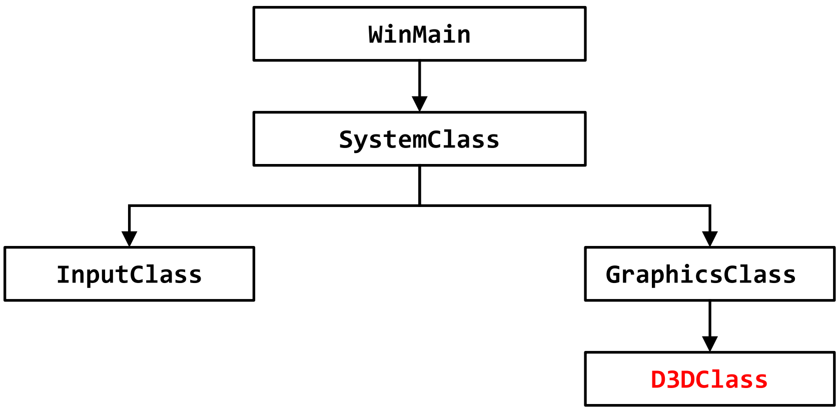
# Tutorial: Initializing Direct3D

*[Original script: www.rastertek.com]*

This tutorial will be the first introduction to working with DirectX 11. We will address how to initialize and shut down Direct3D as well as how to render to a window.

**Framework**



- D3DClass handles all the Direct3D system functions which are **encapsulated** in the GraphicsClass

**graphicsclass.h**

Here is the first change. We have taken out the include for windows.h and instead included the new d3dclass.h.

```
~~#include "windows.h"~~
#include "d3dclass.h"

class GraphicsClass
{
```

And the second change is the new private pointer to the D3DClass, which we have called m_D3D. In case you were wondering we use the prefix m_ on all class variables. That way, we can remember quickly what variables members of the class are and which are not.

```
private:
		D3DClass* m_D3D;
};
```

**graphicsclass.cpp**

If you'll remember from the previous tutorial, this class was entirely empty with no code in it at all. Now that we have a D3DClass member, we will start to fill out some code inside the GraphicsClass to initialize and shutdown the D3DClass object. We will also add calls to BeginScene and EndScene in the Render function so that we are now drawing to the window using Direct3D.

So, the very first change is in the class constructor. Here we initialize the pointer to null for safety reasons as we do with all class pointers.

```
GraphicsClass::GraphicsClass()
{
		m_D3D = 0;
}
```

The second change is in the Initialize function inside the GraphicsClass. Here, we create the D3DClass object and then call the D3DClass Initialize function. We send this function the screen width, screen height, handle to the window, and the four global variables from the Graphicsclass.h file. The D3DClass will use all these variables to setup the Direct3D system. We'll go into more detail about that once we look at the d3dclass.cpp file.

```
bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
	bool result;

	// Create the Direct3D object.
	m_D3D = new D3DClass;
	if(!m_D3D)
	{
		return false;
	}

	// Initialize the Direct3D object.
	result = m_D3D->Initialize(screenWidth, screenHeight, VSYNC_ENABLED, hwnd, FULL_SCREEN,
		SCREEN_DEPTH, SCREEN_NEAR);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize Direct3D", L"Error", MB_OK);
		return false;
	}

	return true;
}
```

The next change is in the Shutdown function in the GraphicsClass. Shut down of all graphics objects occur here so we have placed the D3DClass shutdown in this function. Note that we check to see if the pointer was initialized or not. If it wasn’t, we can assume it was never set up and not try to shut it down. That is why it is important to set all the pointers to null in the class constructor. If it does find the pointer has been initialized, then it will attempt to shut down the D3DClass and then clean up the pointer afterwards.

```
void GraphicsClass::Shutdown()
{
	if(m_D3D)
	{
		m_D3D->Shutdown();
		delete m_D3D;
		m_D3D = 0;
	}

	return;
}
```

