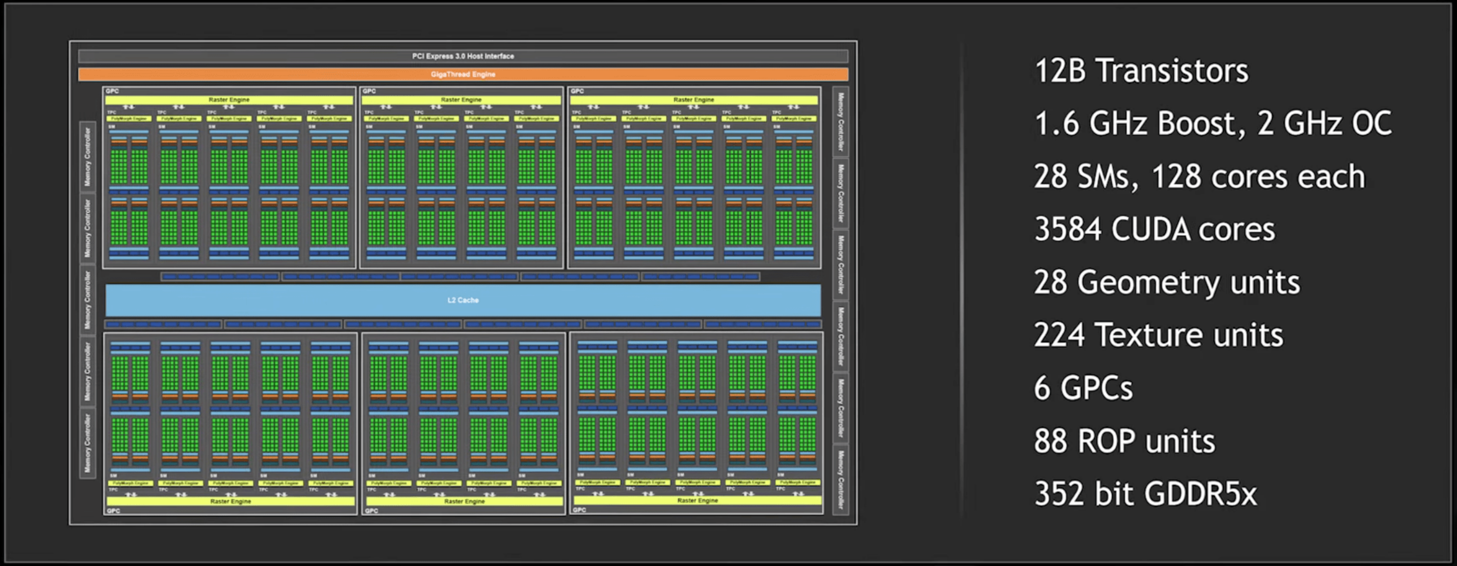
GRAPHICS PROGRAMMING I

HARDWARE RASTERIZATION PART I

DirectX: Introduction

- The last couple of weeks you've implemented your own software rasterizer. This rasterizer runs on the CPU. Just as with the ray tracer, a lot of the calculations can be done in parallel. If we had a lot of threads, we could accelerate our calculations. Hardware that has a lot of threads exists… the Graphics Processing Unit (GPU)!
    - For example: Nvidia 1080Ti has 28 Streaming Multiprocessors (SM). A SM can have a maximum of 2048 threads in flight. Hence, 28*2048 = 57,344 concurrent threads running (max) ☺



12B Transistors

1.6 GHz Boost, 2 GHz OC

28 SMs, 128 cores each

3584 CUDA cores

28 Geometry units

224 Texture units

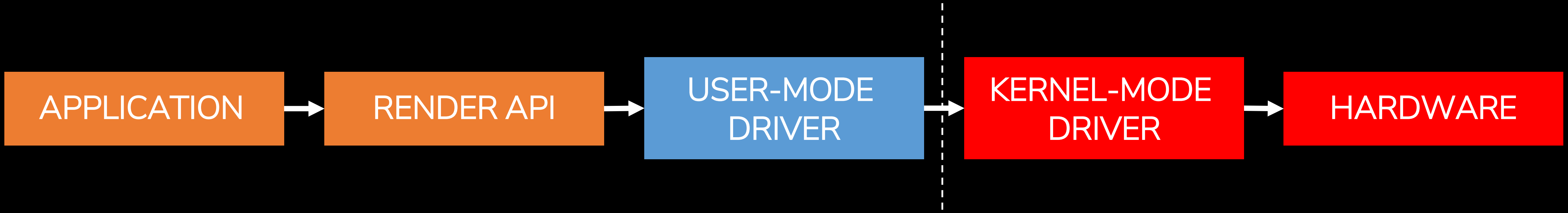
6 GPCs

88 ROP units

352 bit GDDR5x

DirectX: Introduction

- How do we communicate with the GPU? There are multiple options:
  - DirectX, OpenGl, Vulkan, Metal, LibGNM, etc. → 3D Rendering API's
  - CUDA, OpenCl, DirectCompute, etc. → GPGPU API's
- In the end, they are all Application Programming Interfaces (APIs) that simplify programming, hiding all the "nasty" hardware-software interfacing.



APPLICATION → RENDER API → USER-MODE DRIVER → KERNEL-MODE DRIVER → HARDWARE

- This interfacing is interesting, but very low-level! You need a good understanding of application programming interfaces, operating systems, drivers and hardware.
  - https://nouveau.freedesktop.org/wiki/IntroductoryCourse/
  - https://github.com/torvalds/linux/tree/master/drivers/gpu/drm/nouveau

DirectX: Introduction

- We’ll be using DirectX. DirectX is only supported on Windows, but the concepts are similar for other API’s. (https://alain.xyz/blog/comparison-of-modern-graphics-apis)
- DirectX is a collection of APIs for handling tasks related to multimedia:
  - DXGI – Manage low-level tasks like enumeration of hardware devices, presenting rendered frames to an output, controlling gamma and managing a full-screen transition. https://docs.microsoft.com/en-us/windows/win32/direct3ddxgi/d3d10-graphics-programming-guide-dxgi
  - Direct2D – Drawing 2D Graphics
  - Direct3D – Drawing 3D Graphics
  - DirectXMath – Math Library supporting SIMD
  - DirectCompute – GPGPU Computing
  - DirectWrite – Text Rendering
  - … → https://docs.microsoft.com/en-us/windows/win32/directx?redirectedfrom=MSDN
- DirectX evolved over time and some of the API’s got deprecated:
  - DirectInput – Input interface (alternative -> XInput)
  - DirectSound – Audio (alternative -> FMOD)
  - …

DirectX: Introduction

- We'll be using DirectX 11. Why don't we use DirectX 12?
  - Companies are the main target for Microsoft. They tend to do a lot of things themselves and they want full control. With DirectX 12 companies got the control. With DirectX 12 you must implement your own command lists and buffers, do resource management, etc. This reduces the driver overhead and allows for more efficient resource utilization. The downside is more code, and you need a better understanding of the hardware.
  - A lot of other API functionalities are discarded by companies as well. They tend to do the following stuff in-house:
    - Math functions
    - Render state encapsulation through effects
    - 3D model data
    - …
  - Luckily for us there is dedicated community that keeps these 'deprecated' tools/API's alive but be aware it may not be available when working at a company!
- You'll see that using DirectX 11 (on a beginner's level) isn't that hard after programming your own software rasterizer. A lot of the concepts are identical. You'll just have to get used to the API convention, learn which function calls you need, use some typical C++ techniques and learn how to handle data/resources.
- Let's explore the wonderful world of Graphics API's together ☺

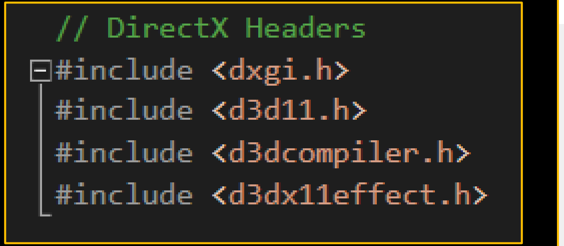
DirectX: Introduction

- DirectX Start Project
  https://github.com/Tomiha/GP1_2223_DirectX_Start

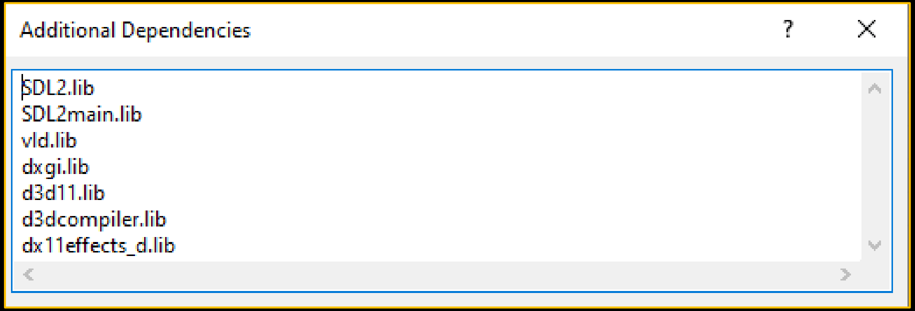
- Uses a Precompiled Header
- Includes required files and libs for DirectX

DirectX: Introduction

- Before we can start rendering something with DirectX 11, we must **initialize** it. All modern rendering API's have a **programmable rendering pipeline**. We'll have a look at the pipeline after the initialization process.
- Let's have a look at our new project!
  - Added the necessary libraries files:
    - d3d11.lib → DirectX 11 API
    - d3dcompiler → DirectX 11 shader compiler functionality
    - dxgi.lib → DXGI API
    - dx11effects.lib → External effect framework
  - Added the necessary includes linked to the libraries defined above.
  - Added the DirectXEffect framework. This is not part of the official DirectX11 SDK, but it will prove very useful.



```
// DirectX Headers
#include <dxgi.h>
#include <d3d11.h>
#include <d3dcompiler.h>
#include <d3dx11effect.h>
```

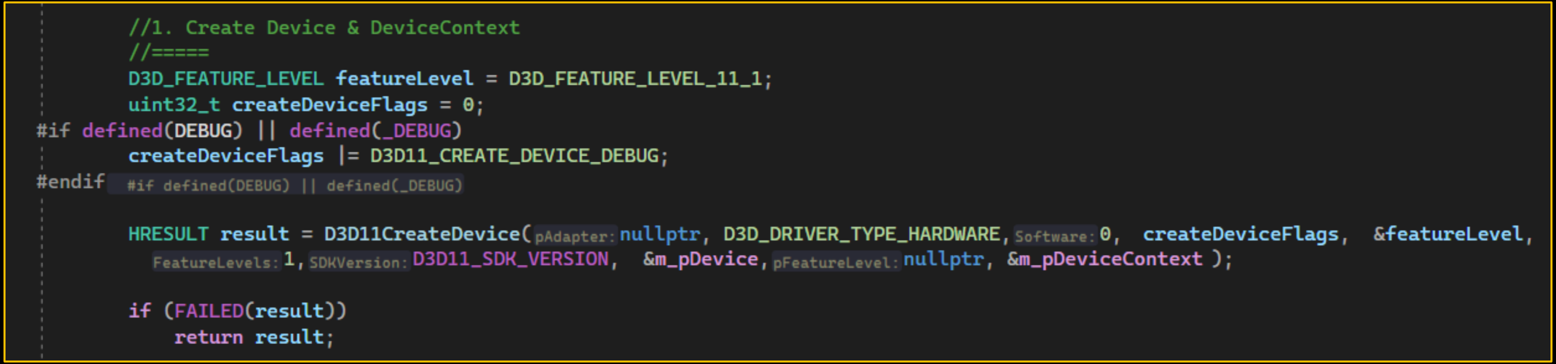


Additional Dependencies

```
SDL2.lib
SDL2main.lib
vld.lib
dxgi.lib
d3d11.lib
d3dcompiler.lib
dx11effects_d.lib
```

DirectX: Initialization

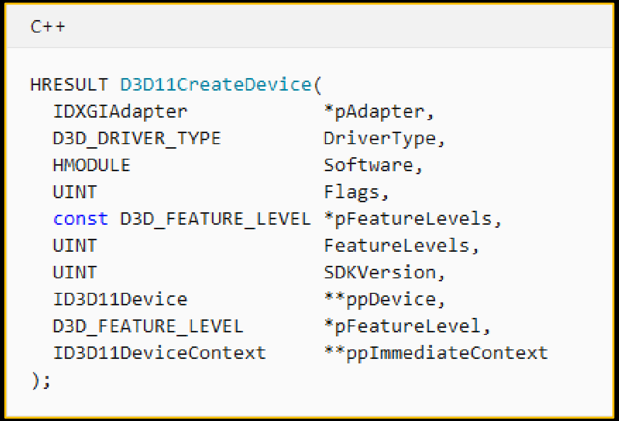
- Let’s initialize DirectX. Start by making a private function in the renderer called InitializeDirectX(), that is getting called in the constructor of the renderer.
- When using DirectX 11 you have two very important members, that are being created when you initialize DirectX:
  - Device → ID3D11Device* : represents the display adapter. It’s used to create resources and to enumerate the capabilities of a display adapter.
  - Device Context → ID3D11DeviceContext* : it contains the circumstance or setting in which a device is used. More specifically, a device context is used to set pipeline states and generate rendering commands using the resources owned by a device. This is NOT thread safe! https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-devices-intro
- We can create this device and device context with the following code:



```
	//1. Create Device & DeviceContext
	//=====
	D3D_FEATURE_LEVEL featureLevel = D3D_FEATURE_LEVEL_11_1;
	uint32_t createDeviceFlags = 0;
#if defined(DEBUG) || defined(_DEBUG)
	createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif #if defined(DEBUG) || defined(_DEBUG)

	HRESULT result = D3D11CreateDevice(pAdapter: nullptr, D3D_DRIVER_TYPE_HARDWARE, Software: 0,  createDeviceFlags,  &featureLevel,
	  FeatureLevels: 1, SDKVersion: D3D11_SDK_VERSION,  &m_pDevice, pFeatureLevel: nullptr, &m_pDeviceContext );

	if (FAILED(result))
	    return result;
```

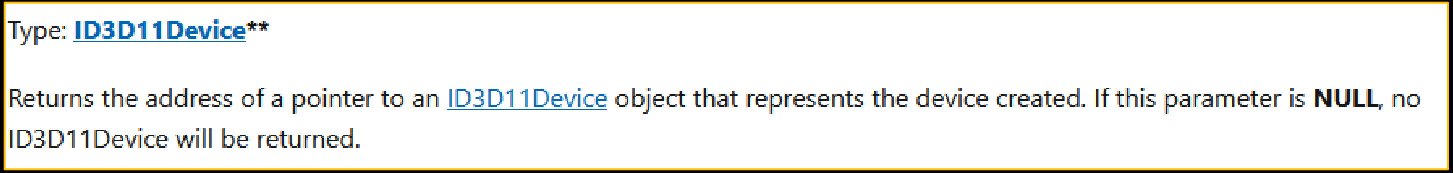
DirectX: Initialization

- When working with DirectX, make extensive use of MSDN and the documentation available!
- When looking at a function:
  - Pay attention to the parameters (input vs output).
  - Read the description of the parameters. It has other hyperlinks!
  - Check the result (return value - HRESULT) after calling the function!
    https://docs.microsoft.com/en-us/windows/win32/seccrypto/common-hresult-values



C++

```
HRESULT D3D11CreateDevice(
  IDXGIAdapter               *pAdapter,
  D3D_DRIVER_TYPE            DriverType,
  HMODULE                    Software,
  UINT                       Flags,
  const D3D_FEATURE_LEVEL    *pFeatureLevels,
  UINT                       FeatureLevels,
  UINT                       SDKVersion,
  ID3D11Device               **ppDevice,
  D3D_FEATURE_LEVEL          *pFeatureLevel,
  ID3D11DeviceContext        **ppImmediateContext
);
```



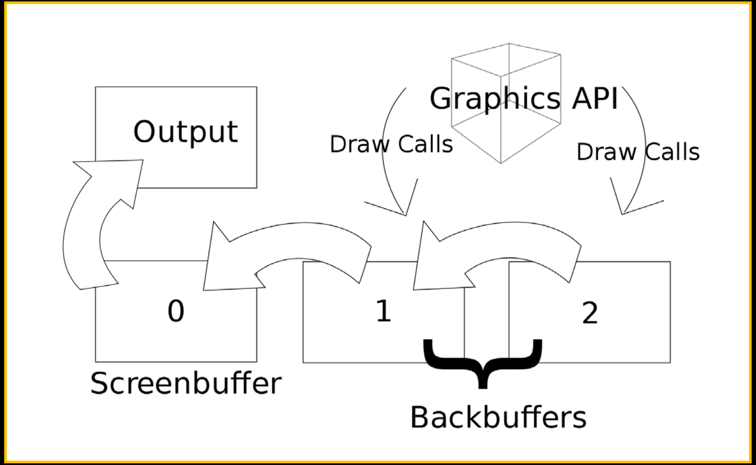
Type: **ID3D11Device****

Returns the address of a pointer to an ID3D11Device object that represents the device created. If this parameter is **NULL**, no ID3D11Device will be returned.

