

Teaching Myself DirectX

The Device

A device is used to create resources and to enumerate the capabilities of a display adapter. A Direct3D device allocates and destroys objects, **renders primitives**, and **communicates with a graphics driver and the hardware**. In Direct3D 11, a device is separated into a device object for creating resources and a device-context object, which performs rendering. A device is represented with an [**ID3D11Device**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nn-d3d11-id3d11device) interface.

**Each application must have at least one device.**

[**D3D11CreateDevice**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-d3d11createdevice) or [**D3D11CreateDeviceAndSwapChain**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-d3d11createdeviceandswapchain) can be used to create a device.

Device Context

A device context is used to set **pipeline** state and generate rendering commands using the resources owned by a device.

Swap Chain

A swap chain is a collection of buffers that are used for displaying frames to the user. Each time an application presents a new frame for display, the first buffer in the swap chain takes the place of the displayed buffer. This process is called swapping or flipping.

A graphics adapter holds a **pointer** to a surface that represents the image being displayed on the monitor, called a **front buffer**. As the monitor is refreshed, the graphics card sends the contents of the front buffer to the monitor to be displayed. However, this leads to a problem when rendering real-time graphics. The heart of the problem is that monitor refresh rates are very slow in comparison to the rest of the computer. Common refresh rates range from 60 Hz (60 times per second) to 100 Hz. If your application is updating the front buffer while the monitor is in the middle of a refresh, the image that is displayed will be cut in half with the upper half of the display containing the old image and the lower half containing the new image. **This problem is referred to as tearing**.

Back buffering is the process of drawing a scene to an off-screen surface, called a back buffer. Note that any surface other than the front buffer is called an **off-screen surface because it is never directly viewed by the monitor**. By using a back buffer, an application has the freedom to render a scene whenever the system is idle (that is, no windows messages are waiting) without having to consider the monitor's refresh rate. Back buffering brings in an additional complication of how and when to move the back buffer to the front buffer.

The process of moving the back buffer to the front buffer is called surface flipping. Because the graphics card simply uses a pointer to a surface to represent the front buffer, a simple pointer change is all that is needed to set the back buffer to the front buffer. When an application asks Direct3D to present the back buffer to the front buffer, Direct3D simply "flips" the two surface pointers. The result is that the back buffer is now the new front buffer, and the old front buffer is the new back buffer. A surface flip is invoked whenever an application asks the Direct3D device to present the back buffer; however, Direct3D can be set up to queue the requests until a vertical sync occurs. This option is referred to as the Direct3D device's presentation interval. Note that the data in the new back buffer may not be reusable, depending on how an application specifies how Direct3D should handle surface flipping. Surface flipping is key in multimedia, animation, and game software; it is equivalent to the way you can do animation with a pad of paper. On each page, the artist changes the figures slightly, so that when you flip rapidly between sheets, the drawing appears animated.

Creating Device , Context and Swap Chain at once

HRESULT D3D11CreateDeviceAndSwapChain(

IDXGIAdapter \*pAdapter,

D3D\_DRIVER\_TYPE DriverType,

HMODULE Software,

UINT Flags,

const D3D\_FEATURE\_LEVEL \*pFeatureLevels,

UINT FeatureLevels,

UINT SDKVersion,

const DXGI\_SWAP\_CHAIN\_DESC \*pSwapChainDesc,

IDXGISwapChain \*\*ppSwapChain,

ID3D11Device \*\*ppDevice,

D3D\_FEATURE\_LEVEL \*pFeatureLevel,

ID3D11DeviceContext \*\*ppImmediateContext

);

1. A pointer to the video adapter to use when creating a [device](https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-devices-intro). Pass **NULL** to use the default adapter, which is the first adapter enumerated by [IDXGIFactory1::EnumAdapters](https://docs.microsoft.com/en-us/windows/desktop/api/dxgi/nf-dxgi-idxgifactory-enumadapters).
2. The [D3D\_DRIVER\_TYPE](https://docs.microsoft.com/en-us/windows/desktop/api/d3dcommon/ne-d3dcommon-d3d_driver_type), which represents the driver type to create.

typedef enum D3D\_DRIVER\_TYPE {

D3D\_DRIVER\_TYPE\_UNKNOWN = 0,

D3D\_DRIVER\_TYPE\_HARDWARE,

D3D\_DRIVER\_TYPE\_REFERENCE,

D3D\_DRIVER\_TYPE\_NULL,

D3D\_DRIVER\_TYPE\_SOFTWARE,

D3D\_DRIVER\_TYPE\_WARP

} ;

<https://docs.microsoft.com/en-us/windows/win32/api/d3dcommon/ne-d3dcommon-d3d_driver_type>

<https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-d3d11createdeviceandswapchain>

***\*\*This function takes swap-chain descriptions and creates device , swap-chain and device context .***

Direct3D interface pointers(Windows interface pointers) need to be called release function for realising the object(freeing the memory) ,for that we should use ComPtr (under wrl.h header)