The Frame function has been updated so that it now calls the Render function each frame.

```
bool GraphicsClass::Frame()
{
	bool result;

	// Render the graphics scene.
	result = Render();
	if(!result)
	{
		return false;
	}

	return true;
}
```

The final change to this class is in the Render function. We call the D3D object to clear the screen to a grey color. After that we call EndScene so that the grey color is presented to the window.

```
bool GraphicsClass::Render()
{
	// Clear the buffers to begin the scene.
	m_D3D->BeginScene(0.5f, 0.5f, 0.5f, 1.0f);
```

```
		// Present the rendered scene to the screen.
		m_D3D->EndScene();

		return true;
}
```

Now let's look at the new D3DClass header file:

**d3dclass.h**

```
#ifndef _D3DCLASS_H_
#define _D3DCLASS_H_
```

First thing in the header is to specify the libraries to link when using this object module. These libraries contain all the Direct3D functionality for setting up and drawing 3D graphics in DirectX as well as tools to interface with the hardware on the computer to obtain information about the **refresh rate** of the monitor, the video card being used, and so forth. You will notice that some DirectX 10 libraries are still used, this is because those libraries were never upgraded for DirectX 11 as their functionality did not need to change.

```
// Linking
#pragma comment(lib, "dxgi.lib")
#pragma comment(lib, "d3d11.lib")
#pragma comment(lib, "d3dx11.lib")
#pragma comment(lib, "d3dx10.lib")
```

The next thing we do is include the headers for those libraries that we are linking to this object module as well as headers for DirectX type definitions and such.

```
// DirectX includes
#include <dxgi.h>
#include <d3dcommon.h>
#include <d3d11.h>
#include <d3dx10math.h>
```

The class definition for the D3DClass is kept as simple as possible here. It has the regular constructor, copy constructor, and destructor. Then more importantly it has the Initialize and Shutdown function. This will be what we are mainly focused on in this tutorial. Other than that, we have a couple helper functions which aren't important to this tutorial and several private member variables that will be looked at when we examine the d3dclass.cpp file. For now, just realize the Initialize and Shutdown functions are what concerns us.

```
class D3DClass
{
public:
		D3DClass();
		D3DClass(const D3DClass&);
		~D3DClass();

		bool Initialize(int, int, bool, HWND, bool, float, float);
		void Shutdown();

		void BeginScene(float, float, float, float);
		void EndScene();

		ID3D11Device* GetDevice();
		ID3D11DeviceContext* GetDeviceContext();

		void GetProjectionMatrix(D3DXMATRIX&);
		void GetWorldMatrix(D3DXMATRIX&);
		void GetOrthoMatrix(D3DXMATRIX&);

		void GetVideoCardInfo(char*, int&);

private:
		bool m_vsync_enabled;
		int m_videoCardMemory;
		char m_videoCardDescription[128];
		IDXGISwapChain* m_swapChain;
		ID3D11Device* m_device;
		ID3D11DeviceContext* m_deviceContext;
		ID3D11RenderTargetView* m_renderTargetView;
		ID3D11Texture2D* m_depthStencilBuffer;
		ID3D11DepthStencilState* m_depthStencilState;
		ID3D11DepthStencilView* m_depthStencilView;
		ID3D11RasterizerState* m_rasterState;
		D3DXMATRIX m_projectionMatrix;
		D3DXMATRIX m_worldMatrix;
		D3DXMATRIX m_orthoMatrix;
};

#endif
```

For those familiar with Direct3D already you may notice we don't have a **view matrix** variable in this class. The reason being is that we will be putting it in a camera class that we will be looking at in future tutorials.