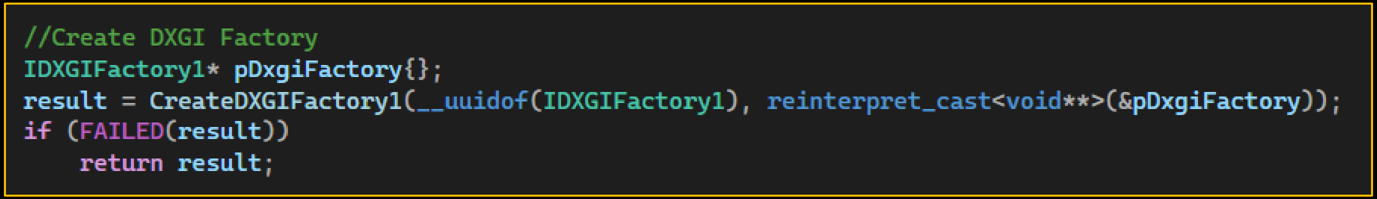
- Use resources as Google! ☺ Learn to solve problems yourself. There is a lot of information out there.

DirectX: Initialization

- Once we have our Device and DeviceContext we want to create our Swap Chain.
- What is a swap chain?
  - In a swap chain there are at least two buffers. The first buffer, the screen buffer or front buffer, is the buffer that is rendered to the output of the videocard. The remaining buffers are known as back buffers. ☺ Each time a new frame is displayed, the buffers swap place.
  - Why? If we would directly write to the screen, and if the buffer isn't locked, we would have artifacts since the monitor refresh rates are very slow in comparison with the rest of the computer! https://docs.microsoft.com/en-us/windows/win32/direct3d9/what-is-a-swap-chain-



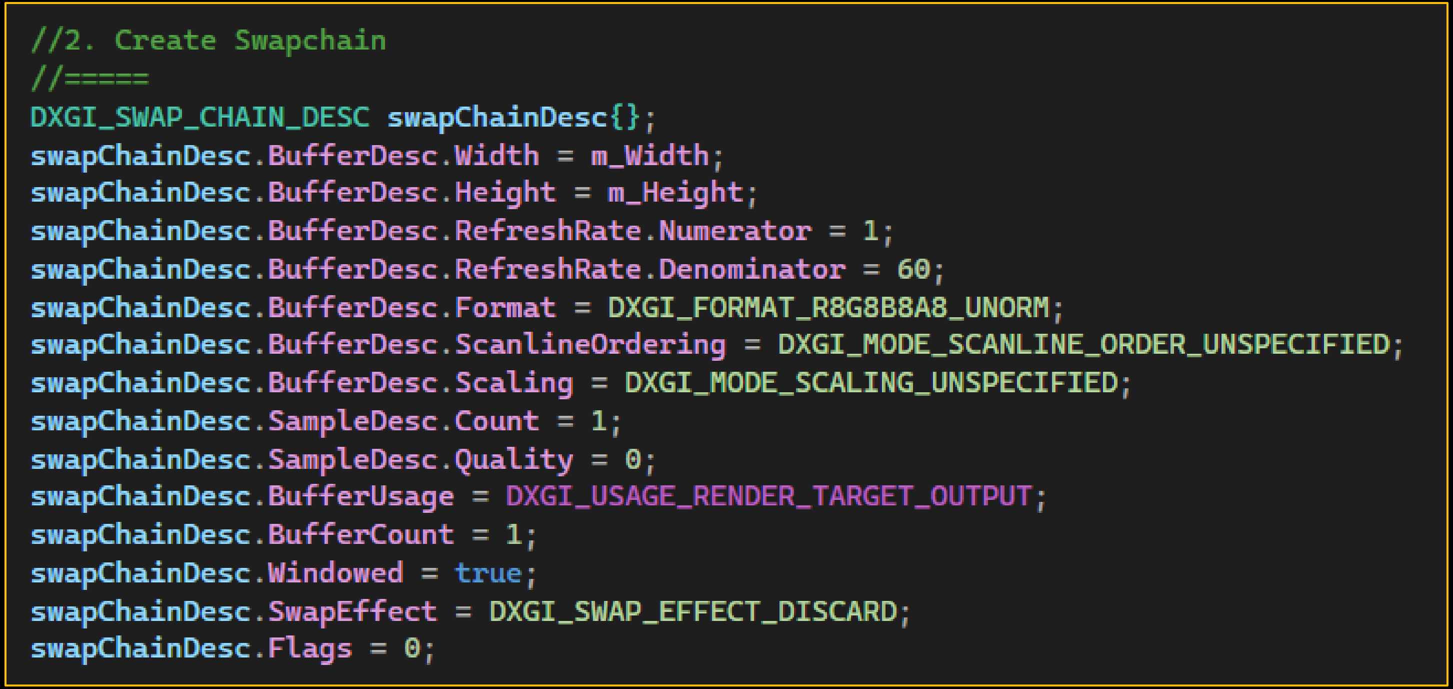
- To setup the swap chain, we need a DGXIFactory1*. We use DXGI because:
  - It gives better performance and saves memory due to the fact it creates the chain according to the GPU hardware. It also has access to some of the device functionality, which again improves performance.



```
//Create DXGI Factory
IDXGIFactory1* pDxgiFactory{};
result = CreateDXGIFactory1(__uuidof(IDXGIFactory1), reinterpret_cast<void**>(&pDxgiFactory));
if (FAILED(result))
    return result;
```

DirectX: Initialization

- If we have the DGXIFactory, we can create the swap chain. In DirectX 11 you let the **device** create all the resources. To determine **what resource** is being created and with what **settings**, you fill in a matching **descriptor**. **Don’t forget to initialize it with { }!**

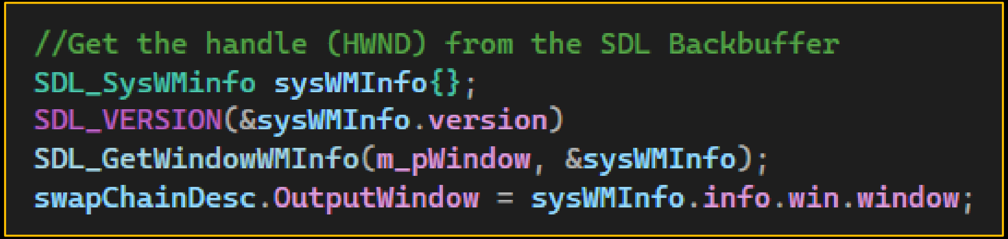


```
//2. Create Swapchain
//=====
DXGI_SWAP_CHAIN_DESC swapChainDesc{};
swapChainDesc.BufferDesc.Width = m_Width;
swapChainDesc.BufferDesc.Height = m_Height;
swapChainDesc.BufferDesc.RefreshRate.Numerator = 1;
swapChainDesc.BufferDesc.RefreshRate.Denominator = 60;
swapChainDesc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
swapChainDesc.BufferDesc.ScanlineOrdering = DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
swapChainDesc.BufferDesc.Scaling = DXGI_MODE_SCALING_UNSPECIFIED;
swapChainDesc.SampleDesc.Count = 1;
swapChainDesc.SampleDesc.Quality = 0;
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount = 1;
swapChainDesc.Windowed = true;
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_DISCARD;
swapChainDesc.Flags = 0;
```

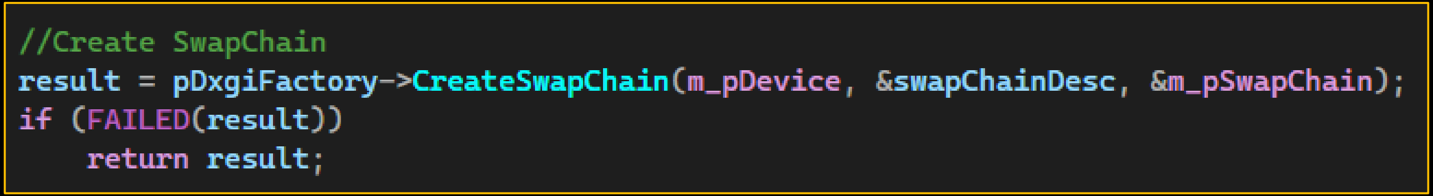
- https://docs.microsoft.com/en-us/windows/win32/api/dxgi/ns-dxgi-dxgi_swap_chain_desc
- https://docs.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-data-conversion

DirectX: Initialization

- When you want to call the function to create the actual swap chain, you'll notice you need to give a handle (HWND) to the window. This handle is from the OS and in our case, is owned by SDL, which controls our window.
- Using the following code, you can get the handle and create the swap chain.



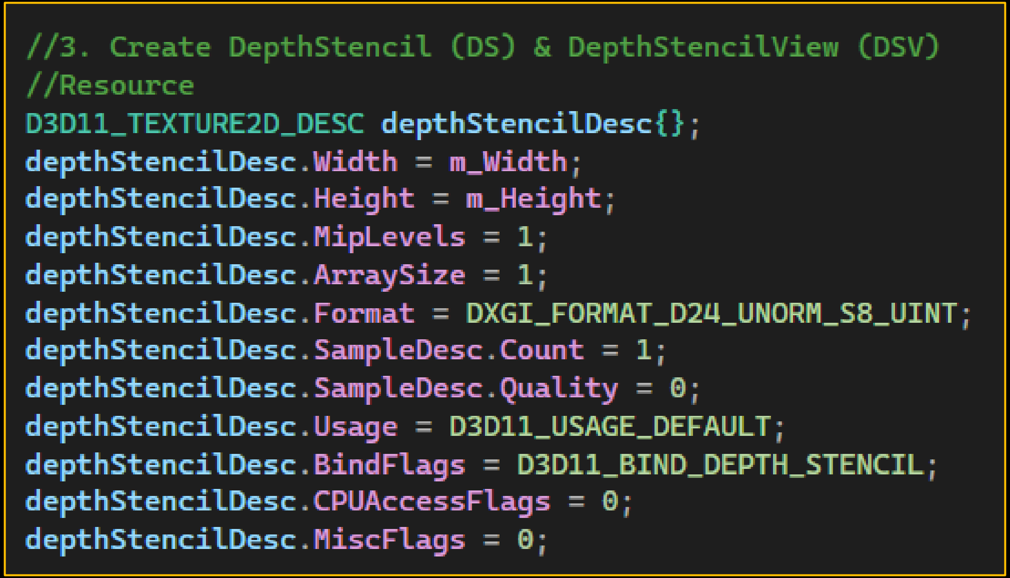
```
//Get the handle (HWND) from the SDL Backbuffer
SDL_SysWMinfo sysWMInfo{};
SDL_VERSION(&sysWMInfo.version)
SDL_GetWindowWMInfo(m_pWindow, &sysWMInfo);
swapChainDesc.OutputWindow = sysWMInfo.info.win.window;
```



```
//Create SwapChain
result = pDxgiFactory->CreateSwapChain(m_pDevice, &swapChainDesc, &m_pSwapChain);
if (FAILED(result))
	return result;
```

DirectX: Initialization

- We also need a Depth Buffer when we want to solve the visibility problem for multiple objects. You again need to create one through the device, using a matching descriptor.



```
//3. Create DepthStencil (DS) & DepthStencilView (DSV)
//Resource
D3D11_TEXTURE2D_DESC depthStencilDesc{};
depthStencilDesc.Width = m_Width;
depthStencilDesc.Height = m_Height;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.ArraySize = 1;
depthStencilDesc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
depthStencilDesc.SampleDesc.Count = 1;
depthStencilDesc.SampleDesc.Quality = 0;
depthStencilDesc.Usage = D3D11_USAGE_DEFAULT;
depthStencilDesc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
depthStencilDesc.CPUAccessFlags = 0;
depthStencilDesc.MiscFlags = 0;
```

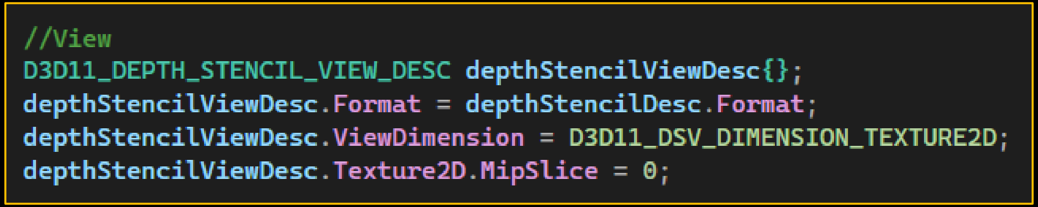
- What is this Stencil Buffer? It’s a buffer used to mask pixels in an image. The mask controls whether a pixel is drawn or not. Thus, using the buffer you enable or disable the drawing to the Render Target on a pixel-by-pixel basis. We will not use it for now though.