**Microsoft::WRL::ComPtr<TypeName> ptr for *RAII***

Microsoft::WRL::ComPtr<ID3D11Device> pDevice;

Microsoft::WRL::ComPtr<IDXGISwapChain> pSwapChain;

Microsoft::WRL::ComPtr<ID3D11DeviceContext> pDeviceContext;

Microsoft::WRL::ComPtr<ID3D11Resource> pBackBuffer;

Microsoft::WRL::ComPtr<ID3D11RenderTargetView> pTarget;

**ID3D11 vs IDXGI**

IDXGI is those interfaces which doesn’t change as frequently as direct3D ( Direct3D9 , Direct3D10, Direct3D11 , Direct3D12)

*It basically factored all the enumeration, display and adapter management, and presentation stuff out of Direct3D. That way, all sorts of graphics APIs can coexist without a need to have separate mechanisms for these common tasks in each of them. It allows, e.g., all the Direct3D APIs (>= 10) to only be concerned with drawing 3D content into buffers and not care about where these buffers come from, or whether and how they are going to be displayed.*

***Giving swapchain description***

DXGI\_SWAP\_CHAIN\_DESC sd = { 0 };

sd.BufferDesc.Width = 0; // look at the window and use it's size

sd.BufferDesc.Height = 0;

sd.BufferDesc.RefreshRate.Numerator = 0; // pick the default refresh rates

sd.BufferDesc.RefreshRate.Denominator = 0;

sd.BufferCount = 1;// one back buffer -> one back and one front doubleb uffering

sd.BufferDesc.Format = DXGI\_FORMAT\_B8G8R8A8\_UNORM; // this is the color format (BGRA) we can also use (RGBA)

sd.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED; // not specifying any scaling because we want the renedred frame same as window size

sd.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED; // how buffer scaning will be done for copying all to video memory

sd.Flags = 0; // not setting any flags

sd.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT; // use the buffer for render target

sd.OutputWindow = handle;

sd.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD; // discard the effects for swapping frames

sd.Windowed = TRUE;

// for antialiasing we don't want it right now

sd.SampleDesc.Count = 1;

sd.SampleDesc.Quality = 0;

***Creating swapchain with device and immediate context***

D3D11CreateDeviceAndSwapChain(0, D3D\_DRIVER\_TYPE\_HARDWARE,

nullptr, 0, nullptr, 0,

D3D11\_SDK\_VERSION, &sd,

&pSwapChain, &pDevice, nullptr,

&pDeviceContext)

\*\*We also have to add a lib file **(d3d11.lib)** we can simply **#pragma comment(lib , “d3d11.lib”)** or goto **project\_properties -> linker -> input -> additional dependencies**

***Presenting new frame (or swapping the buffers of the swapchain)***

pSwapChain->Present(1u, 0u)

Present's first argument is the sync interval, which should be set to 1, and it will block until the monitor is ready to show the next frame.

<https://docs.microsoft.com/en-us/windows/win32/api/dxgi/nf-dxgi-idxgiswapchain-present>

***Clearing the backbuffer with a color***

pDeviceContext->ClearRenderTarget

**\*\*this function takes a render target view and a array of float values representing color (RGBA) .So we have to get the render target view pointer (or Creating a render target from the back buffer)**

// gain access to texture subresource in swap chain (back buffer)

Microsoft::WRL::ComPtr<ID3D11Resource> pBackBuffer;

//getting the id of the type to create (we want the back buffer so 0)

pSwap->GetBuffer( 0,\_\_uuidof(ID3D11Resource), &pBackBuffer );

//creating

Microsoft::WRL::ComPtr<ID3D11RenderTargetView> pTarget

pDevice->CreateRenderTargetView(

pBackBuffer,

nullptr,

&pTarget

);

float color[4] = {1.0f , 1.0f , 1.0f , 1.0f};

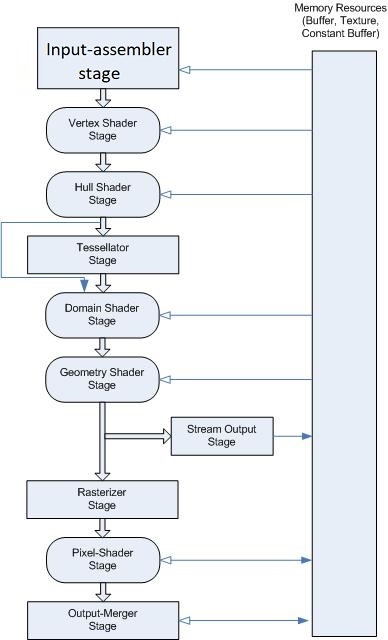
pDeviceContext->ClearRenderTargetView(pTarget ,color )

**\*\*\_\_uuidof is keyword used to retrive the GUID (an unique ID that is used in COM for doing stuffs)**

**This operator is a Microsoft language extension. It returns a GUID value from an expression. The expression can be an interface type name, a class name, or an interface pointer**

\*\*when we are using & operator on a com pointer object it releases the object it’s currently holding and returns a pointer to pointer.