**d3dclass.cpp**

```
#include "d3dclass.h"
```

So, like most classes we begin with initializing all the member pointers to null in the class constructor. All pointers from the header file have all been accounted for here.

```
D3DClass::D3DClass()
{
		m_swapChain = 0;
		m_device = 0;
		m_deviceContext = 0;
		m_renderTargetView = 0;
		m_depthStencilBuffer = 0;
		m_depthStencilState = 0;
		m_depthStencilView = 0;
		m_rasterState = 0;
}

D3DClass::D3DClass(const D3DClass& other)
{
}


D3DClass::~D3DClass()
{
}
```

The Initialize function is what does the entire setup of Direct3D for DirectX 11. We have placed all the code necessary in here as well as some extra stuff that will facilitate future tutorials. We could have simplified it and taken out some items but it is probably better to get all of this covered in a single tutorial dedicated to it.

The screenWidth and screenHeight variables that are given to this function are the width and height of the window we created in the SystemClass. Direct3D will use these to initialize and use the same window dimensions. The hwnd variable is a handle to the window. Direct3D will need this handle to access the window previously created. The fullscreen variable is whether we are running in windowed mode or fullscreen. Direct3D needs this as well for creating the window with the correct settings. The screenDepth and screenNear variables are the depth settings for our 3D environment that will be rendered in the window. The vsync variable indicates if we want Direct3D to render according to the users monitor refresh rate or to just go as fast as possible.

```
bool D3DClass::Initialize(int screenWidth, int screenHeight, bool vsync, HWND hwnd, bool fullscreen, float screenDepth,
		float screenNear)
{
		HRESULT result;
		IDXGIFactory* factory;
```

```
	IDXGIAdapter* adapter;
	IDXGIOutput* adapterOutput;
	unsigned int numModes, i, numerator, denominator, stringLength;
	DXGI_MODE_DESC* displayModeList;
	DXGI_ADAPTER_DESC adapterDesc;
	int error;
	DXGI_SWAP_CHAIN_DESC swapChainDesc;
	D3D_FEATURE_LEVEL featureLevel;
	ID3D11Texture2D* backBufferPtr;
	D3D11_TEXTURE2D_DESC depthBufferDesc;
	D3D11_DEPTH_STENCIL_DESC depthStencilDesc;
	D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc;
	D3D11_RASTERIZER_DESC rasterDesc;
	D3D11_VIEWPORT viewport;
	float fieldOfView, screenAspect;

	// Store the vsync setting.
	m_vsync_enabled = vsync;
```

Before we can initialize Direct3D, we have to get the refresh rate from the video card/monitor. Each computer may be slightly different so we will need to query for that information. We query for the numerator and denominator values and then pass them to DirectX during the setup and it will calculate the proper refresh rate. If we don't do this and just set the refresh rate to a default value which may not exist on all computers then DirectX will respond by performing a blit instead of a buffer flip which will degrade performance and give us annoying errors in the debug output.

```
	// Create a DirectX graphics interface factory.
	result = CreateDXGIFactory(uuidof(IDXGIFactory), (void**)&factory);
	if(FAILED(result))
	{
		return false;
	}

	// Use the factory to create an adapter for the primary graphics interface (video card).
	result = factory->EnumAdapters(0, &adapter);
	if(FAILED(result))
	{
		return false;
	}

	// Enumerate the primary adapter output (monitor).
	result = adapter->EnumOutputs(0, &adapterOutput);
	if(FAILED(result))
	{
		return false;
	}

     // Get the number of modes that fit the DXGI_FORMAT_R8G8B8A8_UNORM display format for the adapter output (monitor).
	result = adapterOutput->GetDisplayModeList(DXGI_FORMAT_R8G8B8A8_UNORM,
		DXGI_ENUM_MODES_INTERLACED, &numModes, NULL);
	if(FAILED(result))
	{
		return false;
	}

	// Create a list to hold all the possible display modes for this monitor/video card combination.
	displayModeList = new DXGI_MODE_DESC[numModes];
	if(!displayModeList)
	{
		return false;
	}

	// Now fill the display mode list structures.
	result = adapterOutput->GetDisplayModeList(DXGI_FORMAT_R8G8B8A8_UNORM,
		DXGI_ENUM_MODES_INTERLACED, &numModes, displayModeList);
	if(FAILED(result))
	{
		return false;
	}

	// Now go through all the display modes and find the one that matches the screen width and height.
	// When a match is found store the numerator and denominator of the refresh rate for that monitor.
	for(i=0; i<numModes; i++)
	{
		if(displayModeList[i].Width == (unsigned int)screenWidth)
		{
			if(displayModeList[i].Height == (unsigned int)screenHeight)
			{
				numerator = displayModeList[i].RefreshRate.Numerator;
				denominator = displayModeList[i].RefreshRate.Denominator;
			}
		}
	}
```

