thing to do otherwise. Basically, if you create a COM object but never close it, it will just keep on running in the background of the computer until your next reboot, even after the program itself closes. Bad. Especially bad if you have a lot of resources in your game. Releasing COM objects let's everything off the hook and allows Windows to take back it's memory.

The Finished Program

Wow! That was quite a start, but it will go downhill from here.

Let's take a look at what we just did. Following is the code we added to our program. The new parts are now in bold.

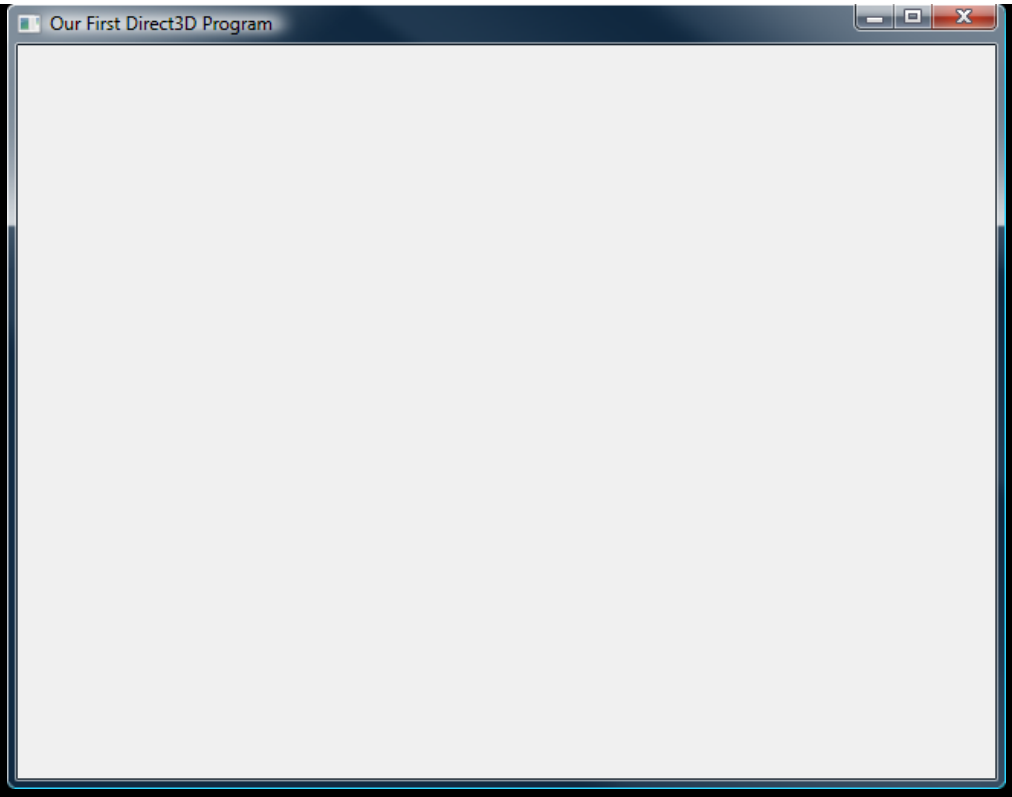
Before you run the code, there are a couple things to take note of.

First, if you don't yet have a DirectX 11 compatible video card, you will need to use reference mode instead of the hardware mode, or your program will crash.

Second, not all versions of Visual Studio (including the latest version) will locate the header files and library files correctly. If you get errors saying d3d11.h cannot be found, try these steps to fix them.

[Main.cpp]

And that's it! If you run this program you should get...a simple blank window, just like before. Only this time, Direct3D is running in the background!



Our First Direct3D Program

Ready...Set...

Well, are you ready to go on?

It isn't much yet, but you have begun the journey into the near-infinite depths of 3D game programming. You've created a window and gotten DirectX start up and close down.

At the end of each lesson, I'll ask a few questions and give a few exercises. If you can do these, you'll be ready for the lessons to come.

Questions

1. How do you call a COM-based function? Why do you have to "release" them?
2. What is the difference between hardware and reference modes?
3. What is the difference between the device and the device context?

Exercises

1. Try out various flag combinations and see what you can use (you won't be able to use most, yet).
2. Get debugging working. Comment a Release() function to see some debug warnings.

Next Lesson: Rendering Frames

GO! GO! GO!

© 2006-2020 DirectXTutorial.com. All Rights Reserved. Expand