# [Window Messages] 4

1.std::map和std::unordered\_map, std::map使用二叉搜索树，它存储的键值对总是按键（key）排序的（可定义operator() > , <等 ）。而std::unordered\_map不保证任何特定的顺序，元素的位置由键的哈希值决定,搜索时间为常数

2.仿函数operator() 跟构造函数的区别

构造函数为特殊成员函数 在对象被创建的时候调用  
Object b(10); 构造函数  
b(10); 调用operator()成员函数

1. constexpr是C++11引入的一个关键字，用于指明表达式或对象的值在编译时就可以确定。这样，编译器就可以在编译时计算出这些值，提高运行时的效率

# [WM\_CHAR / Mouse] 5

1. WM\_CHAR 里面相对（WM\_KEYDOWN）区分大小写,且WM\_CHAR 是由TranslateMessage()产生的

# [Window Framework] 6

1. Window窗口封装类

ChiliWin.h

1. 主要就是包含Window头文件 #include <Windows.h>  
   由于Window.h包含超多命名空间 可在导入前取消定义 如取消Windows里面的Max Min宏定义  
   #define NOMINMAX

定义Windows sdk版本信息

// target Windows 7 or later

#define \_WIN32\_WINNT 0x0601

#include <sdkddkver.h>

1. Window接口类

WindowClass 注册 以及 取消注册 WinClass

Private static WindowClass wndClass; 在Cpp里面定义时候就会调用构造函数注册WindowClass  
Window::WindowClass Window::WindowClass::wndClass;

1. Hanky-panky
2. 构造函数最后加noexcept 主要是为了提高性能

# [Custom Icons / Error Handling]7

# [Keyboard / WM\_SYSKEYDOWN]8

1. 
2. Server Clinet

Server 作为Windows ，Clinet作为我们的Game

创建Kbd类来管理 关于键盘的消息 通过API来通信 这样的设计更具模块化

1. unsigned char keycode

来自WM\_KEYDOWN message , wParam 其所有值都在 1 Byte以内，等于8bit   
2的8次方 所以在unsigned 的情况下是 0-256 , 否贼为-128 -127

1. Windows 为 Keyboard 友原类，Windows 可访问Keyboard 私有函数

Keyboard 是Windows embedded object 生命周期绑定

1. VK\_MENU 表示ALT 为系统按键 like F10

case WM\_KEYDOWN:

case WM\_SYSKEYDOWN:

# [Mouse / Mouse Capture]9

1. VM\_MOUSE里

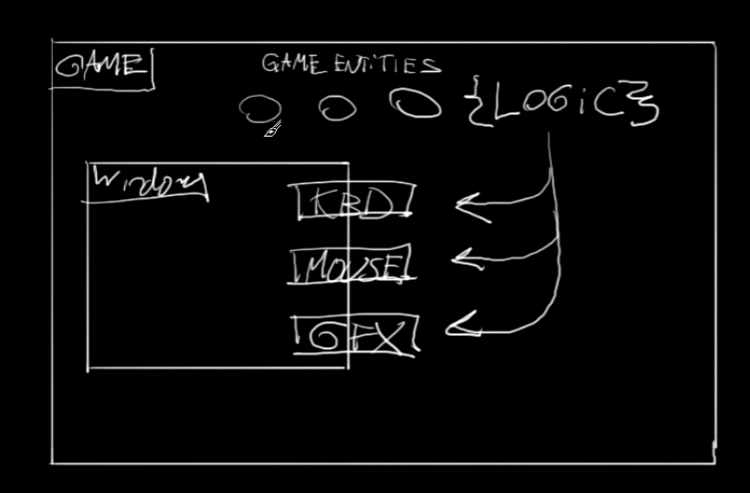
lParam表示 POINTS ; const POINTS pt = MAKEPOINTS(lParam);

wParam 表示鼠标中间键

GET\_WHEEL\_DELTA\_WPARAM(wParam) 大于表示WheelUp

1. kbd 跟 mouse Event的大致流程为  
   WinMain里面创建Window  
   Window不断获得消息，对应的bind到kbd 跟 mouse Event里面  
   WinMain主循环中通过wnd.kbd/wnd.mouse Read（）读出queue里面的Event根据  
   类型做出各种处理
2. 写入类为 ostringstream oss; oss.str();

# [App class / PeekMessage]10

1. 