We now have the numerator and denominator for the refresh rate. The last thing we will retrieve using the adapter is the name of the video card and the amount of memory on the video card.

```
	// Get the adapter (video card) description.
	result = adapter->GetDesc(&adapterDesc);
	if(FAILED(result))
	{
		return false;
	}

	// Store the dedicated video card memory in megabytes.
	m_videoCardMemory = (int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);

	// Convert the name of the video card to a character array and store it.
	error = wcstombs_s(&stringLength, m_videoCardDescription, 128, adapterDesc.Description, 128);
	if(error != 0)
	{
		return false;
	}
```

Now that we have stored the numerator and denominator for the refresh rate and the video card information, we can release the structures and interfaces used to get that information.

```
	// Release the display mode list.
	delete [] displayModeList;
	displayModeList = 0;

	// Release the adapter output.
	adapterOutput->Release();
	adapterOutput = 0;

	// Release the adapter.
	adapter->Release();
	adapter = 0;

	// Release the factory.
	factory->Release();
	factory = 0;
```

Now that we have the refresh rate from the system we can start the DirectX initialization. The first thing we'll do is fill out the description of the **swap chain**. The swap chain is the **front and back buffer** to which the graphics will be drawn. Generally, you use a single back buffer, do all your drawing to it, and then **swap** it to the front buffer which then displays on the user's screen. That is why it is called a swap chain.

```
	// Initialize the swap chain description.
	ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));

	// Set to a single back buffer.
	swapChainDesc.BufferCount = 1;
```

```
// Set the width and height of the back buffer.
swapChainDesc.BufferDesc.Width = screenWidth;
swapChainDesc.BufferDesc.Height = screenHeight;

// Set regular 32-bit surface for the back buffer.
swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
```

The next part of the description of the swap chain is the **refresh rate**. The refresh rate is how many times a second it draws the back buffer to the front buffer. If vsync is set to true in our graphicsclass.h header then this will lock the refresh rate to the system settings (for example 60hz). That means it will only draw the screen 60 times a second (or higher if the system refresh rate is more than 60). However, if we set vsync to false then it will draw the screen as many times a second as it can, however this can cause some visual artifacts.

```
// Set the refresh rate of the back buffer.
if(m_vsync_enabled)
{
	swapChainDesc.BufferDesc.RefreshRate.Numerator = numerator;
	swapChainDesc.BufferDesc.RefreshRate.Denominator = denominator;
}
else
{
	swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;
	swapChainDesc.BufferDesc.RefreshRate.Denominator = 1;
}

// Set the usage of the back buffer.
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;

// Set the handle for the window to render to.
swapChainDesc.OutputWindow = hwnd;

// Turn multisampling off.
swapChainDesc.SampleDesc.Count = 1;
swapChainDesc.SampleDesc.Quality = 0;

// Set to full screen or windowed mode.
if(fullscreen)
{
	swapChainDesc.Windowed = false;
}
else
{
	swapChainDesc.Windowed = true;
}

// Set the scan line ordering and scaling to unspecified.
swapChainDesc.BufferDesc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
swapChainDesc.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;

// Discard the back buffer contents after presenting.
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;

// Don't set the advanced flags.
swapChainDesc.Flags = 0;
```

