3. 삼각형 띄우기

간단하게 삼각형 하나를 화면에 띄우는게 최종 목표

지난 시간 내용 복습할 겸 정리

Engine.h보면 모든 기능들을 클래스 별로 분할을 해서 하나씩 들고 있게 만들어 놨어.

책에는 한 클래스에 모든 내용이 같이 포함되어 있어. 어떤식으로 공부 하냐는 개인의 기호이지만 클래스로 분할하는 연습을 하면 코드의 구조를 연습해보면서 뭐가 뭘 담당하고 역할을 맡는지를 한번 더 고민해보고 생각하게 되기에 공부할 때 유용하다. Engine.h를 보면 핵심 부품이 4가지가 있다는 것을 알 수 있고, 초기화를 할 때 각각에 어떤 부품을 건내줘야 하는지도 확연히 보이는 장점이 있어.

요약을 해보면

Device라는 애는 말그대로 GPU에 접근을 하거나 리소스를 요청을 하거나 할 때 사용하는 인력 사무소 대표라고 했었어. GPU에게 뭘 보내고 싶다고 하면 Device를 통해서 하는 거니까 얘가 가장 핵심이라고 할 수 있어. 다른 애들을 만들 때도 인자로 \_device를 넘기는 것을 알 수 있어.

.

CommandQueue 같은 경우에는 GPU에게 뭔가 요청을 보낼 때 걔를 바로 한번한번 요청할게 생길 때마다 보내는 게 아니라 뭉쳐가지고 한방에 보내기 위한 수단이야. CommandPattern에서 사용하는 그 Queue를 CommandQueue라고 부르고 꼭 DX12에서는 사용해야 하는 걸 알 수 있었고 하나하나씩 저장해서 한번에 뿅 날려주는 용도로 활용을 했어.

SwapChain은 doulbe buffering과 관련된 부분 이었어. 버퍼를 두개 들고 하나는 GPU가 그리는 용도, 하나는 현재 화면에서 출력하는 용도로 사용하다가 다음 frame에서는 역할을 뒤바꿔서 이전 프레임에서 GPU가 열심히 그려준 것을 화면에 노출하고 현재 그리던 것을 다시 GPU가 버퍼 용도로, 그리는 용도로 왔다갔다 하면서 활용을 했었어. 그래야지만 우리 화면에 실시간으로 그려지는 중간 단계가 보이는게 아니라 항상 완성된 결과물만 보이게 되어가지고 깜빡깜빡 거리는 현상이 없어지니까 이렇게 버퍼 2개를 활용한다.

DescriptorHeap은 좀 애매한데 얘같은 경우가 기안서라고 표현을 했지만, 사실은 어떤 리소스를 서술하는 그런 아이. 말 그대로 Descriptor라는 용어가 서술하다라는 의미가 있어. DescriptorHeap클래스에서 사용하는 \_rtv즉RenderTargetView라는 아이 원본 리소스 자체는 SwapChain.h에 ComPtr<ID3D12Resource> \_renderTargets[SWAP\_CHAIN\_BUFFER\_COUNT]; 이렇게 들어가 있어. 얘가 진짜 렌더타겟 버퍼. 이름들도 주의깊게 봐야해. ID3D12Resource라는게 지금 우리가 버퍼로 활용하려는 그 리소스를 말을 하는구나 라는 걸 알 수 있는 거고. 이 resource 자체를 뭔가 요청할 때 이 resource 자체로 보내는게 아니라 그 아이를 서술하는 DescriptorHeap이라는 애를 만들어서 여기있는 애를 이용해서 서술해서 GPU한테 요청할 때도 View를 이용해서 요청한다고 했었어. 말하자면 우리가 어떤 아파트 거래를 한다 했을 때 그 아파트를 실제로 트럭에 실어가지고 갖고 가서 거래를 하는게 아니라 아파트 땅 문서같은걸 들고 그걸 가지고 거래를 하는거 그 아파트 문서에는 아파트 주소와 아파트 이름과 소유자 명의와 온갖 설명이 들어가 있을거야. 그런 느낌으로 실제 리소스 자체를 설명하는 역할을 하는게 이 View라는 걸 알 수 있는 거고, DX12에서는 View라는 거 자체를 한번에 만드는게 아니라 DescriptorHeap이라는 애를 이용해서 만들었다고 살펴 보았어.