**Setting Up the Pipeline and Drawing First Triangle**



<https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline>

The minimum requirement for creating triangles (or any primitives) are **vertex buffer(s)** , **Vertex shader** **, Input layout** , **Pixel shader** , **Viewport** , **Primitive topology and binding the render target**

**Creating and binding Vertex Buffer**

// base structure for every vertex type

struct VertexType

{

float x, y , z;

unsigned char r, g, b , a;

};

//creating an array of vertices that will be drawn

//below points are position for a triangle showing at the middle of the screen

//viewport's range is from -1.0f to 1.0f (x,y,z axis)

//triangle will be drawn using vertices at clockwise

VertexType vtx[] = {

{0.0 , 0.5 , 0.0 , 255 , 0 , 0 },

{0.5 , -0.5 , 0.0 , 0 , 255 , 0 },

{-0.5 , -0.5 , 0.0 , 0 , 0, 255 }

};

//vertex buffer description

D3D11\_BUFFER\_DESC bd = { 0 };

bd.ByteWidth = sizeof(vtx); //total array size

bd.Usage = D3D11\_USAGE\_DEFAULT;// how buffer data will be used (read/write protections for GPU/CPU)

bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;// What type of buffer would it be

bd.CPUAccessFlags = 0u;// we don't want any cpu access for now so setting it to 0 for now

bd.MiscFlags = 0u;// misscellinious flag for buffer configuration (we don't want it now either)

bd.StructureByteStride = sizeof(VertexType); // Size of every vertex in the array

//holds the data pointer that will be used in vertex buffer

D3D11\_SUBRESOURCE\_DATA subd = { 0 };

subd.pSysMem = vtx; // pointer to array so that it can copy all the array data to the buffer

Microsoft::WRL::ComPtr<ID3D11Buffer> VBuffer;

pDevice->CreateBuffer(&bd, &subd, &VBuffer);

UINT stride = sizeof(VertexType); // size of every vertex

UINT offset = 0u; // displacement after which the actual data start (so 0 because no displacement is there)

//statrting slot(from 0) , number of buffers(1 buffer) , pp ,

pDeviceContext->IASetVertexBuffers(0u, 1u, VBuffer.GetAddressOf(), &stride, &offset);

**Creating and Binding VertexShader**

Shaders:

Shaders are GPU programs that runs on GPU hardware. **Vertex shader** - in the most basic form , transforms vertices (bundles of information including spatial coordinates) from model space into screen space.

**\*\*homogeneous coordinates**

**In homogeneous coordinate system, two-dimensional coordinate positions (x, y) are represented by triple-coordinates.**

**Translation of point by the change of coordinate cannot be combined with other transformation by using simple matrix application. Such a combination is essential if we wish to rotate an image about a point other than origin by translation, rotation again translation.**

**To combine these three transformations into a single transformation, homogeneous coordinates are used**.

For three-dimensional geometric transformation, we can choose homogeneous parameter h to any non-zero value. For our convenience take it as one. Each three-dimensional position is then represented with homogeneous coordinates (x, y , z, 1).

**A semantic is a string attached to a shader input or output that conveys information about the intended use of a parameter. Semantics are required on all variables passed between shader stages**.

In general, data passed between pipeline stages is completely generic and is not uniquely interpreted by the system; arbitrary semantics are allowed which have no special meaning

**Creating Shader:**

//the input and output need to be labeled by semantics

struct VertexShaderOut // this is the ouput type

{

float4 color : COLOR; // this is the user defined semantic for color which will be passed to pixel shader

float4 pos : SV\_POSITION; // this is the SYSTEM\_VALUE SEMANTIC this is fixed (and defined by the API)

};

// main is the entry point and it takes 2 parameters

// first one is position and second one is color

// "POSITION" and "COLOR" is the semantic name which will be used to input the data from

// our cpu

// So our main returns a color and a position (for which every color that is returned with will be drawn)

VertexShaderOut main(float3 pos : POSITION , float4 col : COLOR)

{

VertexShaderOut Out;

Out.pos = float4(pos , 1.0f) ;

Out.color = col;

return Out;

}

**Loading Shader:**

For loading we have premade function named **D3DReadFileToBlob** for that we have to add a library “**D3DCompiler.lib**” we can also compile shders at runtime using it.