After setting up the swap chain description, we also need to setup one more variable called the feature level. This variable tells DirectX what version we plan to use. Here we set the feature level to 11.0 which is DirectX 11. You can set this to 10 or 9 to use a lower level version of DirectX if you plan on supporting multiple versions or running on lower end hardware.

```
// Set the feature level to DirectX 11.
featureLevel = D3D_FEATURE_LEVEL_11_0;
```

Now that the swap chain description and feature level have been filled out, we can create the swap chain, the Direct3D device, and the Direct3D **device context**. The Direct3D device and Direct3D device context are very important, they are the interface to all the Direct3D functions. We will use the device and device context for almost everything from this point forward.

Those of you reading this who are familiar with the previous versions of DirectX will recognize the Direct3D device but will be unfamiliar with the new Direct3D device context. Basically, they took the functionality of the Direct3D device and split it up into two different devices so you need to use both now.

Note that if the user does not have a DirectX 11 video card this function call will fail to create the device and device context. Also if you are testing DirectX 11 functionality yourself and don't have a DirectX 11 video card then you can replace D3D_DRIVER_TYPE_HARDWARE with D3D_DRIVER_TYPE_REFERENCE and DirectX will use your CPU to draw instead of the video card hardware. Note that this runs 1/1000 the speed but it is good for people who don't have DirectX 11 video cards yet on all their machines.

```
// Create the swap chain, Direct3D device, and Direct3D device context.
result = D3D11CreateDeviceAndSwapChain(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, 0,
        &featureLevel, 1, D3D11_SDK_VERSION, &swapChainDesc, &m_swapChain, &m_device, NULL,
        &m_deviceContext);
if(FAILED(result))
{
        return false;
}
```

Sometimes this call to create the device will fail if the primary video card is not compatible with DirectX 11. Some machines may have the primary card as a DirectX 10 video card and the secondary card as a DirectX 11 video card. Also, some hybrid graphics cards work that way with the primary being the low power Intel card and the secondary being the high power Nvidia card. To get around this you will need to not use the default device and instead enumerate all the video cards in the machine and have the user choose which one to use and then specify that card when creating the device.

Now that we have a swap chain, we need to get a pointer to the back buffer and then attach it to the swap chain. We'll use the CreateRenderTargetView function to attach the back buffer to our swap chain.

```
// Get the pointer to the back buffer.
result = m_swapChain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&backBufferPtr);
if(FAILED(result))
{
        return false;
}

// Create the render target view with the back buffer pointer.
result = m_device->CreateRenderTargetView(backBufferPtr, NULL, &m_renderTargetView);
if(FAILED(result))
{
        return false;
}

// Release pointer to the back buffer as we no longer need it.
backBufferPtr->Release();
backBufferPtr = 0;
```

We will also need to set up a **depth buffer** description. We'll use this to create a depth buffer so that our polygons can be rendered properly in 3D space. At the same time we will attach a **stencil buffer** to our depth buffer. The stencil buffer can be used to achieve effects such as **motion blur**, **volumetric shadows**, and other things.

```
// Initialize the description of the depth buffer.
ZeroMemory(&depthBufferDesc, sizeof(depthBufferDesc));

// Set up the description of the depth buffer.
depthBufferDesc.Width = screenWidth;
depthBufferDesc.Height = screenHeight;
depthBufferDesc.MipLevels = 1;
depthBufferDesc.ArraySize = 1;
depthBufferDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
```

```
	depthBufferDesc.SampleDesc.Count = 1;
	depthBufferDesc.SampleDesc.Quality = 0;
	depthBufferDesc.Usage = D3D11_USAGE_DEFAULT;
	depthBufferDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
	depthBufferDesc.CPUAccessFlags = 0;
	depthBufferDesc.MiscFlags = 0;
```