DescriptorHeap이라는 애가 DH[ ] view의 연속된 공간이고, 같은 용도로 활용하는 RenderTarget으로 사용할 애들은 [ ] [ ] 이렇게 모아가지고 한번에 배열 형태로 만들었었다고 했었어.

DH[ [ ] [ ] ] 지난번에는 이렇게 2개를 만들어 놨어. 각자는 실질적으로 어떤 원본 리소스를 가리키고 있는 상황이야. [ ]얘는 그 리소스가 무엇무엇 인지를 설명하는 그런 역할이라고 했어. [ ]이게 지금 이 코드에서 말하는 \_rtvHandle[SWAP\_CHAIN\_BUFFER\_COUNT]이라는 거고 혹시 다른 용어가 마음에 든다면 예전엔 View라고 했었어. 결국 요약을 하면 DescriptorHeap안에는 여러가지 View가 있다고 볼 수 있는 거고 각자의 View는 어떤 리소스를 설명하는 용도로 활용이 되고 있다고 일단 결론을 얻을 수 있어.

이렇게 코드를 클래스별로 분할하며 연습을 하다보면 이제 코드가 서로 종속성이 있고 같이 사용하는 애들이 눈에 띈다. 예를 들어 CommandQueue에서 RenderBegin, RenderEnd를 이렇게 해줬는데 가만히 놓고 보면 \_swapChain의 원본 리소스 backBuffer를 넣어 놨으니까 얘를 꺼내가지고 \_swapChain→GetCurrentBackBufferResource().Get() 이렇게 활용하는 것도 볼 수 있고, 실질적으로 그 아이와 관련된 View 자체도 \_descHeap→GetBackBufferView(); 이런식으로 꺼내서 View도 사용할 일이 생기는 것을 볼 수가 있어. 우리는 원본 Buffer와 걔를 설명하는 View를 따로따로 관리하고 있기 때문에 \_descHeap이랑 \_swapChain을 각각 이렇게 따로따로 CommandQueue에 저장해서 사용을 해야 됐었어. 이런걸 보면 경우에 따라 DescriptorHeap을 따로 외부에 빼기 보다는 정말로 필요한 애들의 View를 그냥 같이 원본 리소스랑 같이 관리하는게 어떨까 하는 개선사항이 들기도 함. 그런 개선사항이 생길 때는 그냥 고치는 것도 괜찮은 방법이야. DescriptorHeap에서 관리하던 \_rtvHeap, \_rtvHeapSize, \_rtvHandle 이 3총사를 그냥 SwapChain안에다가 이전을 시키는게 좋을 거 같다는 생각이 들었어. 이 작업을 하고 삼각형을 그려 보기로 하자.

SwapChain에다가 DescriptorHeap의 3총사를 복붙해서 옮겨 놓는다.

\_rtvHeapSize는 없어도 되므로 삭제

이름 통일을 위해 \_renderTargets 이걸 \_rtvBuffer로 바꿔줌. 이름 바꿔주려면 ctrl+r+r 해서 바꿔주면 되는데 no references were found in the selected scope 오류 나면서 안바꿔지면 범위를 including external items로 바꿔주면 됨. 적용하면 싹 다 바뀌어 있는 거 볼 수 있어.

다시 DescriptorHeap으로 가서 Init 함수 부분을 통으로 복사해서 SwapChain에 옮겨 준다. 이름을 CreateRTV로 바꿔줌. SwapChain클래스 안이라 SwapChain인자는 없어도 된다. header에도 선언부 private으로 만들어준다.