DirectX: Initialization

- What is this Render Target you are talking about?
- A render target creates resources for drawing and performs actual drawing operations. In DirectX, a render target consist out of two important parts:
  - Render Target Buffer → ID3D11Resource*
  - Render Target View → ID3D11RenderTargetView*
- DirectX has a certain way of handling resources.
  - ID3D11Resource: the actual data that can be shared by multiple pipeline stages.
  - Resource Views: determines how a resource is used (bound) in the pipeline
    - ID3D11DepthStencilView: access the resource/texture during the depth-stencil testing.
    - ID3D11RenderTargetView: access the resource/texture that is used as a render target.
    - ID3D11ShaderResourceView: access the resource/texture as constant buffer, texture buffer, texture or sampler during rendering.
    - ID3D11UnorderedAccessView: access an unordered resource using a pixel shader or a compute shader (no multisampling though).
  - https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-resources-intro?redirectedfrom=MSDN

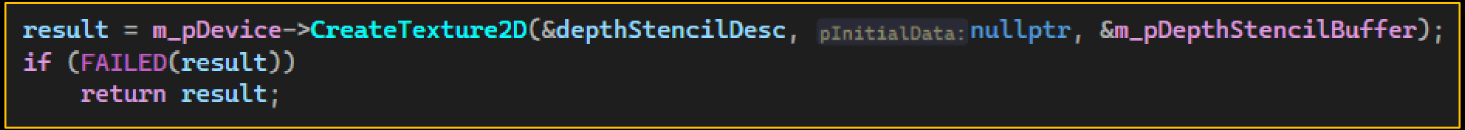
DirectX: Initialization

- The previous descriptor described how to create the resource. Now we also need to describe the resource view for our Depth/Stencil Buffer.

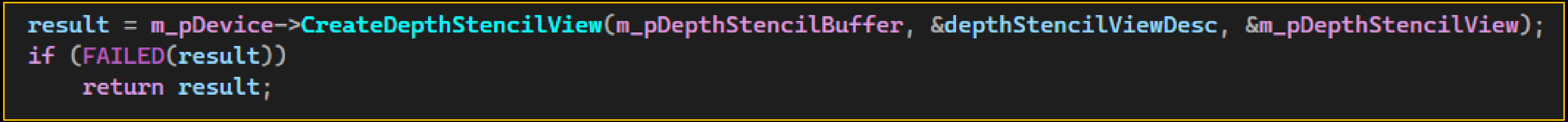


```
//View
D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc{};
depthStencilViewDesc.Format = depthStencilDesc.Format;
depthStencilViewDesc.ViewDimension = D3D11_DSV_DIMENSION_TEXTURE2D;
depthStencilViewDesc.Texture2D.MipSlice = 0;
```

- Now we can create the actual resource and the “matching” resource view.



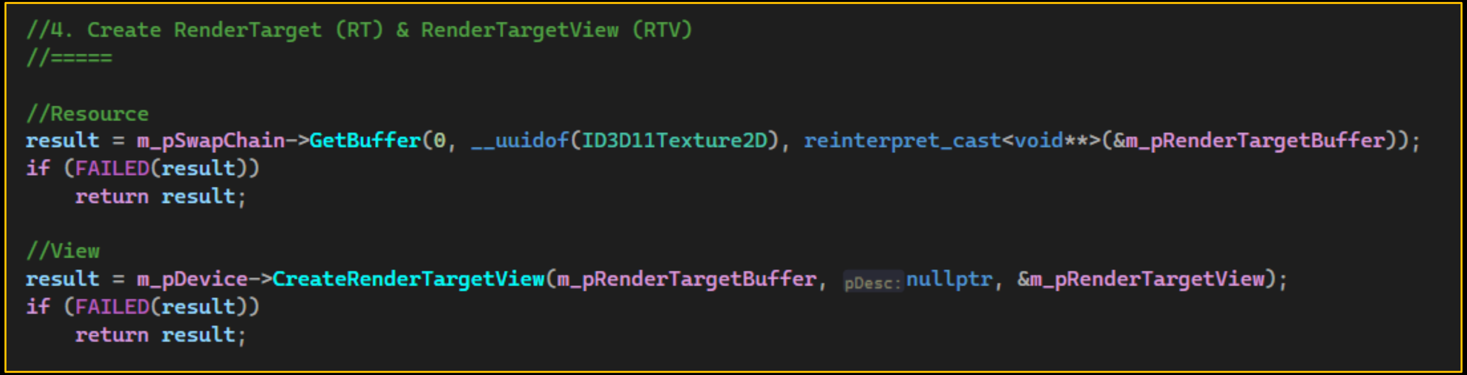
```
result = m_pDevice->CreateTexture2D(&depthStencilDesc, pInitialData:nullptr, &m_pDepthStencilBuffer);
if (FAILED(result))
    return result;
```



```
result = m_pDevice->CreateDepthStencilView(m_pDepthStencilBuffer, &depthStencilViewDesc, &m_pDepthStencilView);
if (FAILED(result))
    return result;
```

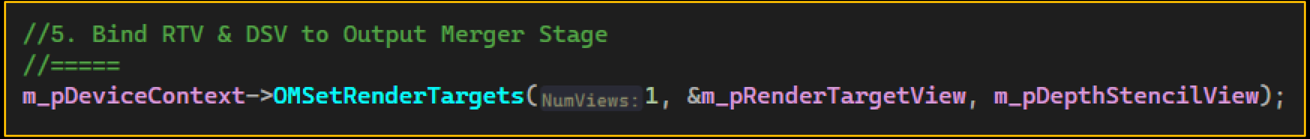
DirectX: Initialization

- Now that we have our depth buffer and back buffer, I want to bind them as the active buffers during rendering.
- As mentioned before, binding happens through resource views. We have one for the depth buffer, but not for the back buffer. We can get the buffer resource from the swap chain using the following code. Once we have the buffer, we can create a resource view for it as well.



```
//4. Create RenderTarget (RT) & RenderTargetView (RTV)
//=====

//Resource
result = m_pSwapChain->GetBuffer(0, __uuidof(ID3D11Texture2D), reinterpret_cast<void**>(&m_pRenderTargetBuffer));
if (FAILED(result))
    return result;

//View
result = m_pDevice->CreateRenderTargetView(m_pRenderTargetBuffer, pDesc: nullptr, &m_pRenderTargetView);
if (FAILED(result))
    return result;
```

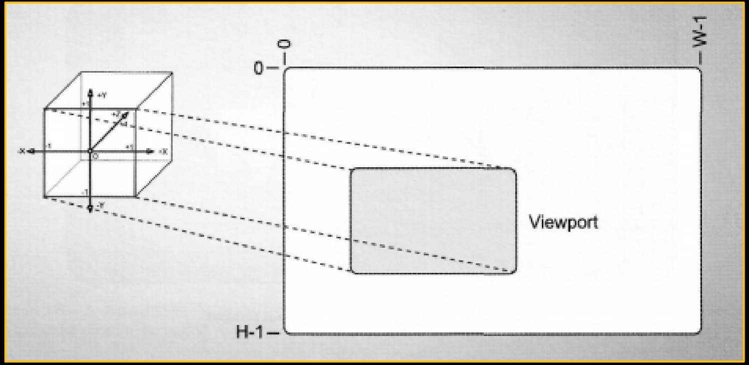
- Using the two views, bind them as the active buffers during the Output Merger Stage.

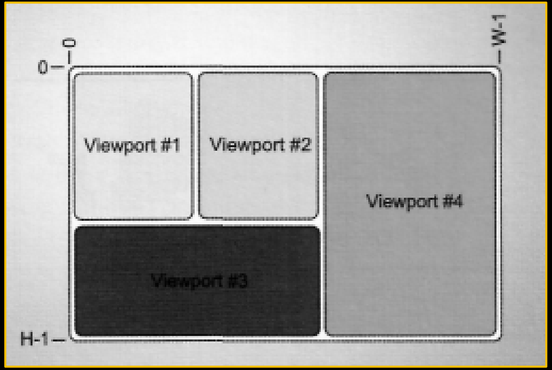


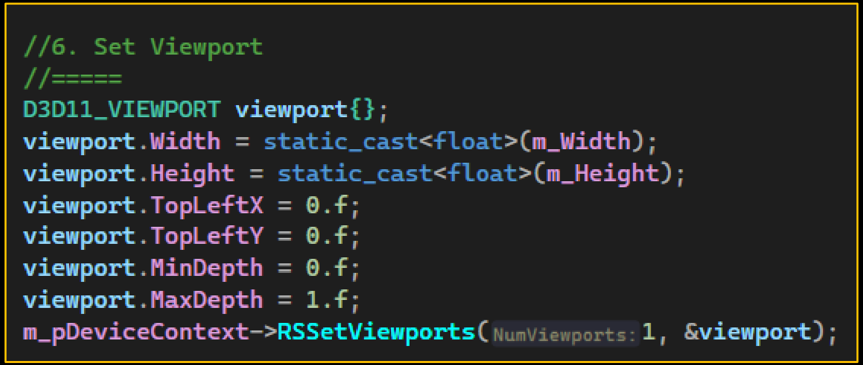
```
//5. Bind RTV & DSV to Output Merger Stage
//=====
m_pDeviceContext->OMSetRenderTargets(NumViews: 1, &m_pRenderTargetView, m_pDepthStencilView);
```

DirectX: Initialization

- Finally, there is still one more thing we need to set for DirectX, the viewport.
- The viewport defines where the content of the back buffer will be rendered on the screen. DirectX will use this viewport to transform the NDC to the correct screen space coordinates.



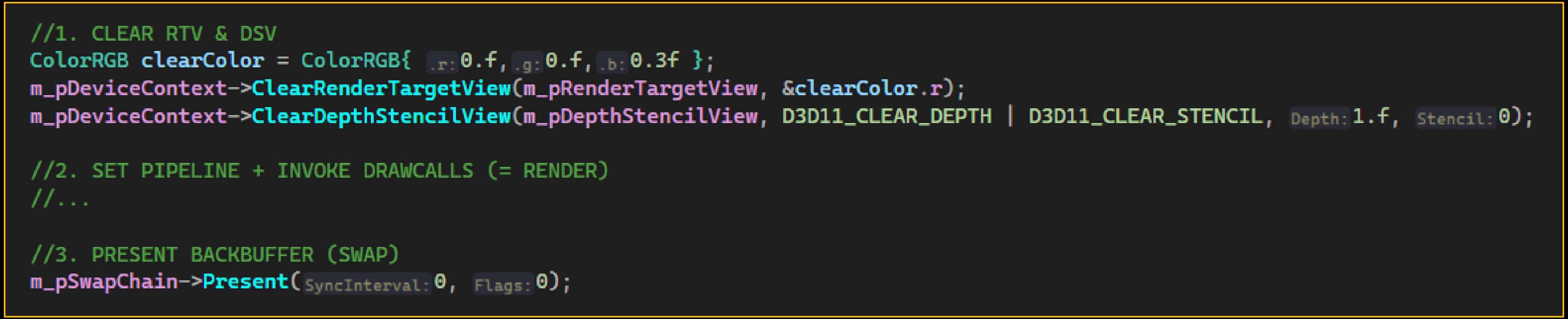




```
//6. Set Viewport
//=====
D3D11_VIEWPORT viewport{};
viewport.Width = static_cast<float>(m_Width);
viewport.Height = static_cast<float>(m_Height);
viewport.TopLeftX = 0.f;
viewport.TopLeftY = 0.f;
viewport.MinDepth = 0.f;
viewport.MaxDepth = 1.f;
m_pDeviceContext->RSSetViewports(NumViewports: 1, &viewport);
```

DirectX: Initialization

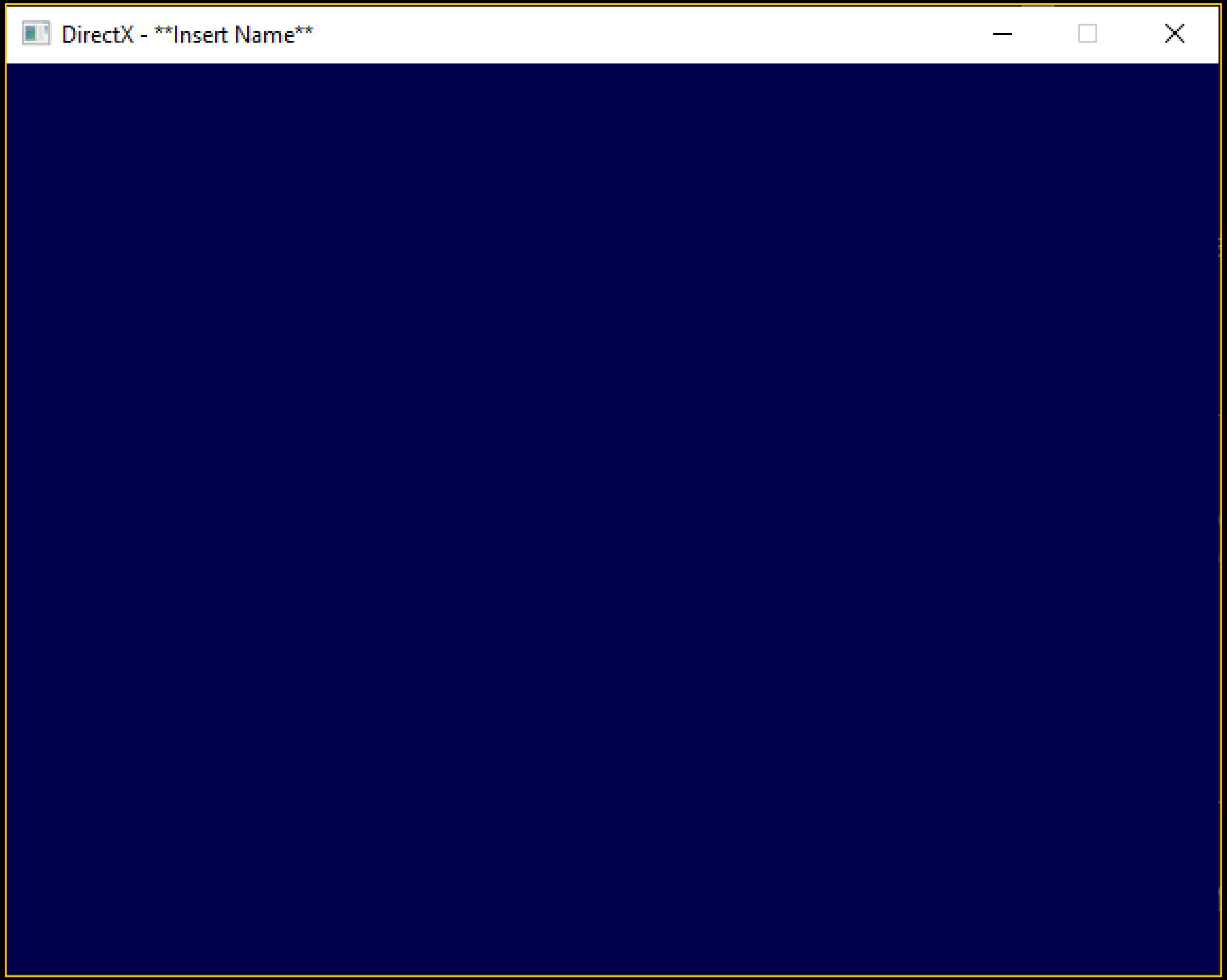
- We are done with setting up the most basic DirectX pipeline. Only thing we now must do is:
  - Clear our buffers every frame → Depending on the buffer the value will be different!
  - Render ☺
  - Present the contents of the backbuffer to the screen. (Swapping)



```
//1. CLEAR RTV & DSV
ColorRGB clearColor = ColorRGB{ .r:0.f, .g:0.f, .b:0.3f };
m_pDeviceContext->ClearRenderTargetView(m_pRenderTargetView, &clearColor.r);
m_pDeviceContext->ClearDepthStencilView(m_pDepthStencilView, D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, Depth:1.f, Stencil:0);

//2. SET PIPELINE + INVOKE DRAWCALLS (= RENDER)
//...

//3. PRESENT BACKBUFFER (SWAP)
m_pSwapChain->Present(SyncInterval:0, Flags:0);
```