Microsoft::WRL::ComPtr<ID3D11VertexShader> vS; // shader pointer

Microsoft::WRL::ComPtr<ID3DBlob> blb; // holds the compiled shader bytecode

D3DReadFileToBlob(L"VertexShader.cso", &blb); // reading the bytecode and storing it to blob

pDevice->CreateVertexShader(blb->GetBufferPointer(), blb->GetBufferSize(), nullptr, &vS); // creating vertex shader

pDeviceContext->VSSetShader(vS.Get(), nullptr, 0u); // binding vertex shader

\*\*By Default the compiled shader outputs to the executable directory , but this can make a problem when we would try to run our program from VisualStudio (it would not find it unless we give it the full path or move it to the working directory). We can move it by right click->properties-> HLSL compiler -> Output Files -> Object file name -> $(ProjectDir)%(Filename).cso

\*\*If we are not creating a shader from Microsoft vertex shader template then

We have to set it’s type as a vertex shader by Right click on shader -> Properties -> HLSL Compiler -> General -> Shader type -> Vertex Shader

**Creating and Binding Input Lauout:**

Microsoft::WRL::ComPtr<ID3D11InputLayout> inpl;

//semantic name , semantic index , format , inputslot , offset , input data class , data step rate

D3D11\_INPUT\_ELEMENT\_DESC ied[] = {

//tells that the first 3 \* 4 \* 8 bits = 32 \* 3 = 96 bits of the vertex struct are for positions for every vertex

{"POSITION",0,DXGI\_FORMAT\_R32G32B32\_FLOAT,0,0,D3D11\_INPUT\_PER\_VERTEX\_DATA,0},

//and next 3 \* 1 \* 8 = 24 bits are color value for each of those vertex

// unorm for automaticall convert 0-255 range 0.0-1.0 range

{"COLOR",0,DXGI\_FORMAT\_R8G8B8A8\_UNORM,0,12u,D3D11\_INPUT\_PER\_VERTEX\_DATA,0}

};

//creating and setting

pDevice->CreateInputLayout(ied, (UINT)std::size(ied), blb->GetBufferPointer(), blb->GetBufferSize(), &inpl);

pDeviceContext->IASetInputLayout(inpl.Get());

The semantic index for the element. A semantic index modifies a semantic, with an integer index number. A semantic index is only needed in a case where there is more than one element with the same semantic. For example, a 4x4 matrix would have four components each with the semantic name

Like in a shader main(float3 pos : POSITION0 , float3 pos2 : POSITION1)

Then we should use the index 0/1

**Creating and binding Pixel Shader :**

//takes the color and returns the color with semantic for target position (it's a SYSTEM VALUE for render target)

float4 main(float4 col : COLOR) : SV\_TARGET

{

return col;

}

Microsoft::WRL::ComPtr<ID3D11PixelShader> ps; // shader pointer

D3DReadFileToBlob(L"PixelShader.cso", &blb); // reading file to blob

pDevice ->CreatePixelShader(blb->GetBufferPointer(), blb->GetBufferSize(), nullptr, &ps); // creating

pDeviceContext->PSSetShader(ps.Get(), nullptr, 0u); // setting

\*\* Rasterizer determines pixel positions ,so pixelshader does not needs to know about the pixel position it only needs to know / determine the color of the pixel on which the pixel shader is called.

\*\* Vertex shaders are called per vertex and pixel shaders are called per pixel

**Setting Render target:**

pDeviceContext->OMSetRenderTargets(1u, pTarget.GetAddressOf(), nullptr);

\*\* The output-merger (OM) stage generates the final rendered pixel color using a combination of pipeline state, the pixel data generated by the pixel shaders, the contents of the render targets, and the contents of the depth/stencil buffers. The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colors.

**Creating and Binding Viewport:**

\*\*a viewport is a two-dimensional (2D) rectangle into which a 3D scene is projected. In Direct3D, the rectangle exists as coordinates within a Direct3D surface that the system uses as a rendering target. The projection transformation converts vertices into the coordinate system used for the viewport. A viewport is also used to specify the range of depth values on a render-target surface into which a scene will be rendered (usually 0.0 to 1.0)

D3D11\_VIEWPORT vp = {};

vp.TopLeftX = 0;

vp.TopLeftY = 0;

vp.Width = 800; //screen height

vp.Height = 600; // screen width

vp.MaxDepth = 1; // maximum depth for z axis

vp.MinDepth = 0; // minimum depth for z axis

pDeviceContext->RSSetViewports(1u, &vp);

**Setting Primitive Topology:**

//draws the vertices as a list of traingles

pDeviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

**now we can draw by calling draw function on context**

pDeviceContext->Draw(std::size(vtx), 0u);

***Full Code***

**VertexShader**

//the input and output need to be labeled by semantics

struct VertexShaderOut // this is the ouput type

{

float4 color : COLOR; // this is the user defined semantic for color which will be passed to pixel shader

float4 pos : SV\_POSITION; // this is the SYSTEM\_VALUE SEMANTIC this is fixed (and defined by the API)

};

// main is the entry point and it takes 2 parameters

// first one is position and second one is color

// "POSITION" and "COLOR" is the semantic name which will be used to input the data from

// our cpu

// So our main returns a color and a position (for which every color that is returned with will be drawn)