내부에 \_rtvHeapSize안들고 있기로 했으니 int32로 스택 메모리에 잠시 활용하고 날려주기로

이것저것 수정

필요한 기능들이 뭉쳐있으니까 관리하기가 편해졌다는 생각이 듦.

CreateRTV를 만들어줬으면 SwapChain::Init()에서 호출해주면 됨. device를 받아줘야 하는데 Init에서 안받고 있으니 ComPtr<ID3D12Device>를 Init의 두번째 인자 부분에 복붙해줘.

SwapChain을 초기화 할 때는 SwapChain을 먼저 만들어주고, 버퍼를 꺼내오고, RTV를 만들어주는 일련의 단계들로 이루어져있으니까 이것도 다른 함수로 빼주도록 할거야. 이렇게까지 해줄 필요는 없지만 뭐가 뭘 하는지 확실히 알게 하기 위해서 분리를 해볼거야.

Init을 복붙해서 이름만 CreateSwapChain으로 바꿔준다. 매개변수로 device 받아줄 필요 없으니까 삭제. 구현부도 만들어준다. Init구현부의 내용을 CreateRTV빼고 싸그리 복붙해준다. 결국 Init에서 CreateRTV하기 전에 CraeteSwapChain을 해가지고 info, dxgi, cmdQueue를 전달을 해서 만들어주면 된다. 이렇게 해서 SwapChain코드를 정리해 봤어. 여기까지 했으면 DescriptorHeap은 필요가 없으니까 삭제를 해주도록 한다. 고침으로서 발생한 빌드를 일일이 찾기 보다는 다시 빌드를 해서 컴파일러한테 찾아달라고 떠 넘겨본다. ctrl+shift+s로 저장하면서 반복해준다.

CommandQueue에서도 DescriptorHeap을 사용하고 있었으니까 얘랑 관련된 애들을 삭제해줘야함.

Engine에서 DescriptorHeap도 날려줘. 여기저기 문제 되는거 수정해줌.

CommandQueue의 \_descHeap→GetBackBufferView() 이 부분만 에러가 남았는데 SwapChain에서 그 부분을 복원시켜줘야해.

\_rtv 3총사와 관련된 부분 수정해줄거야. D3D12\_CPU\_DESCRIPTOR\_HANDLE GetBackRTV() { return \_rtvHandle[\_backBufferIndex]; backBuffer의 RTV를 받아주겠다는 새 함수 만들어준다. GetCurrentBackBufferIndex 이 함수는 날려줌.

GetCurrentBackBufferResource는 이름이 너무 기니 ctrl+r+r로 이름을 GetBackRTVBuffer로 바꿔줌.

리빌드 하고 에러나는 부분 D3D12\_CPU\_DESCRIPTOR\_HANDLE backBufferView = \_swapChain→GetBackRTV(); 이렇게 수정

정상적으로 다 수정되었다면 Client 실행하면 하늘색 창 뜬다.