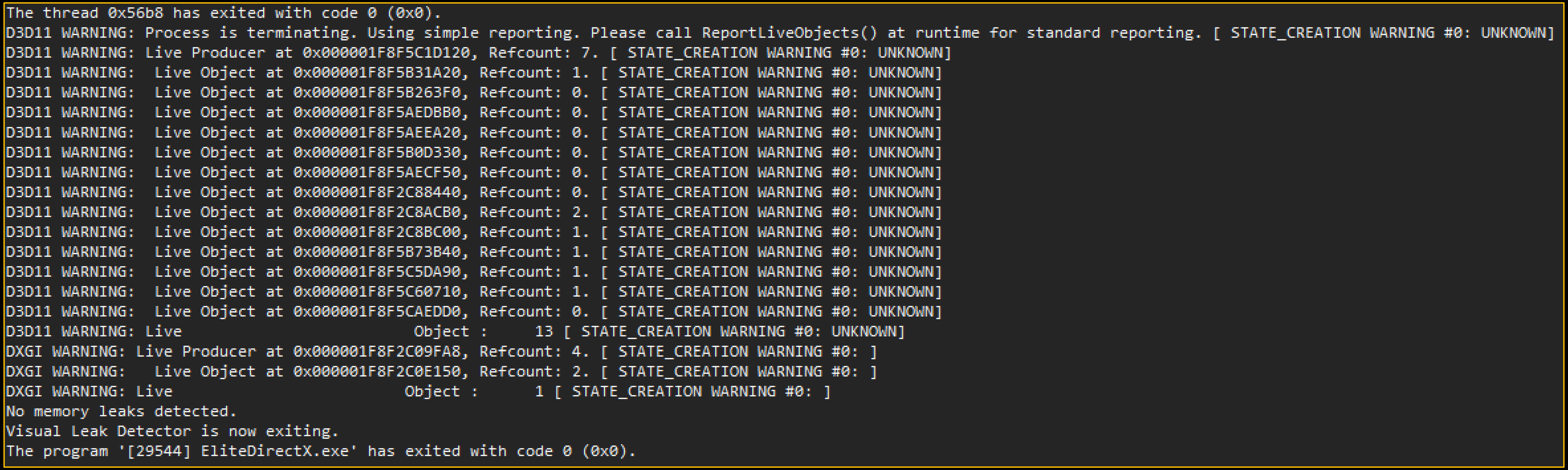
- This obviously happens in the **Render** function of our renderer.

DirectX: Initialization



DirectX - **Insert Name**

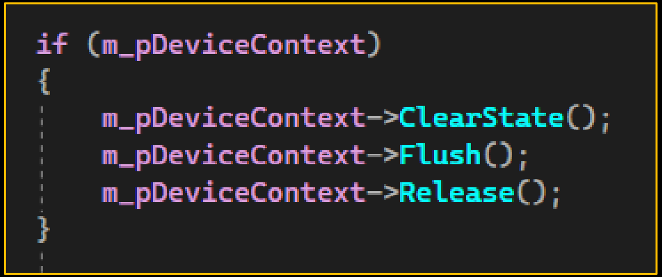
DirectX: Initialization



```
The thread 0x56b8 has exited with code 0 (0x0).
D3D11 WARNING: Process is terminating. Using simple reporting. Please call ReportLiveObjects() at runtime for standard reporting. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Producer at 0x000001F8F5C1D120, Refcount: 7. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5B31A20, Refcount: 1. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5B263F0, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5AEDBB0, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5AEEA20, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5B0D330, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5AECF50, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F2C88440, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F2C8ACB0, Refcount: 2. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F2C8BC00, Refcount: 1. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5B73B40, Refcount: 1. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5C5DA90, Refcount: 1. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5C60710, Refcount: 1. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING:  Live Object at 0x000001F8F5CAEDD0, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live                         Object :     13 [ STATE_CREATION WARNING #0: UNKNOWN]
DXGI WARNING: Live Producer at 0x000001F8F2C09FA8, Refcount: 4. [ STATE_CREATION WARNING #0: ]
DXGI WARNING:   Live Object at 0x000001F8F2C0E150, Refcount: 2. [ STATE_CREATION WARNING #0: ]
DXGI WARNING: Live                         Object :      1 [ STATE_CREATION WARNING #0: ]
No memory leaks detected.
Visual Leak Detector is now exiting.
The program '[29544] EliteDirectX.exe' has exited with code 0 (0x0).
```

DirectX: Initialization

- We have **resource leaks** ☹. This is as bad as memory leaks! Whenever you create a resource **through the device**, you must **release** the resource when you are done with it!
- We release the resources in **reversed order**!
- Some resources, like the device context, might require some extra work.
- We get 7 resource leaks fix them! There is a **hidden leak** of the DXGI factory, so fix that as well!
    - Render Target View
    - Render Target Buffer
    - Depth Stencil View
    - Depth Stencil Buffer
    - Swap Chain
    - Device Context
    - Device
    - DXGIFactory



```
if (m_pDeviceContext)
{
    m_pDeviceContext->ClearState();
    m_pDeviceContext->Flush();
    m_pDeviceContext->Release();
}
```

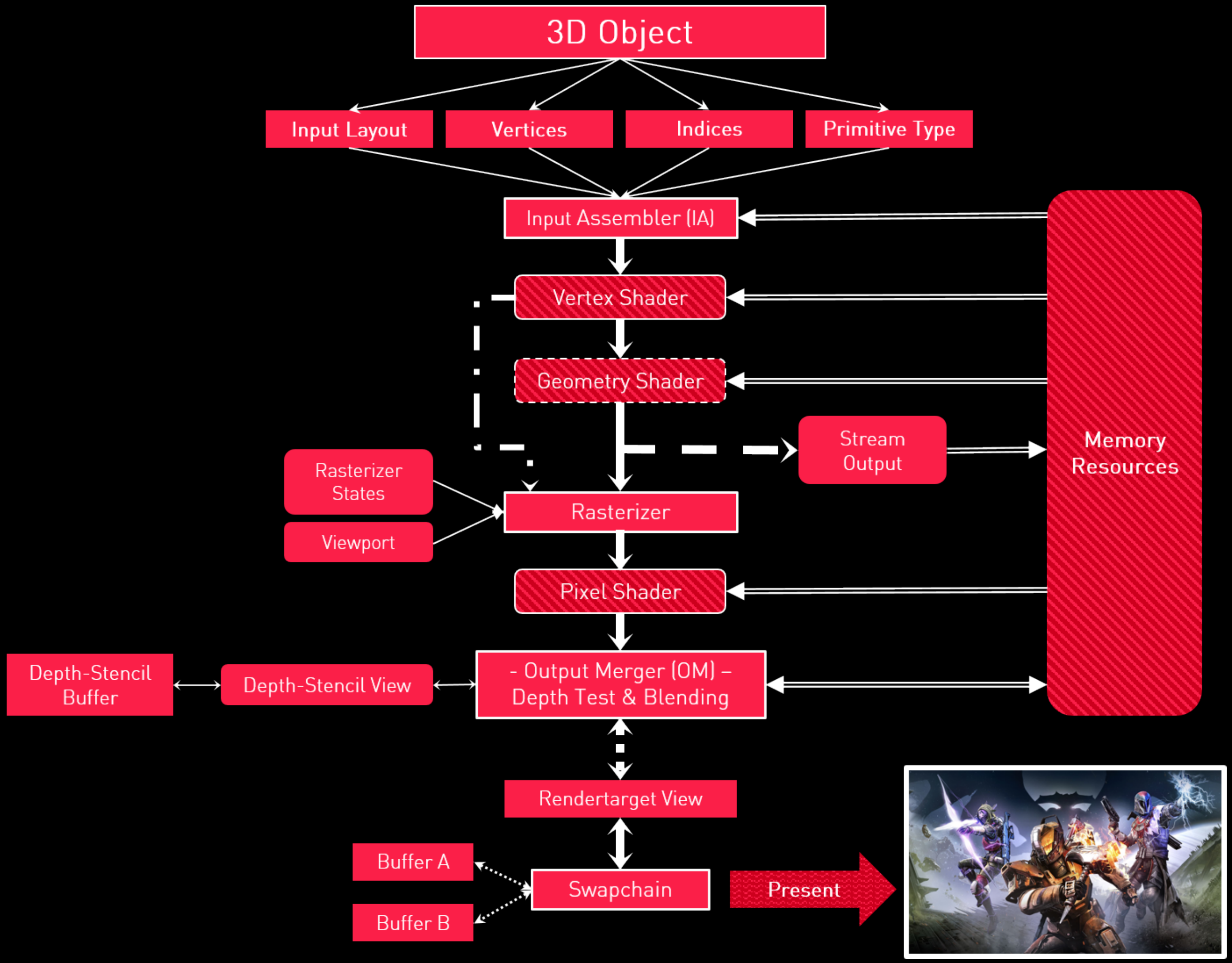
DirectX: Graphics Pipeline

- Now that we have our DirectX 11 up and running, let’s have a look at what we call the Programmable Render Pipeline.
- You need to know this by heart!! ☺

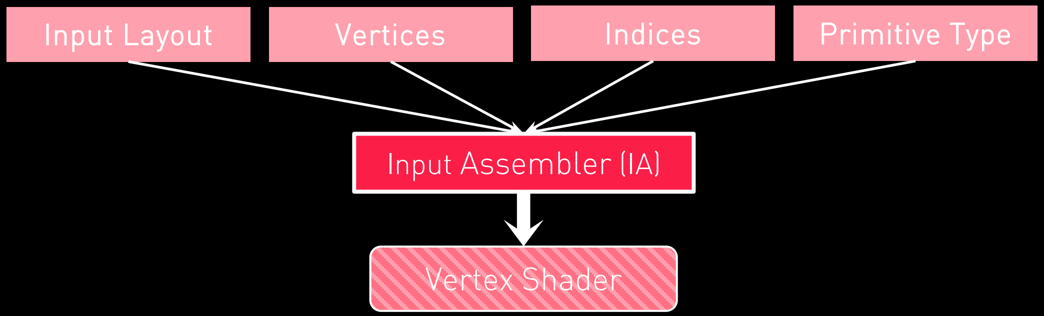
- Are you ready…?

- Sure…?

DirectX: Graphics Pipeline

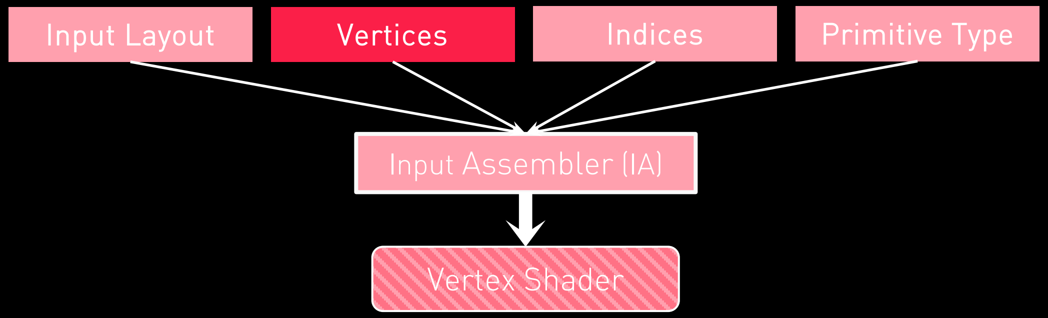


DirectX: Graphics Pipeline



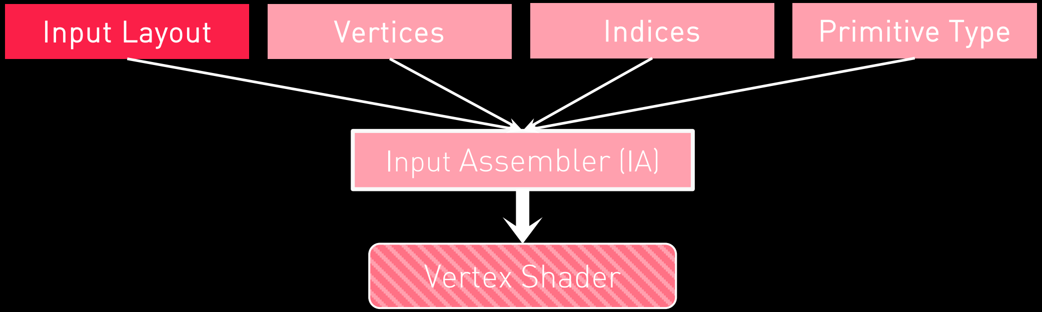
- Input Assembler Stage
  - It reads the geometric data (vertices and indices) from memory.
  - There are different primitive types, and based on the setting it will assemble all the data into the correct type.
  - Let's go over all four parts that define the assembly.

DirectX: Graphics Pipeline

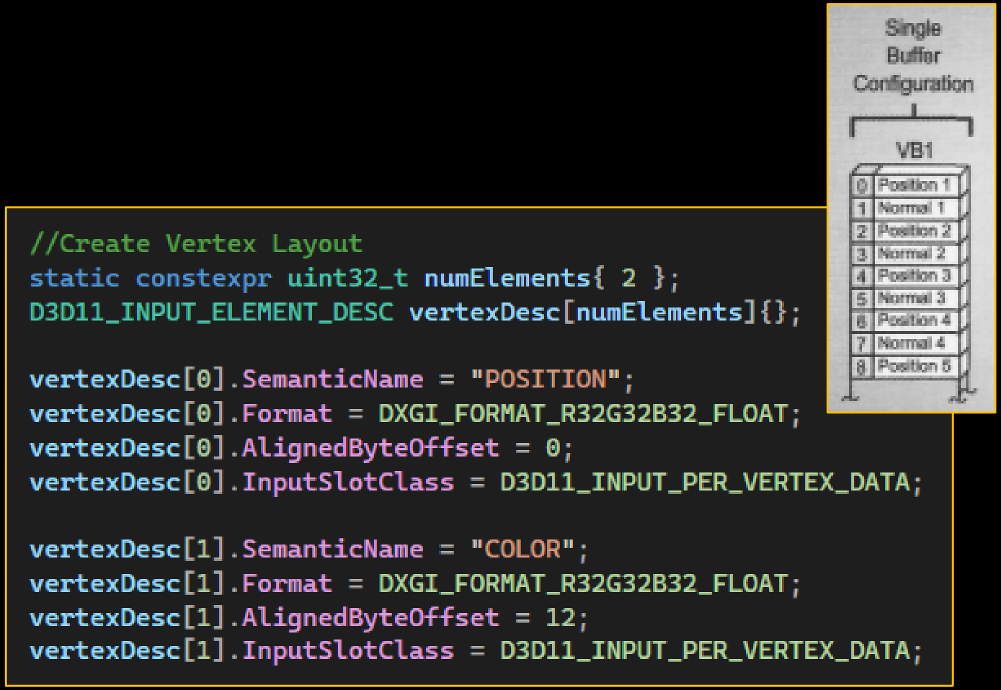


- Vertices:
  - This is basically our vertex buffer. In DirectX you can create different types of buffers:
    - Immutable Buffer: fixed data, thus filled once. GPU can only read it and CPU has no access at all.
    - Dynamic Buffer: can be modified at runtime. Is accessible by both the GPU and CPU.
    - Staging Buffer: resource that supports data transfer from the GPU to the CPU.
  - What a vertex holds is up to you. Just like with the software rasterizer, only a position is mandatory!