VertexShaderOut main(float3 pos : POSITION , float4 col : COLOR)

{

VertexShaderOut Out;

Out.pos = float4(pos , 1.0f) ;

Out.color = col;

return Out;

}

**PixelShader**

//takes the color and returns the color with semantic for target position (it's a SYSTEM VALUE)

float4 main(float4 col : COLOR) : SV\_TARGET

{

return col;

}

**Main**

#include<Windows.h>

#include<d3d11.h>

#include<chrono>

#include<wrl.h>

#include<d3dcompiler.h>

#include<sstream>

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

#pragma comment(lib,"D3DCompiler.lib")

namespace wrl = Microsoft::WRL;

LRESULT \_stdcall dfWnd(HWND handle, UINT msgCode, WPARAM wparam, LPARAM lparam)

{

if (msgCode == WM\_CLOSE)

{

PostQuitMessage(10);

}

return DefWindowProc(handle, msgCode, wparam, lparam);

}

int \_stdcall WinMain(HINSTANCE hinstance, HINSTANCE hprev, LPSTR lpcmd, int cmdshow)

{

try

{

WNDCLASSEX wc = { 0 };

wc.cbSize = sizeof(wc);

wc.style = CS\_OWNDC;

wc.lpfnWndProc = dfWnd;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = hinstance;

wc.hIcon = nullptr;

wc.hCursor = nullptr;

wc.hbrBackground = nullptr;

wc.lpszMenuName = nullptr;

wc.lpszClassName = "ks\_Class";

wc.hIconSm = nullptr;

RegisterClassEx(&wc);

RECT rc = {};

rc.right = 800;

rc.bottom = 600;

AdjustWindowRect(&rc, WS\_CAPTION | WS\_MINIMIZEBOX | WS\_SYSMENU, false);

HWND handle = CreateWindowEx(0, "ks\_Class", "KS\_Window", WS\_CAPTION | WS\_MINIMIZEBOX | WS\_SYSMENU, CW\_USEDEFAULT, CW\_USEDEFAULT, rc.right - rc.left, rc.bottom - rc.top, nullptr, nullptr, hinstance, nullptr);

ShowWindow(handle, SW\_SHOW);

MSG msg;

wrl::ComPtr<ID3D11Device> pDevice;

wrl::ComPtr<IDXGISwapChain> pSwapChain;

wrl::ComPtr<ID3D11DeviceContext> pDeviceContext;

wrl::ComPtr<ID3D11Resource> pBackBuffer;

wrl::ComPtr<ID3D11RenderTargetView> pTarget;

DXGI\_SWAP\_CHAIN\_DESC sd = { 0 };

sd.BufferDesc.Width = 0; // look at the window and use it's size

sd.BufferDesc.Height = 0;

sd.BufferDesc.RefreshRate.Numerator = 0; // pick the default refresh rates

sd.BufferDesc.RefreshRate.Denominator = 0;

sd.BufferCount = 1; // one back buffer -> one back and one front double buffering

sd.BufferDesc.Format = DXGI\_FORMAT\_B8G8R8A8\_UNORM; // this is the color format (BGRA)

sd.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED; // not specifying any scaling because we want the renedred frame same as window size

sd.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED; // how buffer scaning will be done for copying all to video memory

sd.Flags = 0; // not setting any flags

sd.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT; // use the buffer for render target

sd.OutputWindow = handle;

sd.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD; // discard the effects for swapping frames

sd.Windowed = TRUE;

// for antialiasing we don't want it right now

sd.SampleDesc.Count = 1;

sd.SampleDesc.Quality = 0;

if(auto hrcode = D3D11CreateDeviceAndSwapChain(0, D3D\_DRIVER\_TYPE\_HARDWARE, nullptr, 0, nullptr, 0, D3D11\_SDK\_VERSION, &sd, &pSwapChain, &pDevice, nullptr, &pDeviceContext) ; FAILED(hrcode) )

{

throw hrcode;

}

pSwapChain->GetBuffer(0, \_\_uuidof(ID3D11Resource), &pBackBuffer);

pDevice->CreateRenderTargetView(pBackBuffer.Get(), nullptr, &pTarget);

// base structure for every vertex type

struct VertexType

{

float x, y , z;

unsigned char r, g, b , a;

};

//creating an array of vertices that will be drawn

//below points are position for a triangle showing at the middle of the screen

//viewport's range is from -1.0f to 1.0f (x,y,z axis)

//triangle will be drawn using vertices at clockwise

VertexType vtx[] = {

{0.0 , 0.5 , 0.0 , 255 , 0 , 0 },

{0.5 , -0.5 , 0.0 , 0 , 255 , 0 },

{-0.5 , -0.5 , 0.0 , 0 , 0, 255 }

};