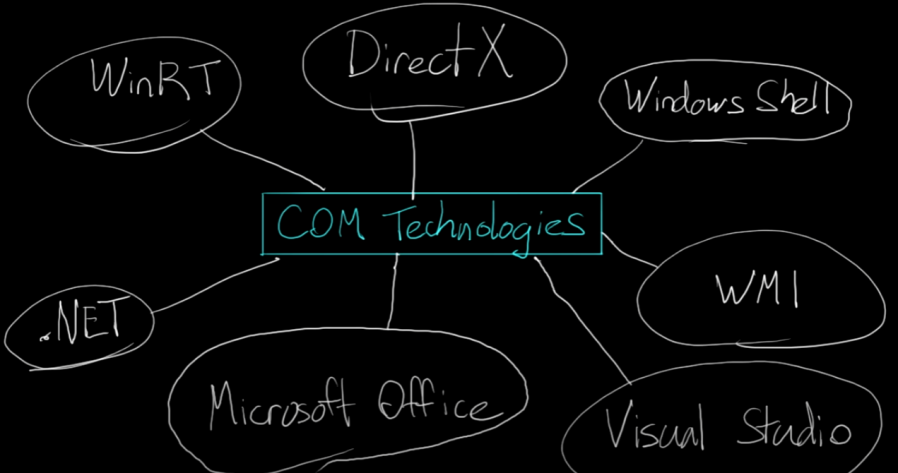
创建一个APP类 来代表GameLogic

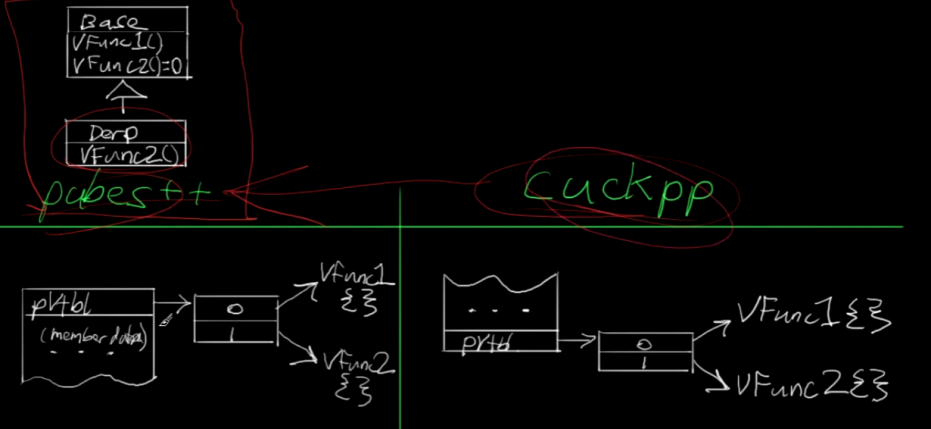
2. std::setprecision(1)<< std::fixed << t <<"s";  
std::setprecision(1) 浮点精度为1 3.1 std::fixed 固定输出精度 即使为0 两者一般连用

1. std::optional<> 可以返回一个不存在值 比如容器 返回optional<>类型也是OK 的不会冲突

# [COM (Component Object Model)]11

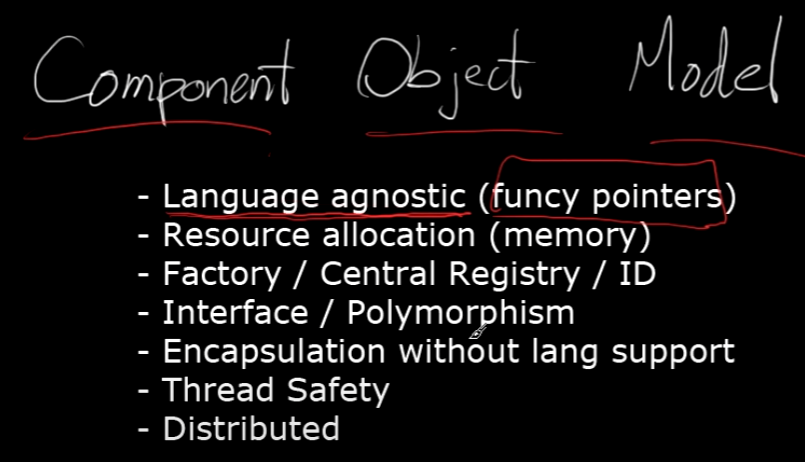
1.



不同编译器对编译出Binary有不同的layOut如下  
 

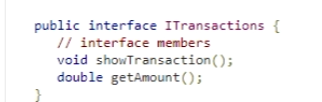
左边vpoint 位于第一个， 右边vpoint在数据最下面 这样右边去调用左边编译出的dll,他的  
Virtual dynamic dispatch mechanism（虚拟动态调度机制） 会去找位于类的最下面指针 这会造成错误