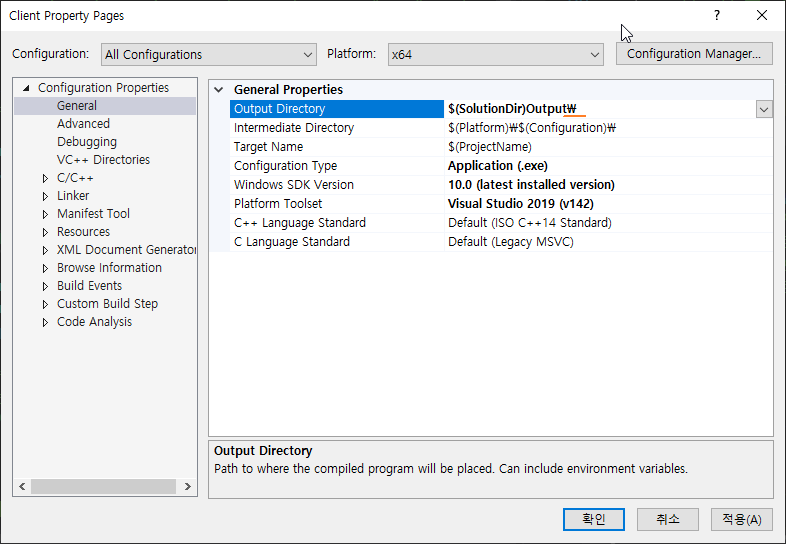
이렇게 거의 첫번째 코드 정리를 해 보았어. 여기서 알 수 있는게 여러가지 있어. 코드 정리 하면서 필요한 기능들이 살짝 달라지다보니까 Init 함수에서 device 인자를 새로 추가를 하고 받아오는 코드도 수정을 하고 이런 부분이 들어갔어. 엄밀히 반대로 생각해보면 사실은 Engine이라는 코드 자체를 전역으로 만들어서 활용을 하고 있다보니까 사실은 굳이 매번마다 받아줄 필요 없이 엔진을 통해서 디바이스를 꺼내오거나, 엔진 전체 객체를 이용해서 CommandQueue를 사용한다거나 하는 것도 가능하다. 그래서 이게 코드의 스타일이라고 말 할 수 있어. 어떤 사람들은 개체지향적인 설계를 중요하게 생각해서 필요할 때 마다 받아와서 저장을 해서 활용하는게 좋다고 생각하는 사람이 있을 수 있고, 코드의 미학을 중시 여기는 타입. 하지만 이런 식으로 코드 수정이 생겼을 때 귀찮아지는 경우가 생길 수 있어. 선생님 같은 경우는 코드의 깔끔함 보다는 무조건 빠른 작업속도. 빨리 작업하는게 장땡이다 하는 타입이기 때문에 특별히 Singleton이거나 전역에서 꺼내 사용하는 것에 대한 거부감이 없는 편. 그래서 앞으로도 코드에서 엔진에 있는 코드에서 꺼내쓴다고 할 경우에는 GEngine을 이용해서 그냥 적극적으로 안에서 일단 꺼내쓰는 형태로 만들도록 할거야. 예를 들어 SwapChain어딘가의 함수에서 매개변수로 device를 받아주거나 할 필요 없이 그냥 필요한 쪽에서 GEngine→GetDevice() 이런식으로 꺼내서 쓰면 된다는 얘기가 된다. 근데 모든 코드를 전역으로 꺼내서 쓰면 코드의 종속성이 쎄지기 때문에 관리하기 어려워질 수 있지만 지금 단계에서는 별써부터 걱정할 필요는 없어.

Engine에 가서 \_dvice, \_cmdQueue, \_swapChain 얘네들을 꺼내 쓸 준비를 넣어보도록 하자.

Engine의 public:에 3총사를 복붙한 다음에 Get함수들 각각 만들어 준다.

CommandQueue의 \_cmdList도 자주 쓰므로CommandQueue클래스에 Get함수를 만들어준다.

Output디렉터리가 슬래시로 끝나지 않았습니다. warning이 뜨는데

속성에 들어가서 \를 하나 붙이면 됨.

EnginePch.h에 돌아와서 자주 활용할 몇가지를 define으로 빼놓도록 할거야.

#define DEVICE GEngine->GetDevice()->GetDevice();

#define CMD\_LIST GEngine->GetCommandQueue->getCmdList()

SwapChain.cpp에서

DEVICE->CreateDescriptorHeap(&rtvDesc, IID\_PPV\_ARGS(&\_rtvHeap));

device를 매개변수로 받아주는게 아니라 이런 식으로 막바로 쓸 수 있다는 얘기

근데 불완전한 클래스 형식에 대한 포인터는 사용할 수 없습니다. 라고 나오는데 왜 그러나면 Device기능을 활용하려면 #include “Engine.h”를 붙여야 해. Engine에다가 Device를 꺼내서 활용하고 있기 때문에 #include “Device.h”까지 붙여줘야해.