//vertex buffer description

D3D11\_BUFFER\_DESC bd = { 0 };

bd.ByteWidth = sizeof(vtx); //total array size

bd.Usage = D3D11\_USAGE\_DEFAULT; // how buffer data will be used (read/write protections for GPU/CPU)

bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER; // What type of buffer would it be

bd.CPUAccessFlags = 0u; // we don't want any cpu access for now so setting it to 0 for now

bd.MiscFlags = 0u; // misscellinious flag for buffer configuration (we don't want it now either)

bd.StructureByteStride = sizeof(VertexType); // Size of every vertex in the array

//holds the data pointer that will be used in vertex buffer

D3D11\_SUBRESOURCE\_DATA subd = { 0 };

subd.pSysMem = vtx; // pointer to array so that it can copy all the array data to the buffer

Microsoft::WRL::ComPtr<ID3D11Buffer> VBuffer;

pDevice->CreateBuffer(&bd, &subd, &VBuffer);

UINT stride = sizeof(VertexType); // size of every vertex

UINT offset = 0u; // displacement after which the actual data start (so 0 because no displacement is there)

//statrting slot(from 0) , number of buffers(1 buffer) , pp ,

pDeviceContext->IASetVertexBuffers(0u, 1u, VBuffer.GetAddressOf(), &stride, &offset);

Microsoft::WRL::ComPtr<ID3D11VertexShader> vS; // shader pointer

Microsoft::WRL::ComPtr<ID3DBlob> blb; // holds the compiled shader bytecode

D3DReadFileToBlob(L"VertexShader.cso", &blb); // reading the bytecode and storing it to blob

pDevice->CreateVertexShader(blb->GetBufferPointer(), blb->GetBufferSize(), nullptr, &vS); // creating vertex shader

pDeviceContext->VSSetShader(vS.Get(), nullptr, 0u); // binding vertex shader

Microsoft::WRL::ComPtr<ID3D11InputLayout> inpl;

//semantic name , semantic index , format , inputslot , offset , input data class , data step rate

D3D11\_INPUT\_ELEMENT\_DESC ied[] = {

//tells that the first 3 \* 4 \* 8 bits = 32 \* 3 = 96 bits of the vertex struct are for positions for every vertex

{"POSITION",0,DXGI\_FORMAT\_R32G32B32\_FLOAT,0,0,D3D11\_INPUT\_PER\_VERTEX\_DATA,0},

//and next 3 \* 1 \* 8 = 24 bits are color value for each of those vertex

// unorm for automaticall convert 0-255 range 0.0-1.0 range

{"COLOR",0,DXGI\_FORMAT\_R8G8B8A8\_UNORM,0,12u,D3D11\_INPUT\_PER\_VERTEX\_DATA,0}

};

//creating and setting

pDevice->CreateInputLayout(ied, (UINT)std::size(ied), blb->GetBufferPointer(), blb->GetBufferSize(), &inpl);

pDeviceContext->IASetInputLayout(inpl.Get());

Microsoft::WRL::ComPtr<ID3D11PixelShader> ps; // shader pointer

D3DReadFileToBlob(L"PixelShader.cso", &blb); // reading file to blob

pDevice ->CreatePixelShader(blb->GetBufferPointer(), blb->GetBufferSize(), nullptr, &ps); // creating

pDeviceContext->PSSetShader(ps.Get(), nullptr, 0u); // setting

pDeviceContext->OMSetRenderTargets(1u, pTarget.GetAddressOf(), nullptr);

D3D11\_VIEWPORT vp = {};

vp.TopLeftX = 0;

vp.TopLeftY = 0;

vp.Width = 800; //screen height

vp.Height = 600; // screen width

vp.MaxDepth = 1; // maximum depth for z axis

vp.MinDepth = 0; // minimum depth for z axis

pDeviceContext->RSSetViewports(1u, &vp);

//draws the vertices as a list of traingles

pDeviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

while (true)

{

if (PeekMessage(&msg, nullptr, 0, 0, PM\_REMOVE) != 0)

{

if (msg.message == WM\_QUIT)

{

break;

}

TranslateMessage(&msg);

DispatchMessage(&msg);

}

if (FAILED(pSwapChain->Present(1u, 0u)))

{

//throw 1;

}

pDeviceContext->Draw(std::size(vtx), 0u);

}

}

catch (HRESULT hrcode)

{

char\* msgBuff;

FormatMessage(

FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |

FORMAT\_MESSAGE\_FROM\_SYSTEM | FORMAT\_MESSAGE\_IGNORE\_INSERTS,

nullptr, hrcode, MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT) , reinterpret\_cast<LPSTR>(&msgBuff) , 0 , nullptr);

MessageBox(nullptr, msgBuff, "Error", MB\_ICONERROR);

LocalFree(msgBuff);

}

return 0;