也就是说明不同编辑器 的编译出的二进制是不兼容的  
2、



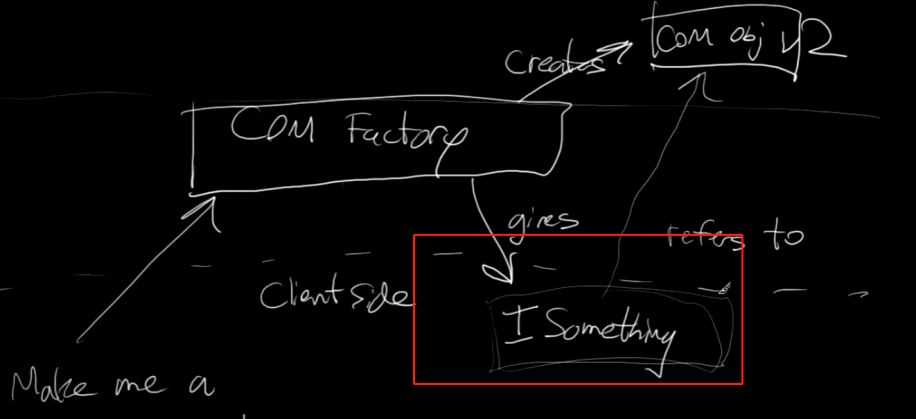
1. Interface

C# 是有Interface关键字



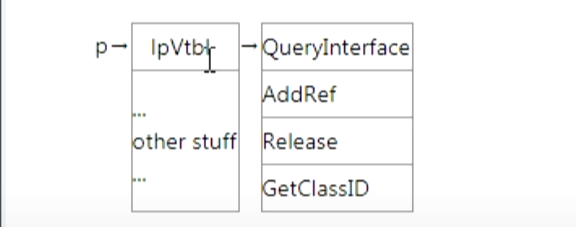
因为C# 不能继承两个类 但是接口可以

C++ 可以多继承，C++里面的接口类 写成Pure virtral function

1. COM中如何使用  
   

在COM 体系中 Create一个Obj后 不会直接使用这个Object而会使用他返回的接口

这样的好处是 当你改变这个OBJ 只要接口不变 Clinet不需要修改（OBJ实现已经变了 只是接口不变）

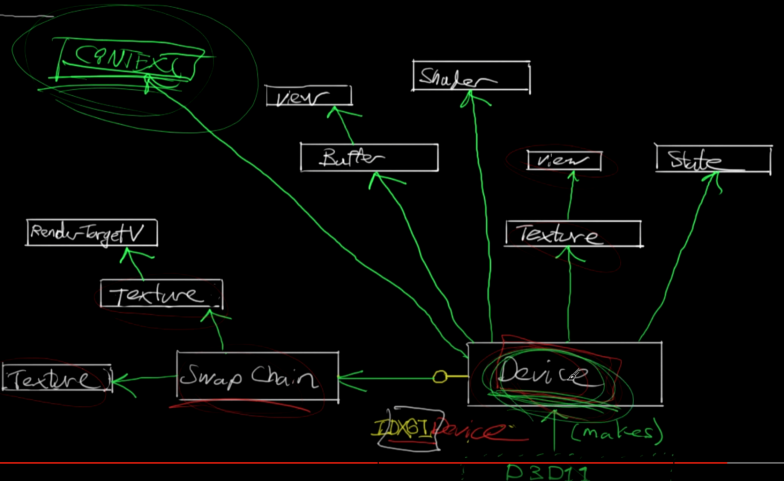


AddRef() Release() QueryInterface()是三个必须实现的接口函数

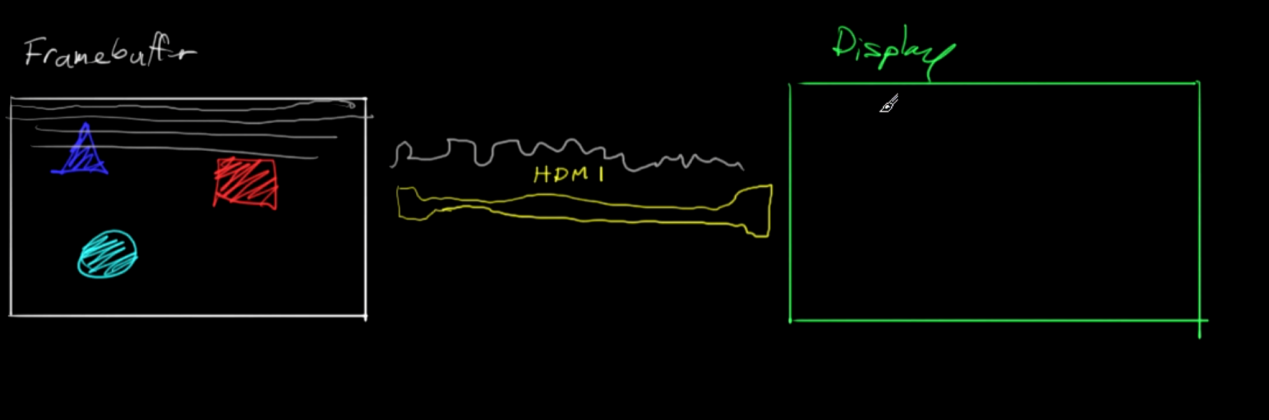
ComObject的生命周期跟shared\_ptr一样

# [D3D Architecture / Swap Chain]12

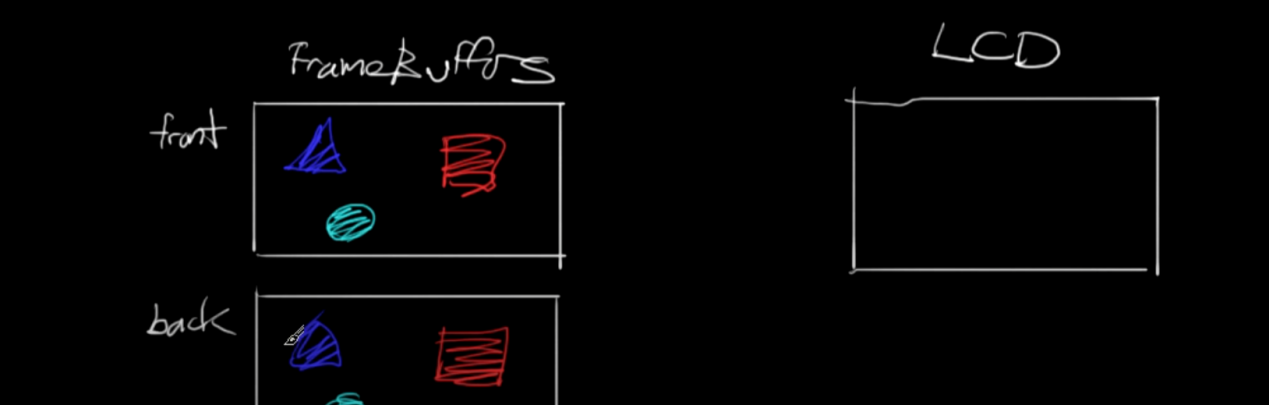
1.