Engine을 들여다 놓으면 Device, CommandQueue, SwapChain 3총사 다 활용할 확률 높은데 필요한 거 한땀 한땀 추가하는 거 피곤할 수 있어.

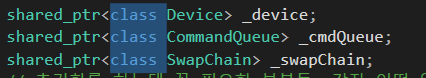
Engine.cpp에서 활용하던 include들을 Engine.h에 옮겨 놓았어.

#include "Device.h"

#include "CommandQueue.h"

#include "SwapChain.h"

Engine.h만 include하면 자동적으로 3총사를 활용한다고 가정을 하고 작업을 해보겠어. 물론 이 3개가 자주자주 바뀌고 내용이 방대하면 헤더에 헤더를 넣는 건 안좋은 습관이지만 지금 같은 경우는 Engine에 다 넣어도 괜찮은 코드들을 관리 차원에서 분리해서 사용하고 있는 것이기 때문에 헤더 자체를 .h에 넣어주도록 했어. 이러면 전방선언 부분들은 더이상 필요 없게 됨. 내가 어떤 아이인지 알기 때문에



여기까지 지난시간에 했던 내용을 고쳐보았어.

이제 삼각형 띄우기 작업을 해보자.

새로운 식구들 몇몇 추가가 되어야 해. Dx 수업이 애매한게 모든 기능들이 한번에 맞물려서 동작하는 그런 기능이다 보니까 하나를 설명하기 위해서는 다음 시간에 설명해야 할 내용까지 써야 하는 상황이 생김. 그러다 보니 처음부터 깊이 이해하려고 하지 말고 거시적인 관점 숲을 보면 공부를 해나가야함.

RootSignature클래스를 만들어준다. 이 클래스에 대한 자세한 내용은 다음 시간에.

// [계약서 / 결재]

// CPU [ ] GPU [ ]

RootSignature를 이용해 설명을 해가지고 내가 어떠어떠한 레지스터, 버퍼를 활용을 하겠다 명시를 해주는 것. 말 그대로 어떤 정책을 사용할 것인지, 어떤 땅을 임대할 것인지 계약서를 쓰는 것.

ComPtr<ID3D12RootSignature> \_signature;

초기화 부분들을 한땀한땀 타이핑 할 이유가 없으니 복붙하자. 자세한건 다음 시간에.

Engine으로 돌아가서 새로운 식구 만들어진거 추가

shared\_ptr<RootSignature> \_rootSignature;

헤더도 추가해 준다.

Get함수도 만들어준다.

Init에서도 초기화 해준다.

이제 셰이더랑 메쉬를 준비해야 함. 삼각형을 그리기 위해서는 각 정점들. 삼각형 모양을 그려준다음에 그것을 GPU한테 넘겨줘야 하기 때문에 Mesh가 필요해. Mesh 까진 아니더라도 어떤 정점을 만들어서 넘겨줄 수 있는 그런 형태가 필요해. 나중에 가면 좀 복잡한 형태의 구조도 넘겨줄 것이기 때문에 미리 Mesh라는 클래스를 만들어주자. 지금은 단순한 삼각형을 넘겨주는 용도로만 활용하게 될거야.

Shader 클래스도 만들어 준다.

Mesh부터 작업

// [유니티짱]과 같이 뭔가 정점으로 이루어진 물체, 정점의 모임

리소스와 뷰를 들고 있어야 해.

Init함수, Render함수를만들어주자

Vertex 아직 안만들었는데, Mesh라는 건 일련의 정점들로 이루어지게 될거야. 정점에는 과연 어떤 정보가 있어야 할지를 정의를 해줘야 해.

EnginePch.h에 들어가서 Vertex 라는 이름의 struct 만들어준다.