Now we create the depth/stencil buffer using that description. You will notice we use the CreateTexture2D function to make the buffers, hence the buffer is just a 2D texture. The reason for this is that once your polygons are sorted and then rasterized, they just end up being **colored pixels** in this 2D buffer. Then this 2D buffer is drawn to the screen.

```
	// Create the texture for the depth buffer using the filled out description.
	result = m_device->CreateTexture2D(&depthBufferDesc, NULL, &m_depthStencilBuffer);
	if(FAILED(result))
	{
		return false;
	}
```

Now we need to setup the depth stencil description. This allows us to control what type of depth test Direct3D will do for each pixel.

```
	// Initialize the description of the stencil state.
	ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));

	// Set up the description of the stencil state.
	depthStencilDesc.DepthEnable = true;
	depthStencilDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
	depthStencilDesc.DepthFunc = D3D11_COMPARISON_LESS;

	depthStencilDesc.StencilEnable = true;
	depthStencilDesc.StencilReadMask = 0xFF;
	depthStencilDesc.StencilWriteMask = 0xFF;

	// Stencil operations if pixel is front-facing.
	depthStencilDesc.FrontFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
	depthStencilDesc.FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_INCR;
	depthStencilDesc.FrontFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
	depthStencilDesc.FrontFace.StencilFunc = D3D11_COMPARISON_ALWAYS;

	// Stencil operations if pixel is back-facing.
	depthStencilDesc.BackFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
	depthStencilDesc.BackFace.StencilDepthFailOp = D3D11_STENCIL_OP_DECR;
	depthStencilDesc.BackFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
	depthStencilDesc.BackFace.StencilFunc = D3D11_COMPARISON_ALWAYS;
```

With the description filled out we can now create a depth stencil state.

```
	// Create the depth stencil state.
	result = m_device->CreateDepthStencilState(&depthStencilDesc, &m_depthStencilState);
	if(FAILED(result))
	{
		return false;
	}
```

With the created depth stencil state we can now set it so that it takes effect. Notice we use the device context to set it.

```
	// Set the depth stencil state.
	m_deviceContext->OMSetDepthStencilState(m_depthStencilState, 1);
```

The next thing we need to create is the description of the view of the depth stencil buffer. We do this so that Direct3D knows to use the depth buffer as a depth stencil texture. After filling out the description we then call the function CreateDepthStencilView to create it.

```
	// Initailze the depth stencil view.
	ZeroMemory(&depthStencilViewDesc, sizeof(depthStencilViewDesc));

	// Set up the depth stencil view description.
	depthStencilViewDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
	depthStencilViewDesc.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
	depthStencilViewDesc.Texture2D.MipSlice = 0;

	// Create the depth stencil view.
	result = m_device->CreateDepthStencilView(m_depthStencilBuffer, &depthStencilViewDesc, &m_depthStencilView);
	if(FAILED(result))
	{
		return false;
	}
```

With that created we can now call OMSetRenderTargets. This will bind the **render target view** and the depth stencil buffer to the output **render pipeline**. This way the graphics that the pipeline renders will get drawn to our back buffer that we previously created. With the graphics written to the back buffer we can then swap it to the front and display our graphics on the user's screen.

```
	// Bind the render target view and depth stencil buffer to the output render pipeline.
	m_deviceContext->OMSetRenderTargets(1, &m_renderTargetView, m_depthStencilView);
```

Now that the render targets are setup, we can continue on to some extra functions that will give us more control over our scenes for future tutorials. First thing is we'll create is a **rasterizer state**. This will give us control over how polygons are rendered. We can do things like make our scenes render in **wireframe mode** or have DirectX draw both **the front and back faces of polygons**. By default, DirectX already has a rasterizer state set up and working the exact same as the one below, but you have no control to change it unless you set up one yourself.

```
	// Setup the raster description which will determine how and what polygons will be drawn.
	rasterDesc.AntialiasedLineEnable = false;
	rasterDesc.CullMode = D3D11_CULL_BACK;
	rasterDesc.DepthBias = 0;
	rasterDesc.DepthBiasClamp = 0.0f;
	rasterDesc.DepthClipEnable = true;
	rasterDesc.FillMode = D3D11_FILL_SOLID;
	rasterDesc.FrontCounterClockwise = false;
	rasterDesc.MultisampleEnable = false;
	rasterDesc.ScissorEnable = false;
	rasterDesc.SlopeScaledDepthBias = 0.0f;

	// Create the rasterizer state from the description we just filled out.
	result = m_device->CreateRasterizerState(&rasterDesc, &m_rasterState);
	if(FAILED(result))
	{
		return false;
	}

	// Now set the rasterizer state.
	m_deviceContext->RSSetState(m_rasterState);
```