这些的父类都是Device(这是一个COM Object)



所有的Color（pixel）都在一个frameBuffer中 从左到右 水平扫描 row by row(到底后会重头开始) 转到成HDMI 格式 再展示到Monitor上



singleBuffer 会造成画面闪烁 一边清除 一边扫描,也会造成渲染不完全

DoubleBuffer 可以解决这一问题 back用于计算一个scene, front不会去处理 单纯扫描

两者切换 可以把back copy 到 front, 也可以改名字 back变成front(flipping presentation)

tearing(画面撕裂) 的问题，swap Buffer 在完成一帧渲染 也就是回到最开始的位置 一段Gap

如果不是 会造成渲染问题 一般我们会开启Vsync 强行等待scan一个buffer的完成 再去swapBuffer,或者 我们可以创建第三个buffer 甚至更多，越多的buffer画面会越流畅 但是延迟会越高

2.

可以从DXGIDevice拿到swapChain 再拿到back Buffer,从backBuffer创建RenderTarget 绑定到PipeLine里面

1. Context

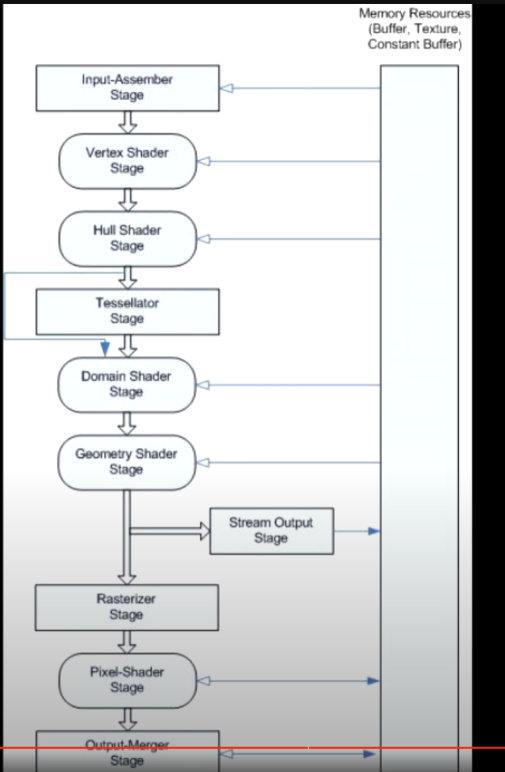
Context，Device 都能代表Render Adpat

Device主要用在SetUp() 创建各种Object, Context 用在Drawing(各种命令)

1. DXGI

IDXGIDevice 是 DirectX 图形接口（DirectX Graphics Interface，DXGI）中的一个接口，用于表示一个可用于进行图形渲染的设备。

IDXGIDevice 接口提供了与图形设备交互的功能



# [Device Init / Fill Screen / Present]13

1.创建Graphics类

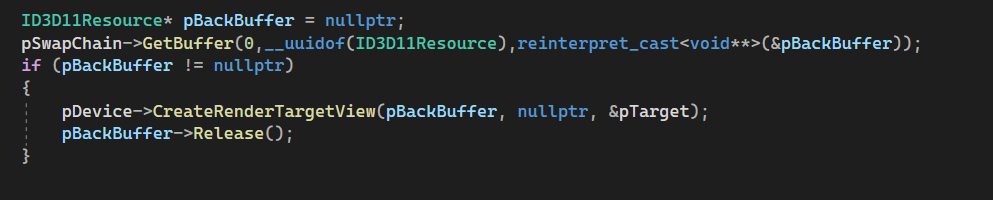
D3D11CreateDeviceAndSwapChain（）

创建了Device,SwapChain,DeviceContext等重要的类

2.Graphics 作为内部对象放在Window 内，目前Winodw类有三个对象Keyboard,Mouse,Graphics

3.RenderTarget的创建过程

RenderTarget 从SwapChain->BackBuffer-> Texture->RenderTarget



ClearBuffer 就是通过RenderTarget

pContext->CleraRenderTargetView();