GPU쪽 메모리에 복사를 일단 해줘야 함.

SwapChain에서도 봤지만 Resource를 만들어 준 다음에 View를 만들어 줘서 View로 나머지 코드에서 활용하는 거랑 유사하다고 볼 수 있어.

버퍼를 만들어준 다음에 그 버퍼를 설명하는 VIEW를 만들어준 다음에 실질적으로 활용을 할 때는 이View를 이용해서 뭔가를 건내준다는 것을 알 수 있어.

Mesh 만들어 주고 Shader까지 만들어줘야지 최종적으로 테스트를 해볼 수 있다.

ROOT\_SIGNATURE도 define으로 만들어놓음

shader는 외주 기업들이 뭘 해야할지 기술하는 명령서라고 했는데 shader단계에도 여러 단계가 있었어. 렌더링 파이프 라인에 여러 단계가 있었는데 시작을 할 때 input assembler다음에 vertex shader라는 게 있었고 dx10, 11에서 추가가 되었던 온갖 기능 있었고, 다음에 rasterizer stage를 지나 pixel shader가 있었어. 각 단계별로 어떤 명령을 할지 따로따로 기술할 수가 있는데 기본적으로 가장 중요하게 활용할거는 vertex shader와 pixel shader 이 두가지를 기본으로 활용한다고 보면 됨. Shader::Init 초기화 하는 함수를 보면 어떤 경로에다가 file을 읽어서 vertex shader와 pixel shader를 각각 만들어주고 있어. 근데 이것만 보면 뭘 하고 있는지 알기 힘드니까 **shader 파일을 만들어 보자** .

Engine 프로젝트에서 우클릭 후 추가->새 항목 가서 보면 HLSL이라고 셰이더 파일도 만들 수가 있어. 이름은 default로 하고 경로는 Resource 산하에 Shader라는 폴더를 만들어서 넣는다. Shader라는 filter를 만들어서 거기에 넣어준다.

여기에 shader를 작성해주는데 C++ 문법이 아니라 hlsli라는 **셰이더 언어**를 이용해야 한다.

C, C++ 언어와 유사하다는 걸 알 수 있어.

셰이더 라는게 결국 **일감 기술서**라고 했었어. 단계별로 뭘 해야할지를 기술할 수 있는데 모든 단계를 다 기술할 필요는 없지만 **vertex shader 단계에서 뭘 해야할지** **pixel 셰이더 단계에서 뭘 해야할지** 이 단계별로 일감을 던져줬다고 보면 됨.

일단 딱히 아무것도 안하고 받은대로 전달하는 shader 파일을 만들었어. **셰이더 파일**로 만들어줬으면 **로드해서 어떤 식으로든 건내줘야 하는데 그 부분이 Shader::Init**

**D3DCompileFromFile**셰이더 파일 자체를 읽어서 파싱(컴파일)을 해서 통과를 하면, Shader.h에 있는 **\_vsBlob, \_psBlob쪽에다 셰이더 파일과 연관된 리소스**가 만들어 진다.

한번 조립해서 활용하면 무슨 일을 하고 있는지 감이 올거야.

최종적으로 만들걸 빌드해주고 조립을 해볼거야.

일련의 부분들을 어디선가 호출을 해줘야 해. Game쪽에서 넣어보도록 할거야.

