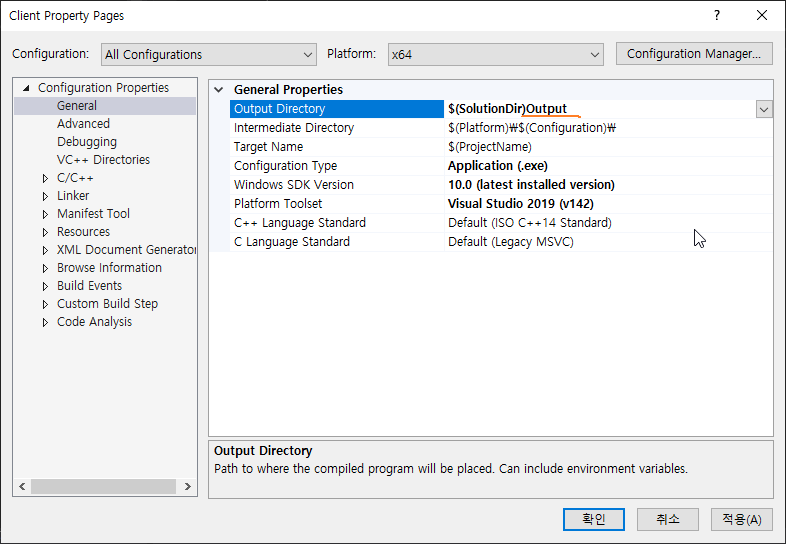
**2. 장치 초기화**

GPU한테 외주를 떠넘기기 위한 사전 작업이라고 할 수 있어. GPU랑 연결을 하고 일감을 하나씩 던지게 된다는 얘기.

지난 시간에 Client 속성->구성 속성→ 일반에서 출력 디렉토리를 안바꿔놨어. Client 빌드물이 x64 폴더에 만들어지는데 x64폴더를 삭제하고 출력 디렉토리를 $(SolutionDir)Output로 설정해줌.

빌드해보면 Output 폴더에 Client.exe 파일 생성되어 있는 거 볼 수 있어.

API 너무 많이 등장해서 멘붕 올수도 있어. 처음 보는 함수들 너무 많이 등장해서 분량에 압도될 수 있지만 외울 필요 없고 다 외우고 기억하는 사람 없어. 조금씩 자기 영역을 확대해 가며 복습을 할 때 MS 문서 보면서 기능을 하나씩 둘씩 익히는 방식으로 공부를 하면 됨. 오늘은 가볍게 코드를 따라치거나 복붙을 해가면서 하나씩 만들어주면 됨.

Engine 필터에 새로운 Engine이란 클래스를 만들어줘.

오늘 사용할 클래스인 CommandQueue, Device, SwapChain, DescriptorHeap 클래스도 만들어줘.

ctrl+shift+s로 다 깔끔하게 저장해준다.

Engine클래스 자체가 가장 핵심적인 기능들을 담게 됨. DirectX12에 관한 책들을 보면 사실 단일 클래스에 모든 내용을 다 때려박아서 구현하고 있어. 물론 이렇게 구현해도 되긴 되는데 처음 공부하는 입장에서 모든 기능들을 한 클래스에 다 때려넣어버리면 멘붕이 오는게 코드가 뒤섞여 있다보니 갈피를 못잡을 확률이 높아. 그래서 조금 귀찮을 수 있지만 하나하나 분할해서 기능을 불리해서 작업을 해줄거야. 실전에서 이렇게까지 해야 하는지는 의문이 들 수 있겠지만 일단은 지금 강의에선 이렇게 진행할거야.

Engine에 기능을 추가해줄거야.

여기서 만들어준 CommandQueue, Device, SwapChain, DescriptorHeap은 다 엔진에서 물고 있는 기능들이라고 보면 됨. 이게 각각 하나의 기능을 담당하는 클래스들이야.

초기화 해주는 Init(), 그려주는 Render() 함수 추가. ctrl + .. 으로 구현부도 cpp파일에 만들어줌.

Engine의 Init을 할 때는 Client 쪽에서 Init 함수를 호출해줄거야. 그 때 받아야 할 정보가 몇가지 있는데 출력 윈도우 핸들이랑, 윈도우의 너비, 높이, 창모드인지, 전체화면인지 등등을 받아줘야해.

EnginePch.h을 열어주고 struct WindowInfo 코드를 추가해줘.