DirectX: Graphics Pipeline

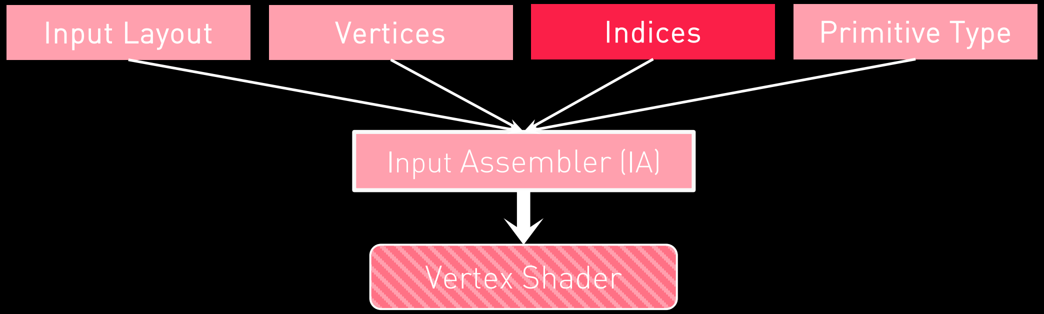


- Input Layout:
  - This describes the layout of the vertex you define. This is necessary so the GPU knows what data is available and how it should traverse the data buffer!
  - When defining the layout pay extra attention to:
    - Format
    - ByteOffset

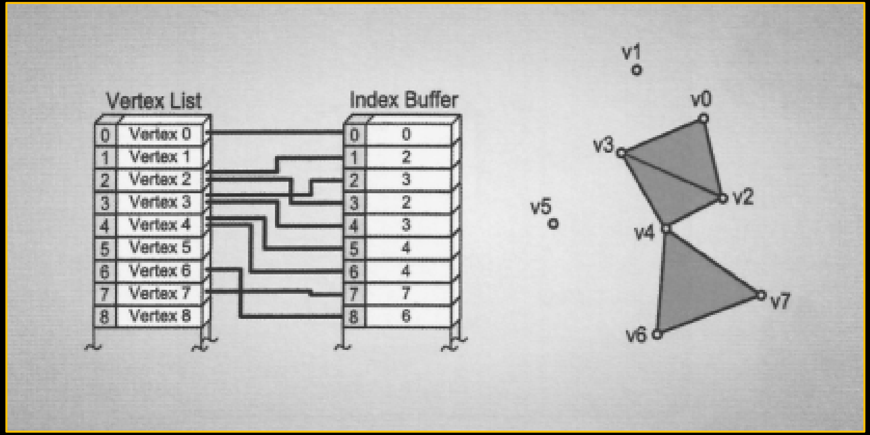


```
//Create Vertex Layout
static constexpr uint32_t numElements{ 2 };
D3D11_INPUT_ELEMENT_DESC vertexDesc[numElements]{};

vertexDesc[0].SemanticName = "POSITION";
vertexDesc[0].Format = DXGI_FORMAT_R32G32B32_FLOAT;
vertexDesc[0].AlignedByteOffset = 0;
vertexDesc[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;

vertexDesc[1].SemanticName = "COLOR";
vertexDesc[1].Format = DXGI_FORMAT_R32G32B32_FLOAT;
vertexDesc[1].AlignedByteOffset = 12;
vertexDesc[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
```

DirectX: Graphics Pipeline



- Indices:
  - Nothing changes compared to the software rasterizer. It defines the sequence of the vertices that form the 3D model. ☺



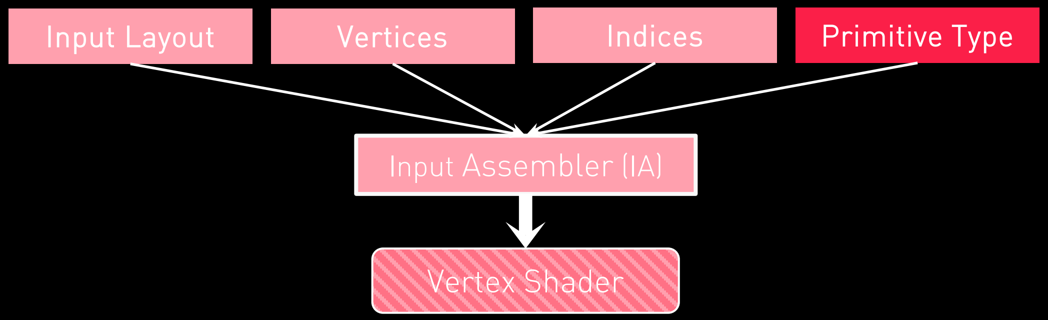
Vertex List

0	Vertex 0
1	Vertex 1
2	Vertex 2
3	Vertex 3
4	Vertex 4
5	Vertex 5
6	Vertex 6
7	Vertex 7
8	Vertex 8

Index Buffer

0	0
1	2
2	3
3	2
4	3
5	4
6	4
7	7
8	6

DirectX: Graphics Pipeline



Input Layout

Vertices

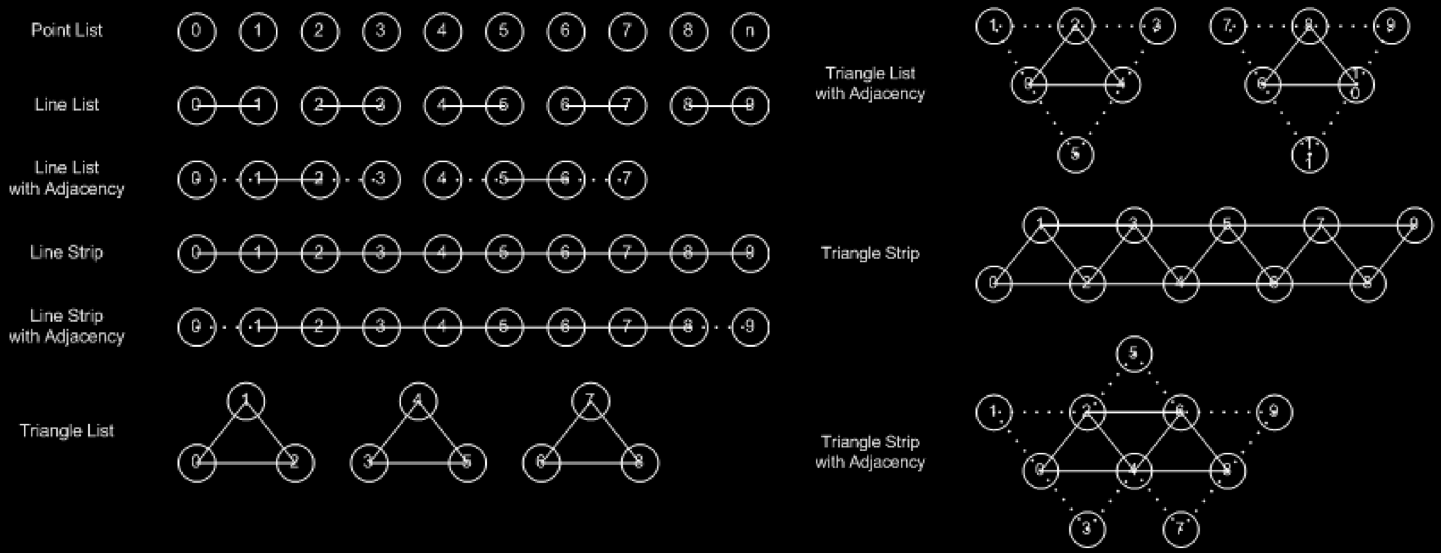
Indices

Primitive Type

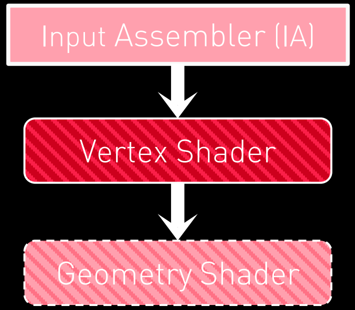
Input Assembler (IA)

Vertex Shader

- **Primitive Type:**

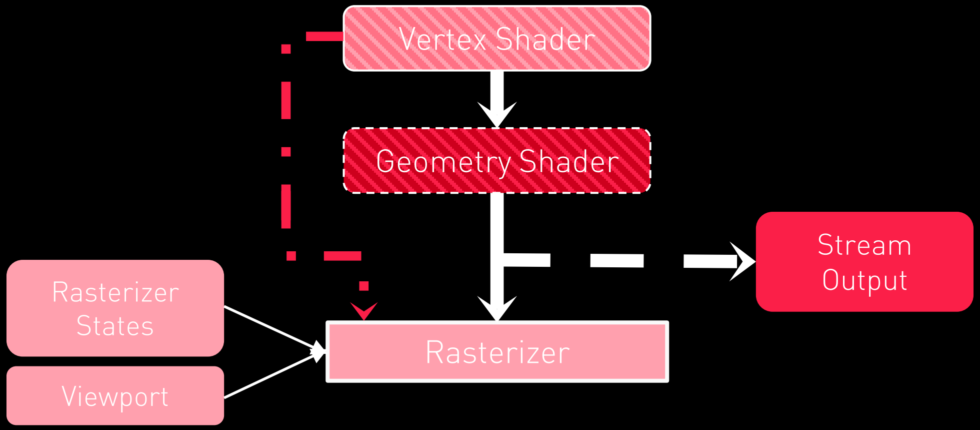

Point List
Line List
Line List with Adjacency
Line Strip
Line Strip with Adjacency
Triangle List
Triangle List with Adjacency
Triangle Strip
Triangle Strip with Adjacency

DirectX: Graphics Pipeline



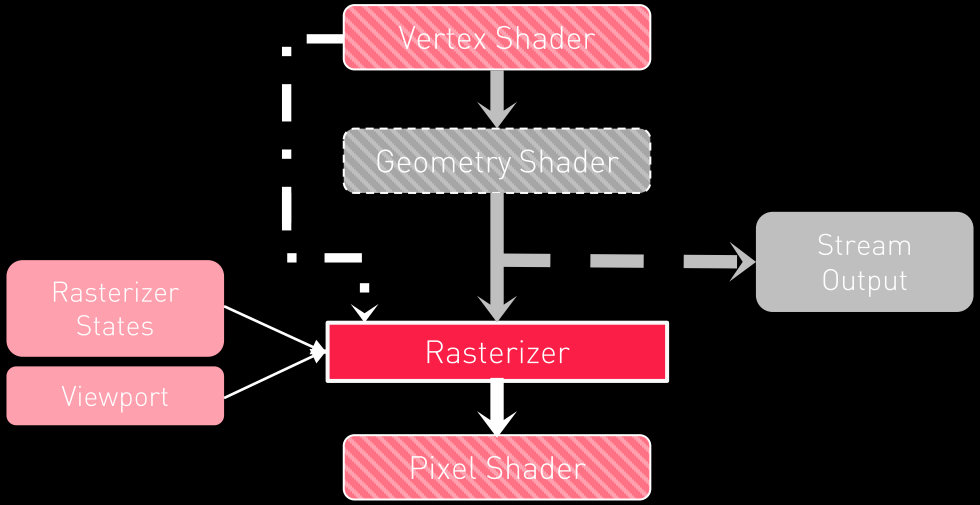
- Vertex Shader:
  - This is equal to your VertexTransformation function from the software rasterizer. Only difference, it doesn't do the perspective divide. This happens in the Rasterizer Stage.
  - So, the main purpose is to transform a vertex from object space to projection space (before the divide).
  - A shader is a function that is getting executed on the GPU in parallel! In DirectX we define a shader in an .hlsl (High Level Shader Language) file. We'll have a look in a minute… ☺
  - Vertex Shaders are also used for:
    - Per Vertex Lighting
    - Displacement Mapping → Snow Deformation
    - Skinning (Animation)
    - …

DirectX: Graphics Pipeline

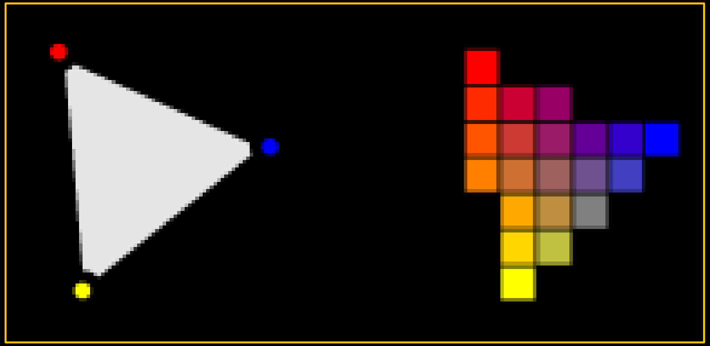


- Geometry Shader:
  - This is an optional shader stage. We won't use it this semester. You can use it for adding or removing geometry on the GPU.
  - There are other optional stages like this, for example, tessellation stage. But we won't pay attention to them for now as well.