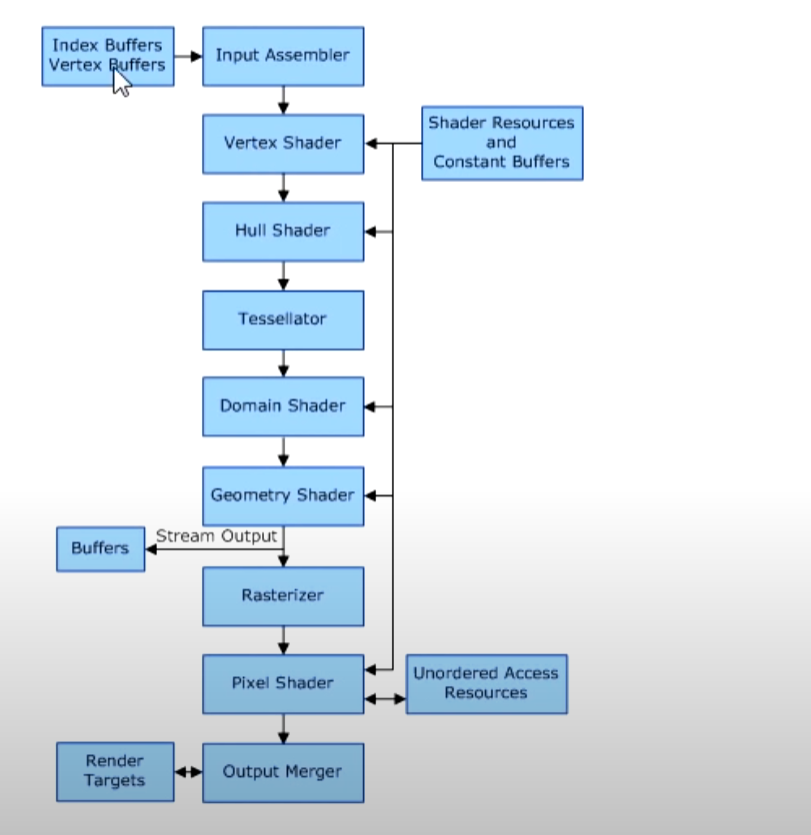
# [Debug Layer Diagnostics]14

# [ComPtr Smart Pointer]15

1.Microsoft::WRL::ComPtr<IDXGISwapChain>

# [First Triangle Pt 1]16

1.D3D11 Draw PipeLine



2.渲染的顺序

1）最终是用pContext->Draw();

首先先创建VertexBuffer,再用InputAssemble组装 IASetVertexBuffers

在调用Draw之前都需要创建及绑定VertexShader,pixelShader

D3DReadFileToBlob

pDevice->CreateVertexShader

pContext->VSSetShader

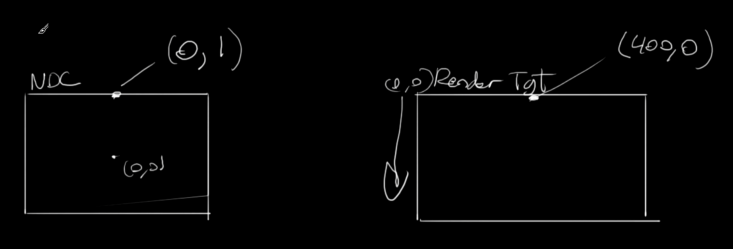
shader hlsl代表的high-level Shader Language ,编译完成的文件是.cso

# [First Triangle Pt 2]17

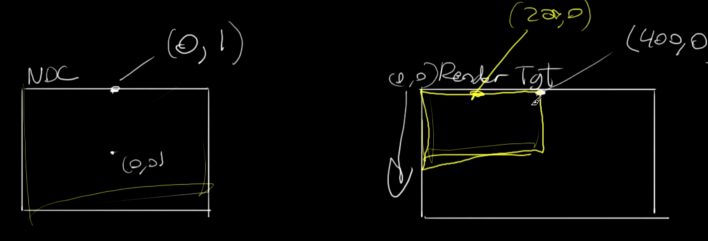
1.创建及绑定pixelShader,最后一步pipeline 是MergeOutput,需要RenderTarget

pContext - > OMSetRenderTarget

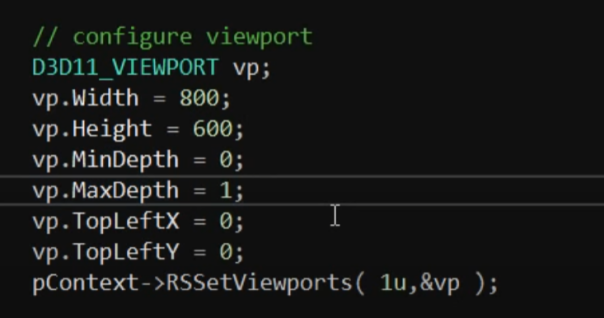
2.viewPort



坐标的Mapping



可以划分RenderTarget,也就是多屏幕，下面可以渲染UI等



3.目前一个Draw的流程

1)创建及绑定一个VertexBuffer，须定义Vertex 和 Vertices 并在描述中绑定顶点

pDevice->CreateBuffer, pContext->IASetVertexBuffers

2)创建vertexShader & pixelShader

将shader文件读入Blob 再绑定

D3DReadFileToBlob, pDevice->CreateVertexShader, pContext->VSSetShader

3)创建InputLayout

主要用于描述Vertex数据结构的意义 与 VertexShader里面对应

pDevice->CreateInputLayout，pContext->IASetInputLayout(pInputLayout.Get());