}

**LetsDrawAPentagon**

***Drawing Indexed:***

***The new verteices will be:***

VertexType vtx[] = {

{0.0 , 0.5 , 0.0 , 255 , 0 , 0 },

{0.5 , -0.5 , 0.0 , 0 , 255 , 0 },

{-0.5 , -0.5 , 0.0 , 0 , 0, 255 },

{-0.5 , 0.0 ,0.0 , 0 , 0 , 255 },

{0.5 , 0.0 ,0.0 , 0 , 255 , 0},

};

**Creating Index buffer and drawing Indexed:**

const unsigned short indices[] = {

0 , 1 , 2,

0 , 2 , 3,

0 , 4 , 1

};

Microsoft::WRL::ComPtr<ID3D11Buffer> IndxBuff;

D3D11\_BUFFER\_DESC indesc = { 0 };

indesc.ByteWidth = sizeof(indices);

indesc.Usage = D3D11\_USAGE\_DEFAULT;

indesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indesc.StructureByteStride = sizeof(unsigned short);

indesc.CPUAccessFlags = 0u;

indesc.MiscFlags = 0u;

D3D11\_SUBRESOURCE\_DATA isd = { 0 };

isd.pSysMem = indices;

pDevice->CreateBuffer(&indesc, &isd, &IndxBuff);

pDeviceContext->IASetIndexBuffer(IndxBuff.Get(), DXGI\_FORMAT\_R16\_UINT, 0u);

**Drawing Indexed:**

pDeviceContext->DrawIndexed(std::size(indices) , 0u , 0u);

**Conastant Buffer:**

A constant buffer allows you to efficiently supply shader constants data to the pipeline. You can use a constant buffer to store the results of the stream-output stage. Conceptually, a constant buffer looks just like a single-element vertex buffer.

A constant buffer can only use a single bind flag (**D3D11\_BIND\_CONSTANT\_BUFFER**), which cannot be combined with any other bind flag. To bind a shader-constant buffer to the pipeline, call one of the following methods: [**ID3D11DeviceContext::GSSetConstantBuffers**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-id3d11devicecontext-gssetconstantbuffers), [**ID3D11DeviceContext::PSSetConstantBuffers**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-id3d11devicecontext-pssetconstantbuffers), or [**ID3D11DeviceContext::VSSetConstantBuffers**](https://docs.microsoft.com/en-us/windows/desktop/api/D3D11/nf-d3d11-id3d11devicecontext-vssetconstantbuffers).

Each shader stage allows up to 15 shader-constant buffers; each buffer can hold up to 4096 constants.

const int angle = 90;

//a rotation around z axis matrix

const float TransformMatrix[4][4] = {

{std::cos(angle) , -std::sin(angle) , 0.0f , 0.0f},

{std::sin(angle) , std::cos(angle) , 0.0f , 0.0f},

{ 0.0f , 0.0f , 1.0f , 0.0f},

{ 0.0f , 0.0f , 0.0f , 1.0f}

};

Microsoft::WRL::ComPtr<ID3D11Buffer> ConstBuff;

D3D11\_BUFFER\_DESC cbuffdsc = { 0 };

cbuffdsc.ByteWidth = sizeof(TransformMatrix);

cbuffdsc.Usage = D3D11\_USAGE\_DYNAMIC; // because we are gonna change it from cpu side (not now but later)

cbuffdsc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

cbuffdsc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE; // giving access to cpu so that we can change it

D3D11\_SUBRESOURCE\_DATA cbuffdt = { 0 };

cbuffdt.pSysMem = TransformMatrix;

pDevice->CreateBuffer(&cbuffdsc, &cbuffdt, &ConstBuff);

DeviceContext->VSSetConstantBuffers(0u, 1u, ConstBuff.GetAddressOf());

***Accessing the cbuffer from vertex shader***

//the input and output need to be labeled by semantics

struct VertexShaderOut // this is the ouput type

{

float4 color : COLOR; // this is the user defined semantic for color which will be passed to pixel shader

float4 pos : SV\_POSITION; // this is the SYSTEM\_VALUE SEMANTIC this is fixed (and defined by the API)

};

//cbuffer for constant buffer

cbuffer cbf

{

row\_major matrix transform; // the matrix (in cpu side the matrix is in row order but in gpu it's in coloumn order so fixing it)

};

// main is the entry point and it takes 2 parameters

// first one is position and second one is color

// "POSITION" and "COLOR" is the semantic name which will be used to input the data from

// our cpu

// So our main returns a color and a position (for which every color that is returned with will be drawn)

VertexShaderOut main(float3 pos : POSITION , float4 col : COLOR)

{

VertexShaderOut Out;

Out.pos = mul(float4(pos , 1.0f) , transform);

Out.color = col;

return Out;

}

**Using DirectX Math library:**

The DirectXMath API provides SIMD-friendly C++ types and functions for common linear algebra and graphics math operations common to DirectX applications. The library provides optimized versions for Windows 32-bit (x86), Windows 64-bit (x64), and Windows on ARM/ARM64 through SSE, AVX, and ARM-NEON intrinsics support in the Visual C++ compiler.