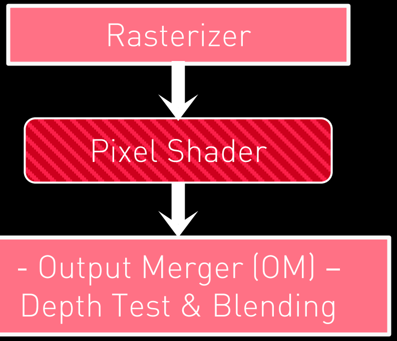
DirectX: Graphics Pipeline



- Rasterizer:
  - This does what you've been doing the last couple of weeks. ☺ So, it does:
    - Culling
    - Clipping
    - Homogenous/Perspective Divide
    - Viewport Transformation
    - Fragment Generation
    - Attribute Interpolation

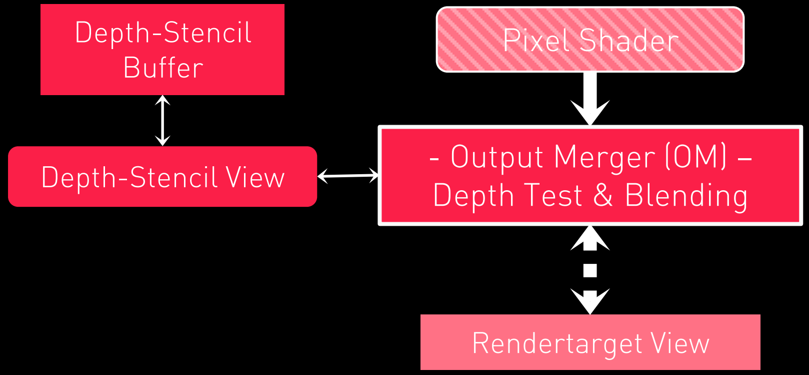


DirectX: Graphics Pipeline



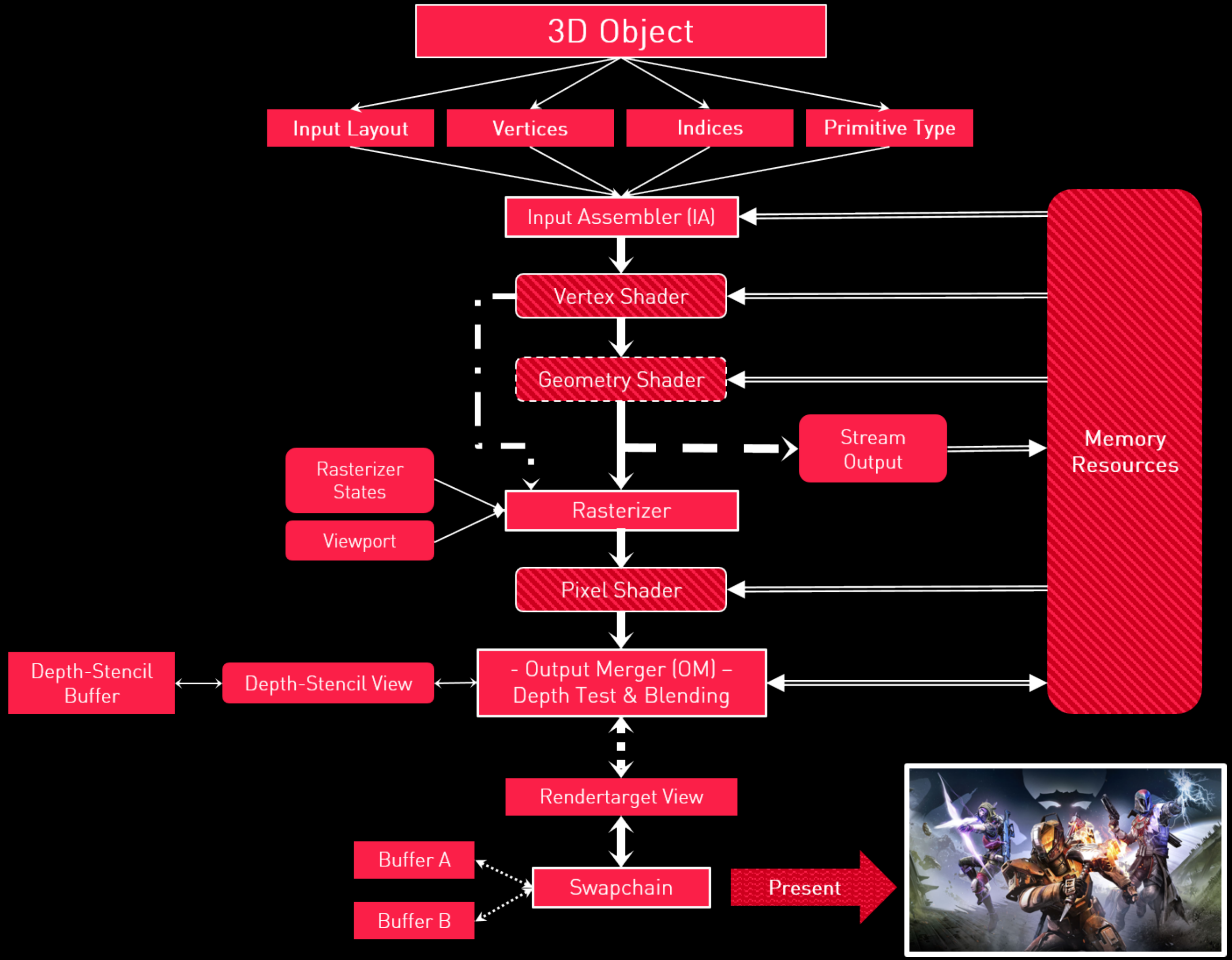
- Pixel Shader:
  - This is executed after the rasterizer and does what your PixelShading function does.
  - Just as the vertex shader, it's a function that gets executed on the GPU and it is written in .hlsl.
  - For your information, internally the pixel shader gets executed on 2x2 pixel tiles. This means these 4 pixels gets calculated in parallel. This gives us some extra information (derivative quantities) for other techniques. Don't worry about this though.
  - https://docs.microsoft.com/en-us/windows/win32/direct3d11/pixel-shader-stage
  - Pixel can be clipped (discarded) by the HLSL clip function. Can help performance in some cases.
  - Pixel can be occluded by another pixel fragment by the Depth Test in the Output Merger Stage.

DirectX: Graphics Pipeline



- Output Merger:
  - Performs the Depth and Stencil Test after the pixel shader.
  - Does this mean we do unnecessary calculations in the pixel shader? No, there are hardware optimizations that perform early depth testing, etc. Don’t worry! ☺
  - It also does blending in case we use transparency. Later more!
  - In the end it writes the result to the back buffer.

DirectX: Graphics Pipeline

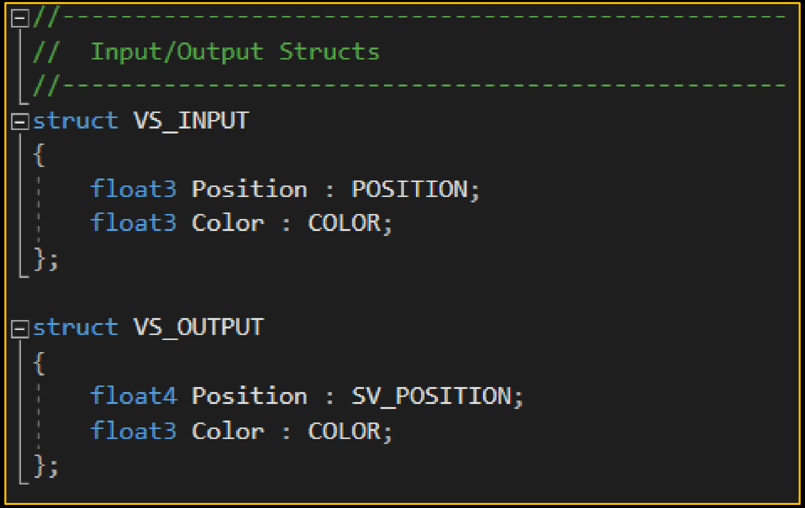


DirectX: Rendering

- Let’s start rendering!
- We need three things:
  - Shader performing our Vertex Transformation and Pixel Shading.
  - Create an effect class that represents our shader. We need this to create our input layout.
  - Mesh representation, in other words, our data buffers.
- Let’s start by creating our shader…
- We use the Effect Framework, which means we are going to create .fx files instead of .hlsl. They are still written in HLSL, but they have some extra features which will proof useful in the future.
- Create an .fx file called PosCol3D.fx, put it in a resources folder (just like your meshes and textures) and definitely DON’T add it to Visual Studio!
  Why? You don’t want to compile this file using the VS compiler but use the DirectX Shader Compiler (fxc.exe) instead!
- Open the file in your preferred text editor ☺

DirectX: Rendering

- We start by defining how a vertex looks like. Just like in C++, create a **struct** that **matches** the layout. It's important that the vertex struct in C++ matches with the one in HLSL!

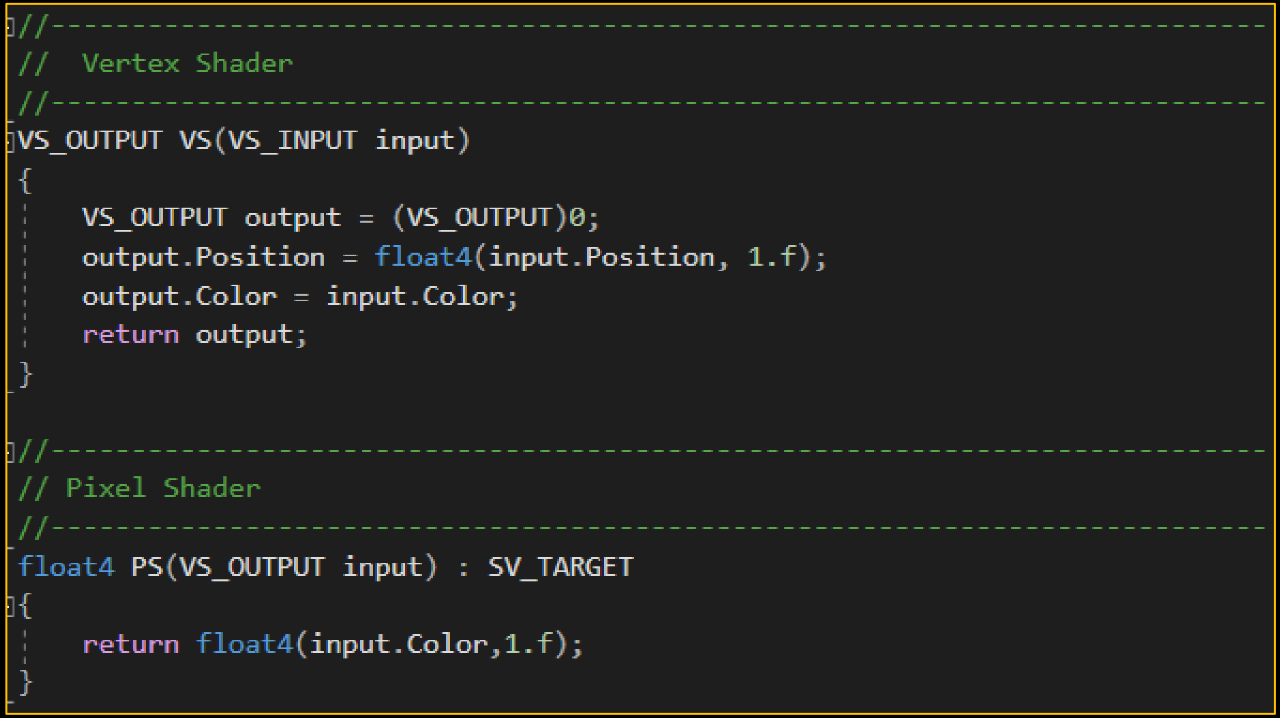


```
//---------------------------------------------------
//  Input/Output Structs
//---------------------------------------------------
struct VS_INPUT
{
    float3 Position : POSITION;
    float3 Color : COLOR;
};

struct VS_OUTPUT
{
    float4 Position : SV_POSITION;
    float3 Color : COLOR;
};
```

- **Semantics** can be used to reference GPU variables. Some are required though:
    - SV_POSITION: contains the position of the vertex after the transformation and is required in the rasterizer.
    - SV_TARGET: used to reference to which render target you want to render. Yes, you can bind multiple!
- https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics

DirectX: Rendering

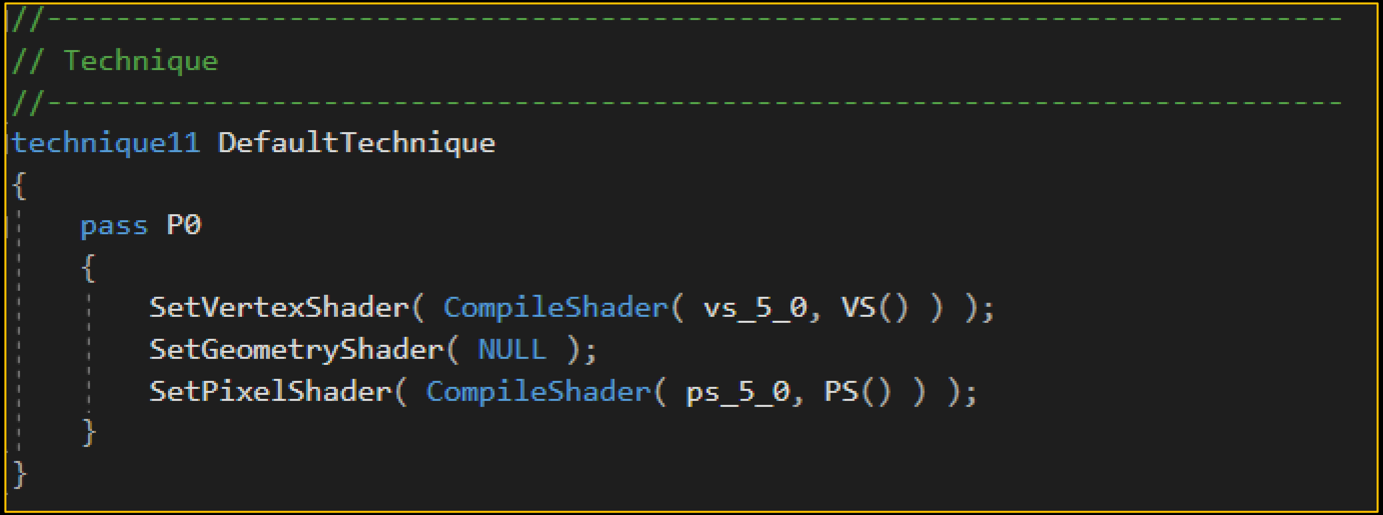
- Once we have the structs defined, create our shader functions.
- Don’t forget to **forward data from one stage to another**. If you don’t you **lose** the information!



```
//--------------------------------------------------------------------------
//  Vertex Shader
//--------------------------------------------------------------------------
VS_OUTPUT VS(VS_INPUT input)
{
    VS_OUTPUT output = (VS_OUTPUT)0;
    output.Position = float4(input.Position, 1.f);
    output.Color = input.Color;
    return output;
}

//--------------------------------------------------------------------------
// Pixel Shader
//--------------------------------------------------------------------------
float4 PS(VS_OUTPUT input) : SV_TARGET
{
    return float4(input.Color,1.f);
}
```

DirectX: Rendering

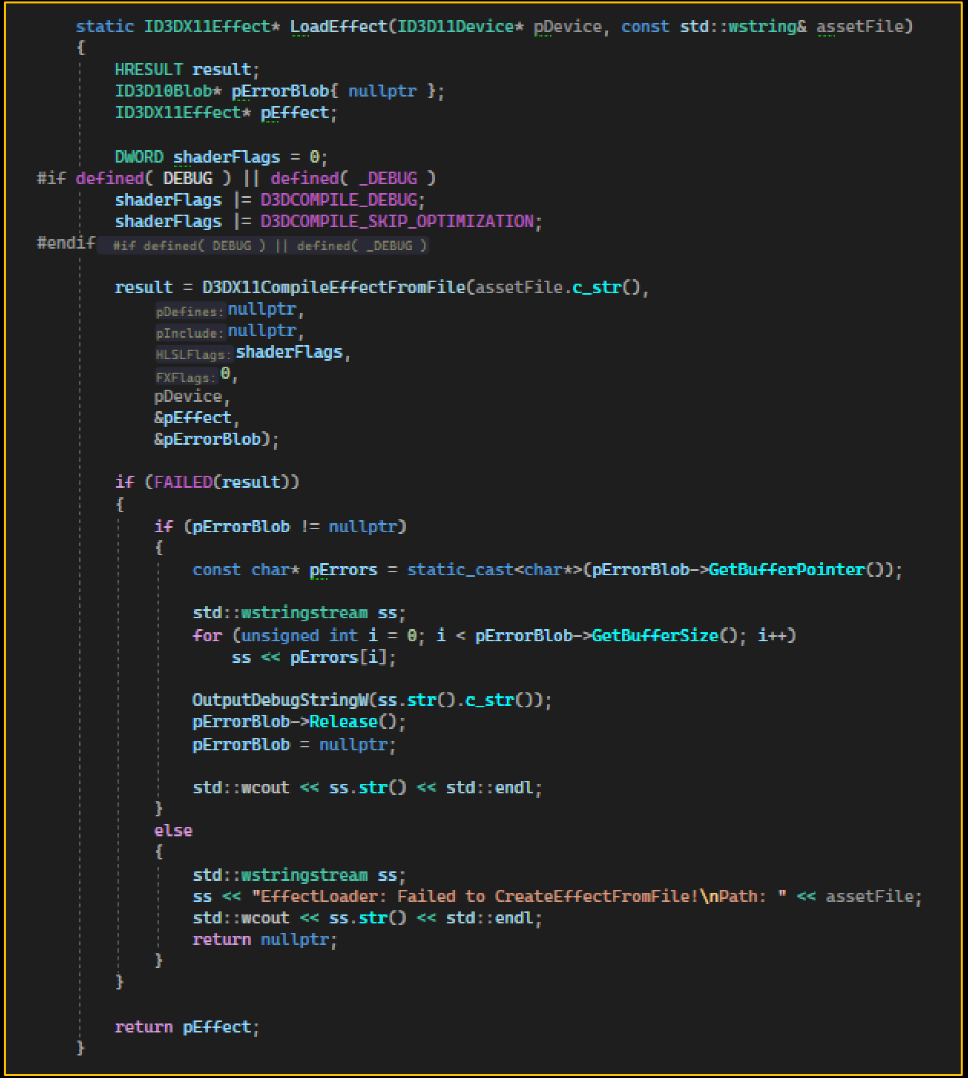
- Finally, because we are using the Effect Framework, we must define a **technique**.
- The technique is the **“actual shader”** because it defines which functions to use for which stage. A technique can have multiple **passes** that run in a sequential order.