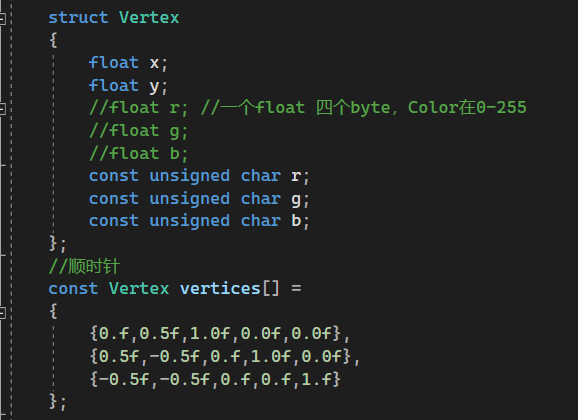
4）设定拓跋 目前是三角 IASetInputLayout

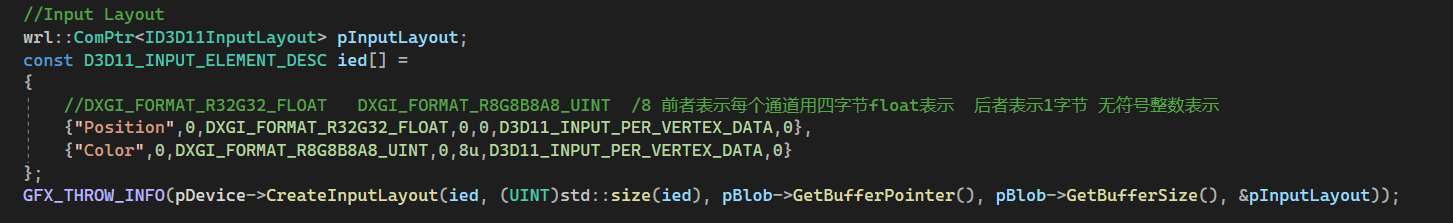
5)绑定Viewport RSSetViewports

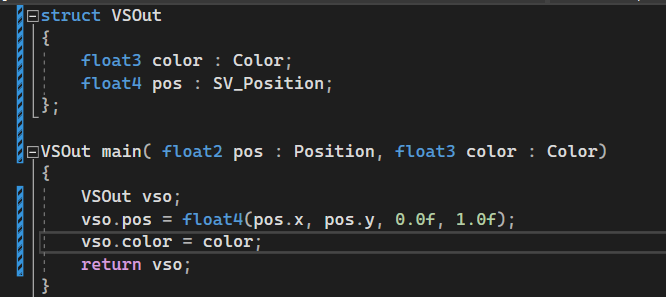
6)绑定RenderTarget OMSetRenderTargets

# [Pipeline Experiments]18

1.给Vertex加入颜色







需要改变Vetex结构体，Layout，VertexShader ,PixelShader

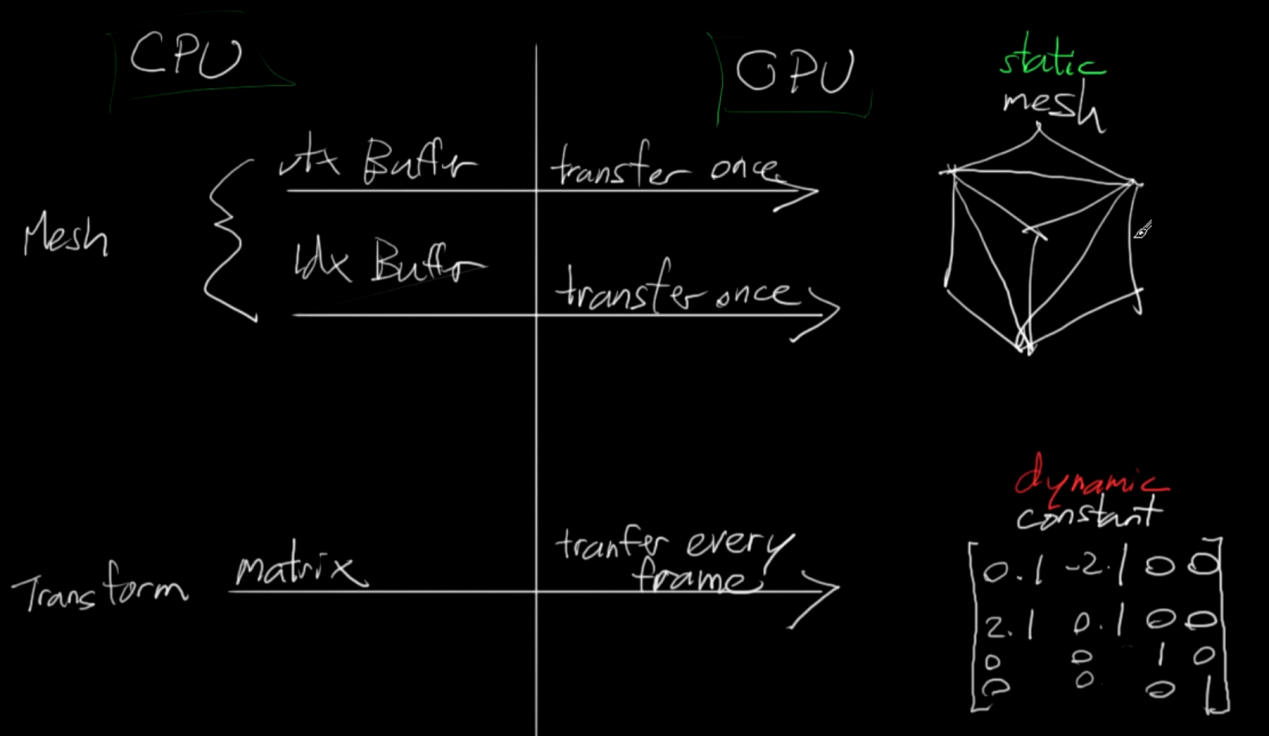
2.DXGI\_FORMAT\_R8G8B8A8\_UNORM 跟 DXGI\_FORMAT\_R8G8B8A8\_UINT的区别

前者输出到Color会Mapping 255->1 ,128 ->0.5

3.Vertex结构体里面也可以包含结构体

# [Constant Buffers]19

1.