The **viewport** also needs to be setup so that Direct3D can map **clip space coordinates** to the **render target space**. Set this to be the entire size of the window.

```
	// Setup the viewport for rendering.
	viewport.Width = (float)screenWidth;
	viewport.Height = (float)screenHeight;
	viewport.MinDepth = 0.0f;
	viewport.MaxDepth = 1.0f;
	viewport.TopLeftX = 0.0f;
	viewport.TopLeftY = 0.0f;

	// Create the viewport.
	m_deviceContext->RSSetViewports(1, &viewport);
```

Now we will create the **projection matrix**. The projection matrix is used to translate the 3D scene into the 2D viewport space that we previously created. We will need to keep a copy of this matrix so that we can pass it to our **shaders** that will be used to render our scenes.

```
	// Setup the projection matrix.
	fieldOfView = (float)D3DX_PI / 4.0f;
	screenAspect = (float)screenWidth / (float)screenHeight;

	// Create the projection matrix for 3D rendering.
	D3DXMatrixPerspectiveFovLH(&m_projectionMatrix, fieldOfView, screenAspect, screenNear,
		screenDepth);
```

We will also create another matrix called the **world matrix**. This matrix is used to convert the vertices of our objects into vertices in the 3D scene. This matrix will also be used to **rotate**, **translate**, and **scale** our objects in 3D space. From the start we will just initialize the matrix to the **identity matrix** and keep a copy of it in this object. The copy will be needed to be passed to the shaders for rendering also.

```
	// Initialize the world matrix to the identity matrix.
	D3DXMatrixIdentity(&m_worldMatrix);
```

This is where you would generally create a **view matrix**. The view matrix is used to calculate the position of where we are looking at the scene from. You can think of it as a camera and you only view the scene through this camera. Because of its purpose we're going to create it in a camera class in later tutorials since logically it fits better there and just skip it for now.

And the final thing we will setup in the Initialize function is an **orthographic projection matrix**. This matrix is used for rendering 2D elements like user interfaces on the screen allowing us to skip the 3D rendering. You will see this used in later tutorials when we look at rendering **2D graphics** and fonts to the screen.

```
	// Create an orthographic projection matrix for 2D rendering.
	D3DXMatrixOrthoLH(&m_orthoMatrix, (float)screenWidth, (float)screenHeight, screenNear, screenDepth);

	return true;
}
```

The Shutdown function will release and clean up all the pointers used in the Initialize function, it's straight forward. However, before doing that we put in a call to force the swap chain to go into windowed mode first before releasing any pointers. If this is not done and you try to release the swap chain in full screen mode it will **throw some exceptions**. So, to avoid that happening we just always force windowed mode before shutting down Direct3D.

```
void D3DClass::Shutdown()
{
	// Before shutting down set to windowed mode or when you release the swap chain it will throw an exception.
	if(m_swapChain)
	{
		m_swapChain->SetFullscreenState(false, NULL);
	}

	if(m_rasterState)
	{
		m_rasterState->Release();
		m_rasterState = 0;
	}

	if(m_depthStencilView)
	{
		m_depthStencilView->Release();
		m_depthStencilView = 0;
	}

	if(m_depthStencilState)
	{
		m_depthStencilState->Release();
		m_depthStencilState = 0;
	}
	if(m_depthStencilBuffer)
	{
		m_depthStencilBuffer->Release();
		m_depthStencilBuffer = 0;
	}

	if(m_renderTargetView)
	{
		m_renderTargetView->Release();
		m_renderTargetView = 0;
	}

	if(m_deviceContext)
	{
		m_deviceContext->Release();
		m_deviceContext = 0;
	}

	if(m_device)
	{
		m_device->Release();
		m_device = 0;
	}

	if(m_swapChain)
	{
		m_swapChain->Release();
		m_swapChain = 0;
	}

	return;
}
```