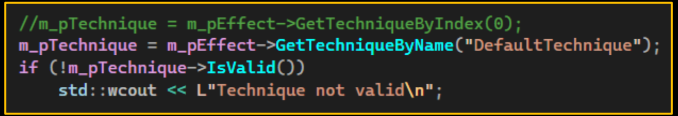
```
//----------------------------------------------------------------------------
// Technique
//----------------------------------------------------------------------------
technique11 DefaultTechnique
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_5_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_5_0, PS() ) );
    }
}
```

DirectX: Rendering

- Now we are ready to compile and load this effect file into memory!
- Create an effect class and add the following function that will compile and load your effect.
- In the constructor of the effect class, accept the pointer to the ID3D11Device and a path that determines which effect to load.
- In the constructor, load the effect using the function defined to the right, store the resulting pointer in a member of type ID3DX11Effect.
- Also get the technique and store it in a data member. We'll need this later for the Input Layout.
- Write two getter functions that gives you access to the two data members.
- Don't forget to release your resources in the destructor!



```
    static ID3DX11Effect* LoadEffect(ID3D11Device* pDevice, const std::wstring& assetFile)
    {
        HRESULT result;
        ID3D10Blob* pErrorBlob{ nullptr };
        ID3DX11Effect* pEffect;

        DWORD shaderFlags = 0;
#if defined( DEBUG ) || defined( _DEBUG )
        shaderFlags |= D3DCOMPILE_DEBUG;
        shaderFlags |= D3DCOMPILE_SKIP_OPTIMIZATION;
#endif  #if defined( DEBUG ) || defined( _DEBUG )

        result = D3DX11CompileEffectFromFile(assetFile.c_str(),
            pDefines: nullptr,
            pInclude: nullptr,
            HLSLFlags: shaderFlags,
            FXFlags: 0,
            pDevice,
            &pEffect,
            &pErrorBlob);

        if (FAILED(result))
        {
            if (pErrorBlob != nullptr)
            {
                const char* pErrors = static_cast<char*>(pErrorBlob->GetBufferPointer());

                std::wstringstream ss;
                for (unsigned int i = 0; i < pErrorBlob->GetBufferSize(); i++)
                    ss << pErrors[i];

                OutputDebugStringW(ss.str().c_str());
                pErrorBlob->Release();
                pErrorBlob = nullptr;

                std::wcout << ss.str() << std::endl;
            }
            else
            {
                std::wstringstream ss;
                ss << "EffectLoader: Failed to CreateEffectFromFile!\nPath: " << assetFile;
                std::wcout << ss.str() << std::endl;
                return nullptr;
            }
        }

        return pEffect;
    }
```

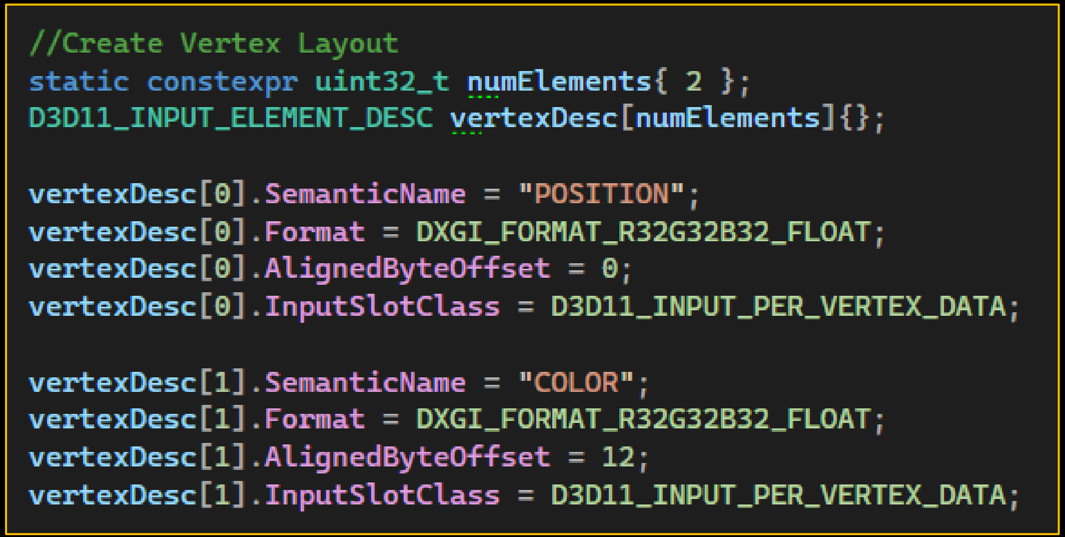


```
//m_pTechnique = m_pEffect->GetTechniqueByIndex(0);
m_pTechnique = m_pEffect->GetTechniqueByName("DefaultTechnique");
if (!m_pTechnique->IsValid())
    std::wcout << L"Technique not valid\n";
```

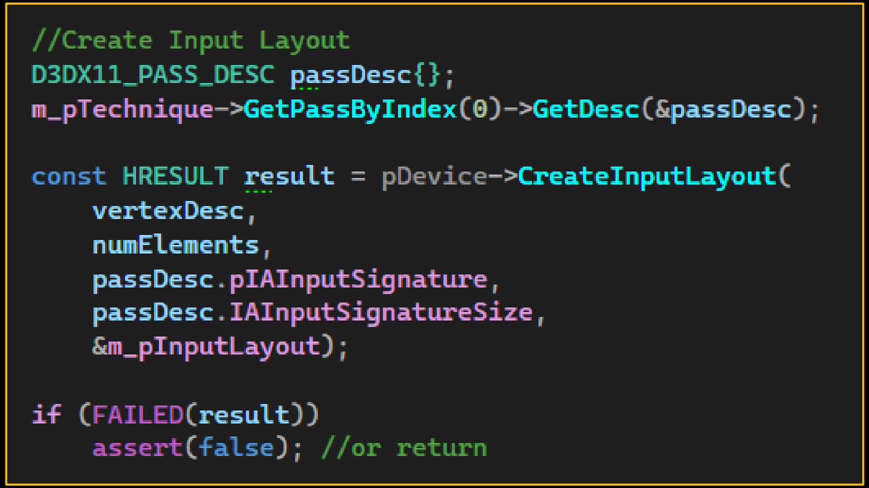
DirectX: Rendering

- With this up and running, we can now create an 3D mesh representation.
- Create a class for the mesh representation. Also define a Vertex struct that has the same layout as the one defined in the shader.
- Make sure the constructor accepts the ID3D11Device*, a container that holds the raw vertex data and one that hold the raw index data, like your software rasterizer.
- In the constructor:
  - Create an instance of the effect class you just created. This could be optimized, but that’s engine design. Don’t worry too much for now!
  - Create the vertex layout using, again, a matching descriptor.
  - Through the technique of the effect, create the input layout, using the vertex layout descriptor.
  - Create the DirectX Vertex Buffer and Index Buffer, using the device and a descriptor ☺
- Don’t forget to release the resources you’ve created through the device in the destructor.

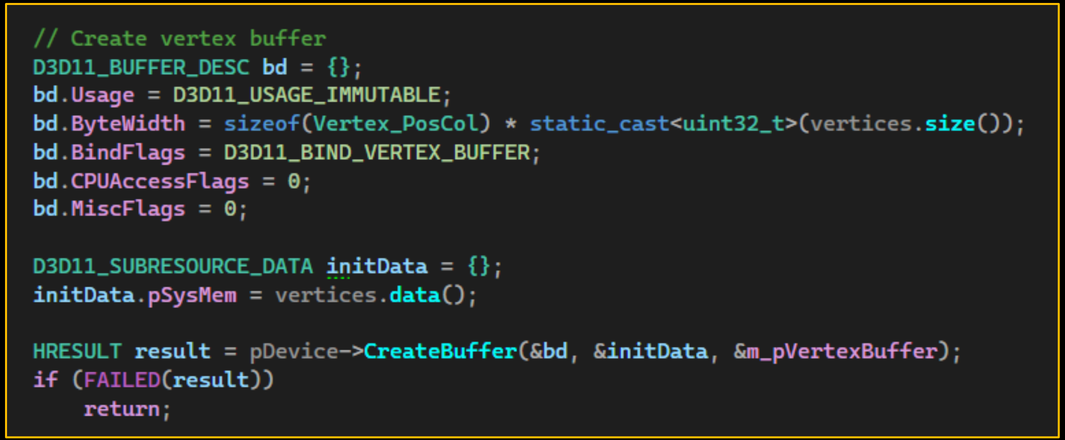
DirectX: Rendering



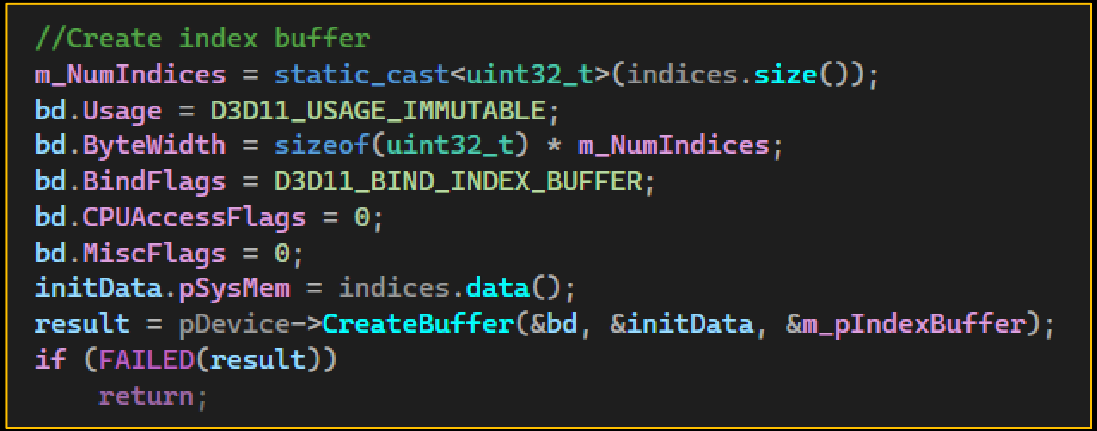
```
//Create Vertex Layout
static constexpr uint32_t numElements{ 2 };
D3D11_INPUT_ELEMENT_DESC vertexDesc[numElements]{};

vertexDesc[0].SemanticName = "POSITION";
vertexDesc[0].Format = DXGI_FORMAT_R32G32B32_FLOAT;
vertexDesc[0].AlignedByteOffset = 0;
vertexDesc[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;

vertexDesc[1].SemanticName = "COLOR";
vertexDesc[1].Format = DXGI_FORMAT_R32G32B32_FLOAT;
vertexDesc[1].AlignedByteOffset = 12;
vertexDesc[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
```



```
//Create Input Layout
D3DX11_PASS_DESC passDesc{};
m_pTechnique->GetPassByIndex(0)->GetDesc(&passDesc);

const HRESULT result = pDevice->CreateInputLayout(
    vertexDesc,
    numElements,
    passDesc.pIAInputSignature,
    passDesc.IAInputSignatureSize,
    &m_pInputLayout);

if (FAILED(result))
    assert(false); //or return
```



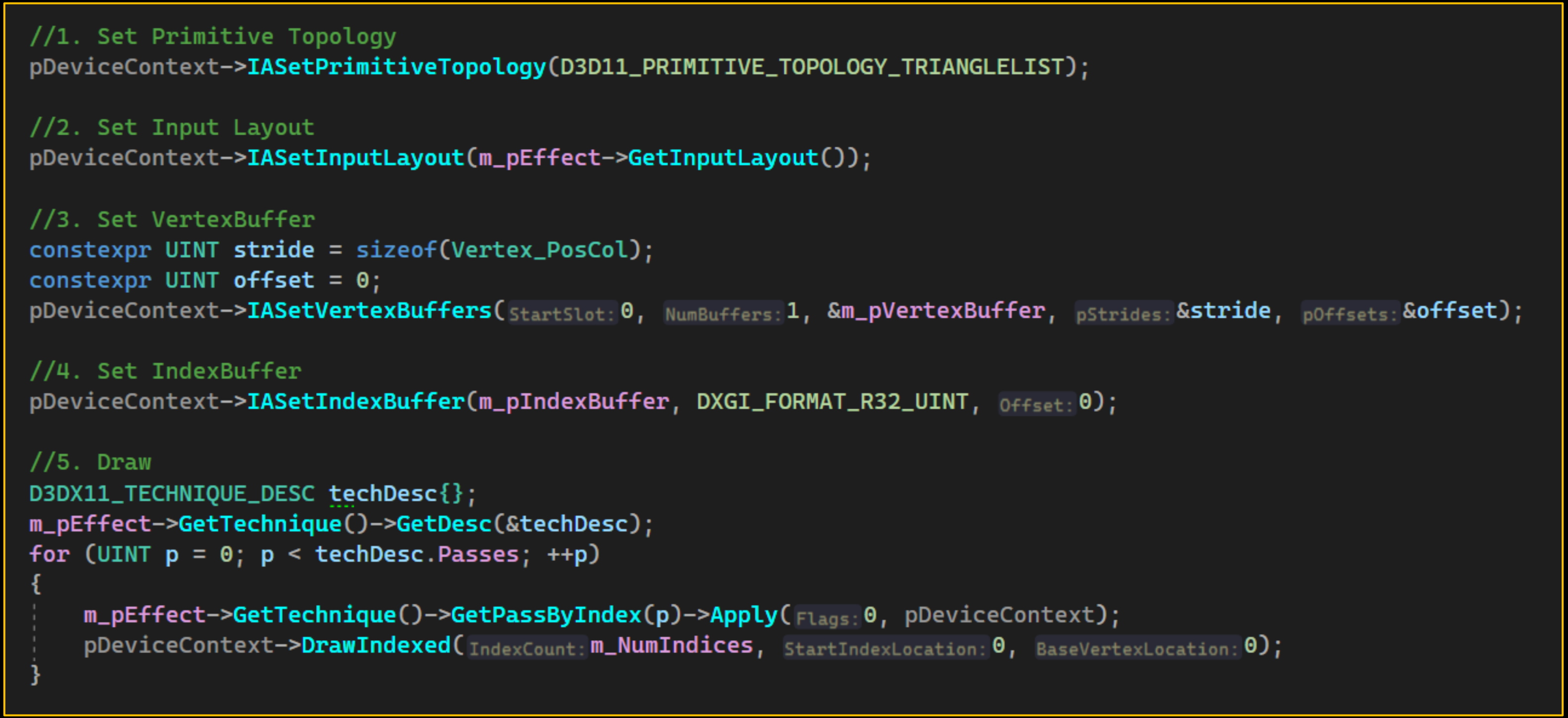
```
// Create vertex buffer
D3D11_BUFFER_DESC bd = {};
bd.Usage = D3D11_USAGE_IMMUTABLE;
bd.ByteWidth = sizeof(Vertex_PosCol) * static_cast<uint32_t>(vertices.size());
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;

D3D11_SUBRESOURCE_DATA initData = {};
initData.pSysMem = vertices.data();

HRESULT result = pDevice->CreateBuffer(&bd, &initData, &m_pVertexBuffer);
if (FAILED(result))
    return;
```



```
//Create index buffer
m_NumIndices = static_cast<uint32_t>(indices.size());
bd.Usage = D3D11_USAGE_IMMUTABLE;
bd.ByteWidth = sizeof(uint32_t) * m_NumIndices;
bd.BindFlags = D3D11_BIND_INDEX_BUFFER;
bd.CPUAccessFlags = 0;
bd.MiscFlags = 0;
initData.pSysMem = indices.data();
result = pDevice->CreateBuffer(&bd, &initData, &m_pIndexBuffer);
if (FAILED(result))
    return;
```