为什么顶点计算要放在GPU

1)在CPU每帧进行运算 bandwith消耗太大

bandwidth"（带宽）是指在特定时间内传输数据的能力或速率

在存储设备中，带宽指的是设备从存储介质（如硬盘、固态驱动器）读取或写入数据的速率。存储设备的带宽决定了数据传输的效率和速度。

2)dynamic constant Buffer

tMatrix Transfor every frame, 不是更新1千多个顶点 还是更新16个点 4x4的矩阵

3)GPU High parallel processor"（高并行处理器）

2.dynamic constant Buffer

Matrix 不会绑定在Vertex,绑定在Dynamic constant Buffer, 这个Buffer，Shader draw都能够调用

# [DirectXMath]20

1. #include <DirectXMath.h>

M DxMath数据结构做过优化 不要直接使用他的使用 而是用API

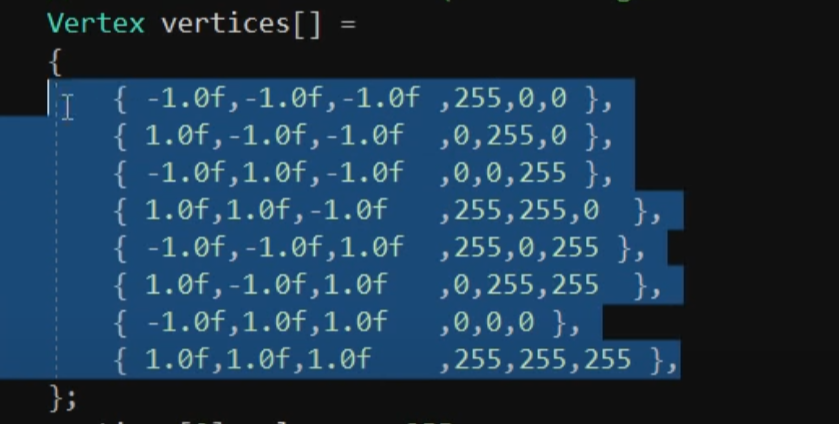
Dx::XMMatrix

# [3D Cube / Z-buffer]21

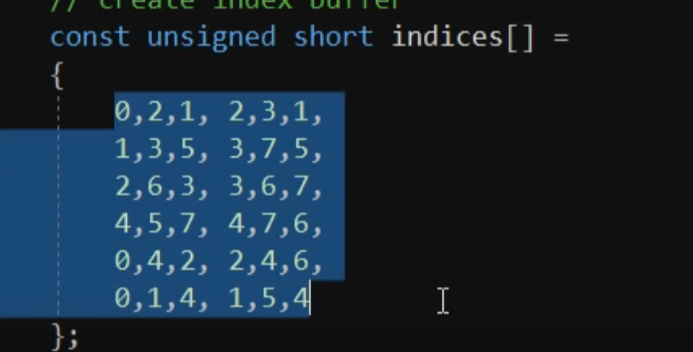
1.Draw Cube

CubeVertices

|  |
| --- |
|  |



CubeIndices



需要调整Vertex数据结构 layout,添加透视投影矩阵，VertexShader等

2.Cube Face Draw Solid Color

两种方案 每个顶点给一个颜色 需要单独的16个点 增加了一倍

另外是使用各一个ConstantBuffer 存的是颜色，给每个三角形 一个单独的颜色

ConstantBuffer(这个Buffer会直接绑定到Shader,Shader里面可以声明这个CBuffer 直接用)