***Using math functions for rotation***

Replace the hardcoded rotation matrix with this

DirectX::XMMATRIX TransformMatrix = DirectX::XMMatrixRotationZ(angle);

That’s all we need to do. The XMMATRIX struct is 4 x 4 matrix ,same as we created before.

**Rendering a 3D cube:**

*Vertices for the cube:*

{ -1.0f,-1.0f,-1.0f , 255 , 0 , 0 },

{ 1.0f,-1.0f,-1.0f , 0 , 255 , 0 },

{ -1.0f,1.0f,-1.0f , 0 , 0 , 255 },

{ 1.0f,1.0f,-1.0f , 255 , 255,0},

{ -1.0f,-1.0f,1.0f , 0 , 255 , 255 },

{ 1.0f,-1.0f,1.0f ,255 , 0 , 255 },

{ -1.0f,1.0f,1.0f ,200 , 0 , 255},

{ 1.0f,1.0f,1.0f , 100 , 200 , 150 },

*Indices for Drawing this cube:*

0,2,1, 2,3,1,

1,3,5, 3,7,5,

2,6,3, 3,6,7,

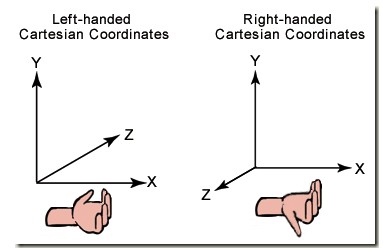
4,5,7, 4,7,6,

0,4,2, 2,4,6,

0,1,4, 1,5,4

**Transformation cbuffer for rotating and translating**

DirectX::XMMATRIX TransformMatrix = DirectX::XMMatrixTranspose(DirectX::XMMatrixRotationZ(angle) \* DirectX::XMMatrixRotationX(angle) \* DirectX::XMMatrixRotationY(angle) \* DirectX::XMMatrixTranslation(0.0 , 0.0 , 4.0f)\* DirectX::XMMatrixPerspectiveLH(1.0f , 3.0f / 4.0f , 0.5 , 10.0f));



\*\*A Projection is a design technique used to display a n-dimensional object on a n − 1 dimensional surface. These projections rely on visual perspective and aspect analysis to project a complex object for viewing capability on a simpler plane

\*\*Simply how we are looking through the camera to the 3d world

Here 3.0f / 4.0f is the aspect ratio of the screen (600 / 800) (w / d) so that our model looks OK!!

**Giving Color to Each face of the cube:**

We wil make a const buffer of colors and give that to pixel shader , and the pixel shader will get the Triangle ID look up in that buffer and draw that pixel with the color.

**VertexShader**

//cbuffer for constant buffer

cbuffer cbf

{

matrix transform; // the matrix (in cpu side the matrix is in row order but in gpu it's in coloumn order so fixinf it)

};

float4 main(float3 pos : POSITION) : SV\_POSITION

{

return mul(float4(pos , 1.0f) , transform);

}

now our vertex shader would only take the positions

**PixelShader**

cbuffer cb

{

float4 color[6];

};

//takes the color and returns the color with semantic for target position (it's a SYSTEM VALUE)

float4 main(uint ID : SV\_PRIMITIVEID) : SV\_TARGET

{

return color[ID / 2];

}

The system variable SV\_PRIMITIVE knows which triangle it’s working with so for total 12 triangles 2 triangles will have same color(each face contains 2 triangles) ,as bcause our vertices and indices are orderd(mostly vertices) each face has same color.

const float color[][4] = { {0.0f , 1.0f , 1.0f} ,

{1.0f , 0.0f , 1.0f} ,

{0.5f , 0.2f , 1.0f},

{0.8f , 0.0f , 0.5f},

{1.0f , 1.0f , 0.5f},

{0.6f , 0.2f , 0.2f}

};

Microsoft::WRL::ComPtr<ID3D11Buffer> ColorBuff;

D3D11\_BUFFER\_DESC cbuffdsc = { 0 };

cbuffdsc.ByteWidth = sizeof(color);

cbuffdsc.Usage = D3D11\_USAGE\_DYNAMIC;

cbuffdsc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

cbuffdsc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

D3D11\_SUBRESOURCE\_DATA cbuffdt = { 0 };

cbuffdt.pSysMem = color;

pDevice->CreateBuffer(&cbuffdsc, &cbuffdt, &ColorBuff);

pDeviceContext->PSSetConstantBuffers(0u, 1u, ColorBuff.GetAddressOf());

**The input layout also has been changed**

D3D11\_INPUT\_ELEMENT\_DESC ied[] = {

//tells that the first 3 \* 4 \* 8 bits = 32 \* 3 = 96 bits of the vertex struct are for positions for every vertex

{"POSITION",0,DXGI\_FORMAT\_R32G32B32\_FLOAT,0,0,D3D11\_INPUT\_PER\_VERTEX\_DATA,0}};