DirectX: Rendering

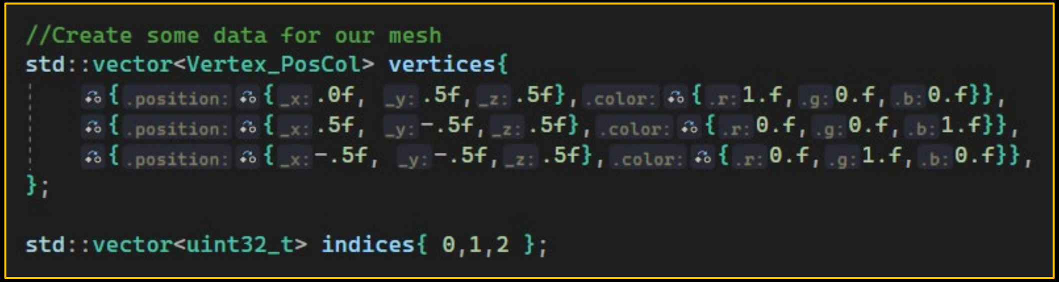
- Last thing we need to do is actual rendering ☺
- Whenever you want to render you have to set all the correct data in the device context, defining the state of the render pipeline, and call “render”.



```
//1. Set Primitive Topology
pDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

//2. Set Input Layout
pDeviceContext->IASetInputLayout(m_pEffect->GetInputLayout());

//3. Set VertexBuffer
constexpr UINT stride = sizeof(Vertex_PosCol);
constexpr UINT offset = 0;
pDeviceContext->IASetVertexBuffers(StartSlot: 0, NumBuffers: 1, &m_pVertexBuffer, pStrides: &stride, pOffsets: &offset);

//4. Set IndexBuffer
pDeviceContext->IASetIndexBuffer(m_pIndexBuffer, DXGI_FORMAT_R32_UINT, Offset: 0);

//5. Draw
D3DX11_TECHNIQUE_DESC techDesc{};
m_pEffect->GetTechnique()->GetDesc(&techDesc);
for (UINT p = 0; p < techDesc.Passes; ++p)
{
    m_pEffect->GetTechnique()->GetPassByIndex(p)->Apply(Flags: 0, pDeviceContext);
    pDeviceContext->DrawIndexed(IndexCount: m_NumIndices, StartIndexLocation: 0, BaseVertexLocation: 0);
}
```

DirectX: Rendering

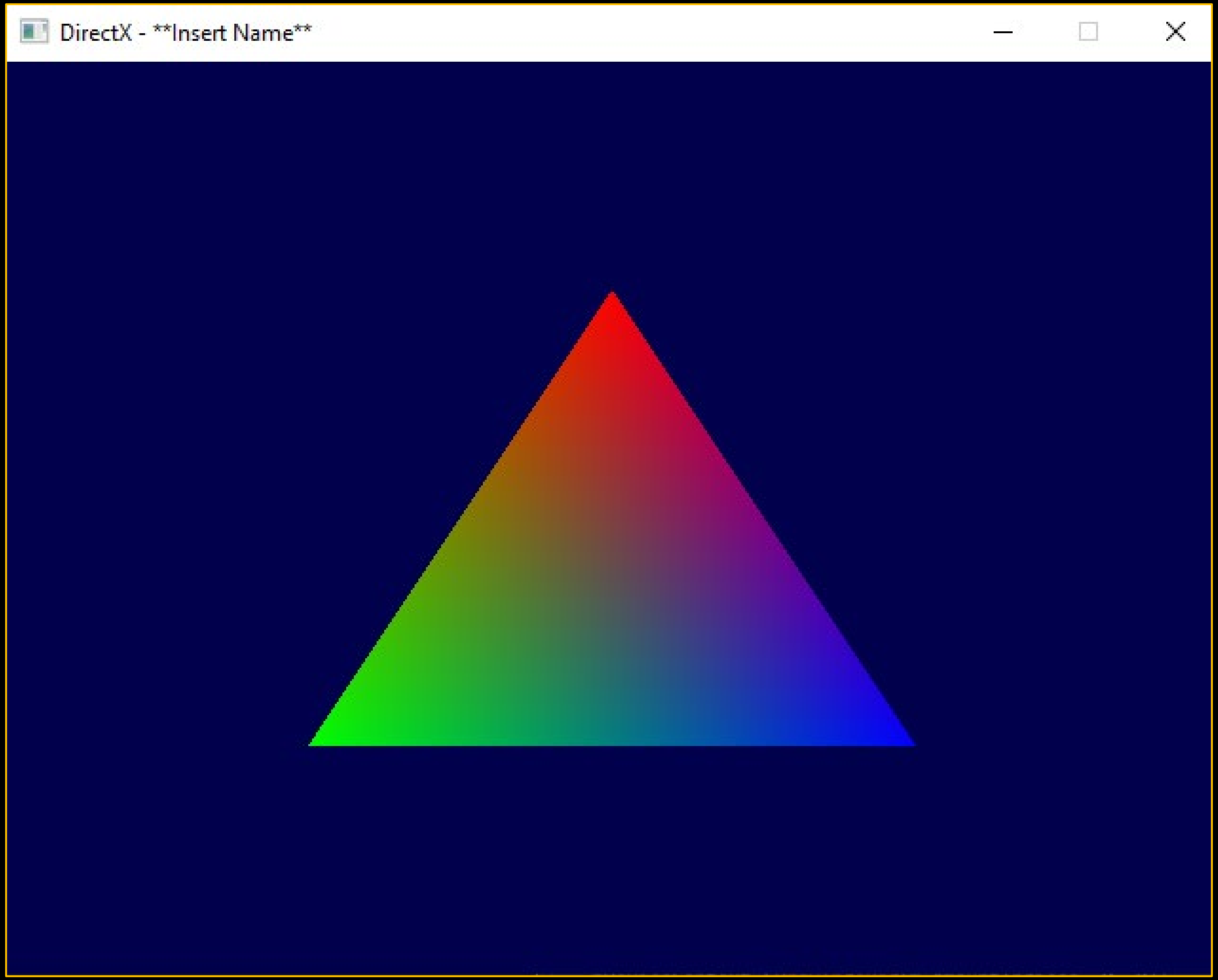
- Put the code from the previous slide in a Render function, create an instance of the primitive and call render before the Present in the renderer.
- Which data should I use for the vertex buffer?
- Remember DirectX uses a left-handed coordinate system. (Winding Order…)



```
//Create some data for our mesh
std::vector<Vertex_PosCol> vertices{
	{ .position: { _x: .0f,  _y: .5f, _z: .5f}, .color: { .r: 1.f, .g: 0.f, .b: 0.f}},
	{ .position: { _x: .5f,  _y: -.5f, _z: .5f}, .color: { .r: 0.f, .g: 0.f, .b: 1.f}},
	{ .position: { _x: -.5f,  _y: -.5f, _z: .5f}, .color: { .r: 0.f, .g: 1.f, .b: 0.f}},
};

std::vector<uint32_t> indices{ 0,1,2 };
```

- If you've done everything correctly, you should get the following awesome result.

DirectX: Rendering



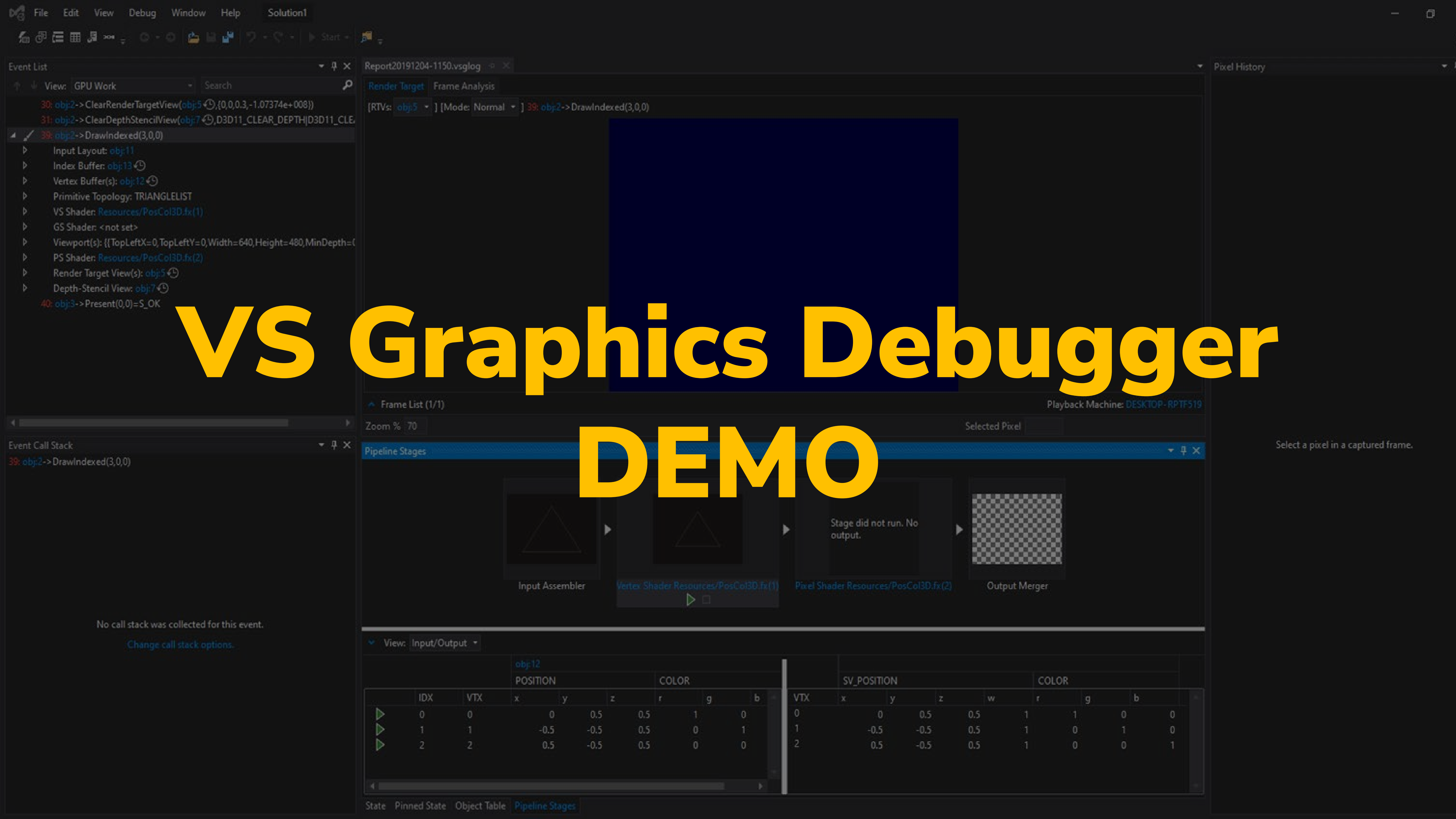
DirectX - **Insert Name**

DirectX: Rendering

- So, after this class you should really understand:
  - Graphics Pipeline & (basic) Shaders
  - Device vs Device Context
  - Descriptors
  - Resource vs Resource Views

- But what if you don’t get this result…. ☺

VS Graphics Debugger DEMO



View: GPU Work

```
30: obj:2->ClearRenderTargetView(obj:5,{0,0,0.3,-1.07374e+008})
31: obj:2->ClearDepthStencilView(obj:7,D3D11_CLEAR_DEPTH|D3D11_CLE
39: obj:2->DrawIndexed(3,0,0)
    Input Layout: obj:11
    Index Buffer: obj:13
    Vertex Buffer(s): obj:12
    Primitive Topology: TRIANGLELIST
    VS Shader: Resources/PosCol3D.fx(1)
    GS Shader: <not set>
    Viewport(s): {{TopLeftX=0,TopLeftY=0,Width=640,Height=480,MinDepth=(
    PS Shader: Resources/PosCol3D.fx(2)
    Render Target View(s): obj:5
    Depth-Stencil View: obj:7
40: obj:3->Present(0,0)=S_OK
```

[RTVs: obj:5] [Mode: Normal] 39: obj:2->DrawIndexed(3,0,0)

Frame List (1/1)

Playback Machine: DESKTOP-RPTF519

Zoom % 70

Selected Pixel

Select a pixel in a captured frame.

Event Call Stack

39: obj:2->DrawIndexed(3,0,0)

No call stack was collected for this event.

Change call stack options.

Vertex Shader Resources/PosCol3D.fx(1)

Stage did not run. No output.

Pixel Shader Resources/PosCol3D.fx(2)

View: Input/Output

		obj:12					
		POSITION			COLOR		
IDX	VTX	x	y	z	r	g	b
0	0	0	0.5	0.5	1	0	
1	1	-0.5	-0.5	0.5	0	1	
2	2	0.5	-0.5	0.5	0	0	

	SV_POSITION				COLOR		
VTX	x	y	z	w	r	g	b
0	0	0.5	0.5	1	1	0	0
1	-0.5	-0.5	0.5	1	0	1	0
2	0.5	-0.5	0.5	1	0	0	1

GOOD LUCK!