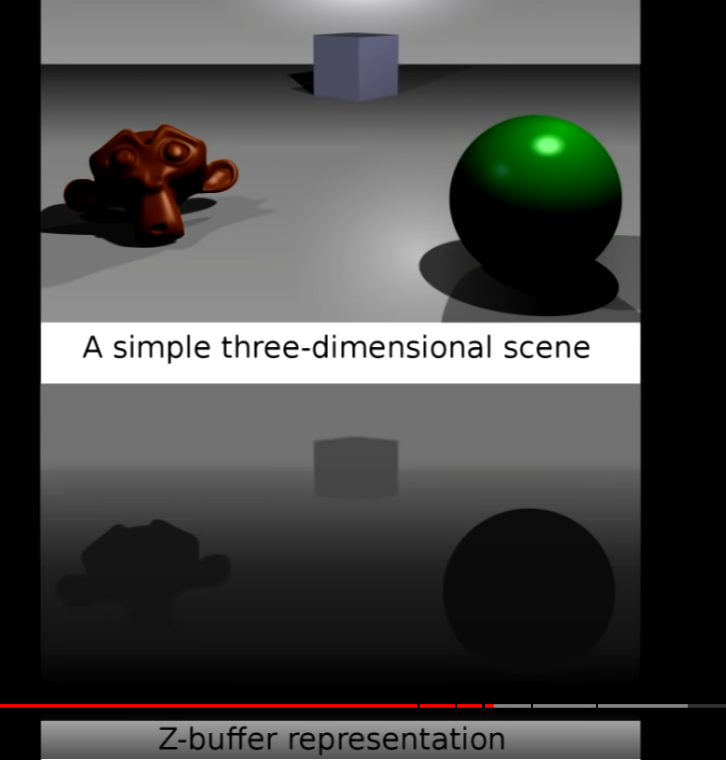
3.Draw Order

如果Draw两个Cube 后面的会一直在前一个 前面，即使他的Z 更大

这里需要ZBuffer

D3D11初始化创建Device的时候，就自动创建了frameBuffer,frameBuffer实际上就是ColorBuffer里面有pixel颜色

ZBuffer 需要主动创建，本质上也就是一个图片 存的是Vertices的深度信息



4.创建ZBuffer





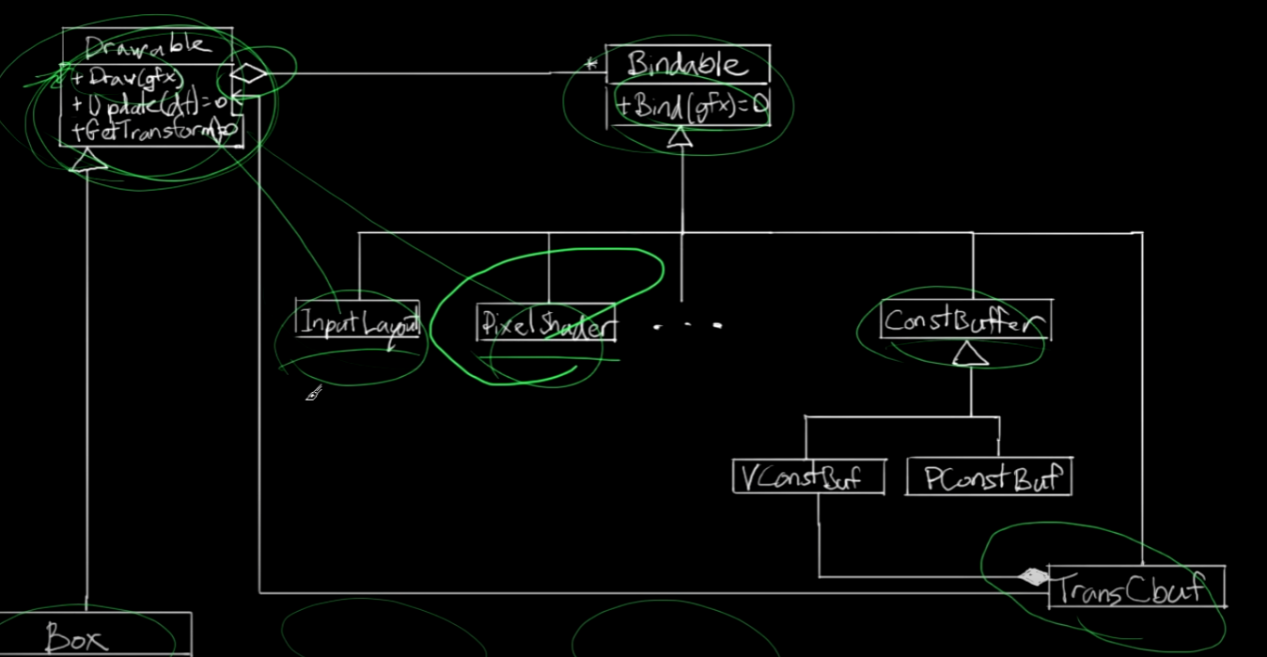
还需要给StencilBuffer创建Texture 大小Match SwapChain,创建DepthStencilView(跟给SwapChian创建RenderTargetView一样)

最后绑定 第二个参数就是RenderTarget



# [Bindable / Drawable System Part 1]22

1.UML



# [Texture Mapping]25

1. gdiplus.h (load image)

用gdiPlus这个开源 去加载图片

使用的时候需要注册 退出的时候需要销毁 创建了一个gdiplusManager的类 使用RefCount保证只会初始化和销毁一次

1. 创建Texture类 继承自Bindable

(1)所有Bindable的类 构造参数都要带Graphics& gfx,构造函数中 创建出需要保存的COM Object, 以Texture为例 就是创建Texture2d 再根据 Texture2d 生成TextureView

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> pTextureView;

(2)继承Bind函数

主要就是绑定

void Bind(Graphics& gfx)noexcept override;

GetContext(gfx)->PSSetShaderResources(0u, 1u, pTextureView.GetAddressOf());

1. Sampler Texture的绑定还需要Sampler的绑定

基本Samper就是对Texture怎么表现的描述

D3D11\_SAMPLER\_DESC samplerDesc = {};

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

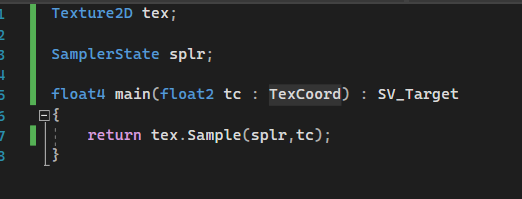
samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

GetContext(gfx)->PSSetSamplers(0, 1, pSampler.GetAddressOf());

1. Shader

PixelShader里面声明 Texture2d ,TextureSampler,TexCoord由VertexShader传入



5.Drawable





构造函数中创建需要的Bindable