**shared\_ptr<Mesh> mesh = make\_shared<Mesh>();**

**shared\_ptr<Shader> shader = make\_shared<Shader>();**

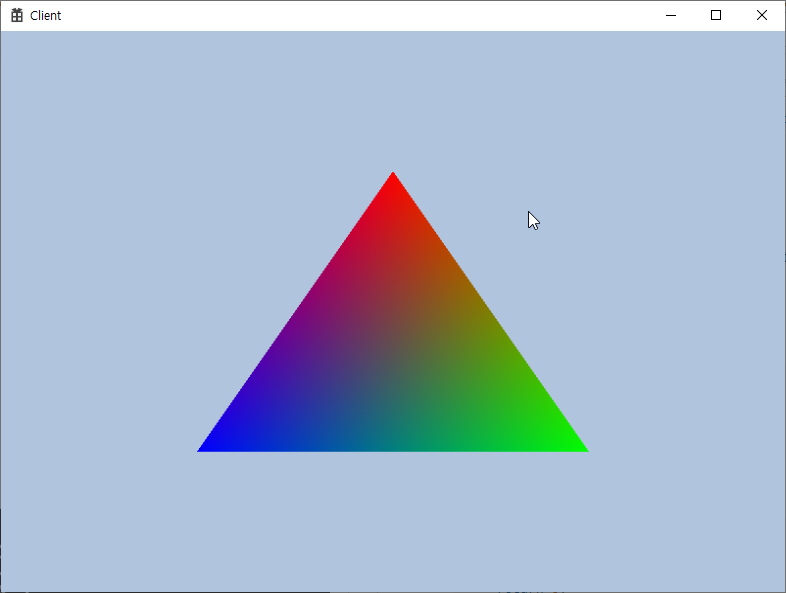
추가해 주고

Engine.h 에 Mesh, Shader 헤더 추가해줌.

Engine다시 빌드 하고

Game::Init에 삼각형을 하나 만들어 볼거야.

엄밀히 말하면 Mesh가 Render()안의 RenderBegin(), RenderEnd()사이에 들어가야 하는 건데 테스트 위해서 Game::Update()에서 RenderBegin(), RenderEnd()사이에 넣어 호출해 준다.

점 3개에만 색을 지정해 줬는데 중간에 색이 섞여 채워짐 알 수 있어. rasterizer state에서 채워준거.

pixel shader에서는 딱히 색깔을 변화하지 않았다는 거 알 수 있는거고, vs, ps에서 넣어준 그 상태로 출력된 거 알 수 있어.

좌표는 뭘 의미하냐. 중점이 0, 0이라고 보면 됨. 왼쪽 끝이 -1, 오른쪽 끝이 +1, 위로 가면 +1, 아래가 -1. Vec3의 세번째 값은 사실상 무시되고 있음. 다중에 가면 이걸 이용해서 딴 연산 할 수 있지만 지금은 무시.

세가지의 점에 대한 정보만 넣어줬을 뿐인데 어찌됐건 렌더링 파이프라인 단계를 거치게 되면서 기보적으로 사이에 있는 정점들까지 채워줬다는 걸 알 수 있어.

나중에 가면 defaul.hlsli에 어마어마한 작업들을 시키게 된다. 그럴려면 지금처럼 단순한게 RootSignature를 비워두고 만드는게 아니라 이제 훨씬 더 내용이 복잡하게 됨. 결국 DirectX가 외주를 맡기기 위한 피나는 노력이라고 볼 수 있어.

지금 단계에는 전체 코드 이해가 잘 안되겠지만 사실 초기화하는 부분은 중요하지 않아. 여기서 렌더링과 관련된 고급 기법을 익히는게 중요한거지 이런 세세한 함수들이 뭘 하는지는 당장엔 알 필요가 없어. 하나라도 놓치고 가면 찝찝해서 넘어가지 못하는 분들이 있는데 DirectX나 운영체제 따라 만들기 식 책들을 보면 알겠짐나 모든 기능을 알 수가 없어. 그래서 지금은 복붙하면서 만드는 흐름을 따라가고 있다는 걸로 위안을 삼으면서 딸아오면 됨.

RootSignature와 관련된 깊은 내용들과 GPU 메모리 등. GPU에도 레지스터가 있는데 그것들을 활용하는 다양한 방법들에 대해 추가로 알아보게 될거야.

삼각형 외주에서 발전시켜 가며 세련된 3D게임을 만드는 과정을 따라 가게 될 거야.