In the D3DClass we have a couple helper functions. The first two are BeginScene and EndScene. BeginScene will be called whenever we are going to draw a new 3D scene at the beginning of each frame. All it does is initializes the buffers, so they are blank and ready to be drawn to. The other function is Endscene, it tells the swap chain to display our 3D scene once all the drawing has completed at the end of each frame.

```
void D3DClass::BeginScene(float red, float green, float blue, float alpha)
{
	float color[4];


	// Setup the color to clear the buffer to.
	color[0] = red;
	color[1] = green;
	color[2] = blue;
	color[3] = alpha;

	// Clear the back buffer.
	m_deviceContext->ClearRenderTargetView(m_renderTargetView, color);

	// Clear the depth buffer.
	m_deviceContext->ClearDepthStencilView(m_depthStencilView, D3D11_CLEAR_DEPTH, 1.0f, 0);

	return;
}

void D3DClass::EndScene()
{
	// Present the back buffer to the screen since rendering is complete.
	if(m_vsync_enabled)
	{
		// Lock to screen refresh rate.
		m_swapChain->Present(1, 0);
	}
	else
	{
		// Present as fast as possible.
		m_swapChain->Present(0, 0);
	}

	return;
}
```

These next functions simply get pointers to the Direct3D device and the Direct3D device context. These helper functions will be called by the framework often.

```
ID3D11Device* D3DClass::GetDevice()
{
	return m_device;
}


ID3D11DeviceContext* D3DClass::GetDeviceContext()
{
	return m_deviceContext;
}
```

The next three helper functions give copies of the **projection**, **world**, and **orthographic** matrices to calling functions. Most shaders will need these matrices for rendering so there needed to be an easy way for outside objects to get a copy of them. We won't call these functions in this tutorial but I'm just explaining why they are in the code.

```
void D3DClass::GetProjectionMatrix(D3DXMATRIX& projectionMatrix)
{
	projectionMatrix = m_projectionMatrix;
	return;
}


void D3DClass::GetWorldMatrix(D3DXMATRIX& worldMatrix)
{
	worldMatrix = m_worldMatrix;
	return;
}


void D3DClass::GetOrthoMatrix(D3DXMATRIX& orthoMatrix)
{
	orthoMatrix = m_orthoMatrix;
	return;
}
```

The last helper function returns by reference the name of the video card and the amount of dedicated memory on the video card. Knowing the video card name and amount of video memory can help in debugging on different configurations.

```
void D3DClass::GetVideoCardInfo(char* cardName, int& memory)
{
	strcpy_s(cardName, 128, m_videoCardDescription);
	memory = m_videoCardMemory;
	return;
}
```

**Summary**

So now we are finally able to initialize and shut down Direct3D as well as render a color to the window. Compiling and running the code will produce the same window as the last tutorial but Direct3D is initialized now, and the window is cleared to grey. Compiling and running the code will also show if your compiler is set up properly and if it can see the headers and libraries files from the DirectX SDK.

When you see a lot of C4005 warnings (due to the DirectX macro redefinitions) during the compilation, use

Project properties → C/C++ → Advanced → Disable Specific Warnings: 4005

Or

Add the follow code to all the headers that use DirectX libraries:

```
#pragma warning( disable : 4005)
```

**Exercise 1-3**

1. Change the clear(background) color to yellow.
2. Add a code that prints out the video card name and memory amount to a text file: “VideoInfo.txt”.