EnginePch.cpp를 살펴보면, 우리가 엔진 클래스를 전역으로 사용하고 싶어, 언제 어디서든 꺼내쓸 수 있게 하고 싶은데. 그렇게 하려면 방법이 여러가지가 있는데 대표적으로 싱글톤이라는 디자인 패턴이 있고, 그게 아니라 간단하게 하기 위해서는 전역 클래스로 만드는 것도 방법이야. 여기서는 전역으로 unique\_ptr<Engine> GEngine;

전역이긴 전역인데 다른 애들이 이 GEngine에 대한 존재를 모르니까 활용할 수가 없어. EnginePch.h 헤더에 가서 이 아이가 등장할 거라고 선언을 해줘야 하는데 extern을 붙여서 아직은 너는 모르겠지만 unique\_ptr<Engine> GEngine이란 애가 등장할거야 라고 선포를 해줬다고 보면 됨. 이렇게 하면 EnginePch를 사용하는 모든 아이들이 GEngine에 접근을 해서 사용할 수 있게 되는거.

이렇게 하고 컴파일 하면 문제가 되는 부분은 이 <Engine>이란 부분을 모르는 거. #include Engine.h를 써도 통과를 하지만 이럴 땐 일반적으로 전방선언을 한다. extern unique\_ptr<class Engine> GEngine; 이렇게 class Engine이란 것을 한번 위에다 만들어줘야지만 Engine이라는 거는 당장에는 모르겠지만 나중에 알려줄거야 라고 시점을 미룰 수 있어. class Engine으로 따로 위에 빼줘도 되지만 <Engine>에 class 붙이면 같은 효과가 됨.

HelloEngine() 더이상 활용 안함.

Engine을 틈틈히 빌드를 해준다.

Engine.h, .cpp로 돌아와 하나씩 기능을 넣어줄거야.

Engine.h에 추가한 클래스 4개를 shared\_ptr 변수로 선언해줘.

Engine.cpp의 Init()에서 make\_shared<>()로 초기화를 해준다.

**1. Device**

**인력 사무소: GPU**와 핑퐁을 하는 역할

Device.h 내용 채워줘

ComPtr<ID3D12Debug> \_debugController; // 디버그 용도 기능

ComPtr<IDXGIFactory> \_dxgi; // 화면 관련 기능들

ComPtr<ID3D12Device> \_device; // 각종 객체 생성

Device.cpp Init()에서

::D3D12GetDebugInterface(IID\_PPV\_ARGS(&\_debugController));

\_debugController→EnableDebugLayer();

::CreateDXGIFactory(IID\_PPV\_ARGS(&\_dxgi));

::D3D12CreateDevice(nullptr, D3D\_FEATURE\_LEVEL\_11\_0, IID\_PPV\_ARGS(&\_device));

이렇게 3총사 만들어 줬어.

Ddvice를 한땀한땀 만들어줬고 나머지도 이렇게 만들어줄거야. (코드의 주석 참조)

Engine.cpp에 헤더들 include해주기

\_device→Init(); 호출해주기

**2. CommandQueue**

CommandQueue.h [외주 목록]에 일감을 차곡차곡 기록했다가 한 방에 요청하는 것

ComPtr<ID3D12CommandQueue> \_cmdQueue; // 일감을 밀어 넣다가 나중에 한방에 GPU한테 요구를 할 때 활용

ComPtr<ID3D12CommandAllocator> \_cmdAlloc; // 일감 자체를 넣어주기 위한 메모리 공간을 할당하는 역할

ComPtr<ID3D12GraphicsCommandList> \_cmdList; // 말 그대로 리스트

ComPtr<ID3D12Fence> \_fence;

Fence는 CommandQueue랑 연동을 해서 사용을 하게 됨. 지금까지 일감을 모아서 외주를 줬는데 외주를 준거를 105번이라고 하면 105번이 다 끝날 때 까지 기다리겠다 할 때 fence를 이용한다.

Init에서 cmdQueue를 new로 만드는게 아니라 COM 객체의 특정 함수를 통해 간접적으로 만들어주고 있어.

device->CreateCommandQueue(&queueDesc, IID\_PPV\_ARGS(&\_cmdQueue));

device->CreateCommandAllocator(D3D12\_COMMAND\_LIST\_TYPE\_DIRECT, IID\_PPV\_ARGS(&\_cmdAlloc));

device->CreateCommandList(0, D3D12\_COMMAND\_LIST\_TYPE\_DIRECT, \_cmdAlloc.Get(), nullptr, IID\_PPV\_ARGS(&\_cmdList));

\_cmdQueue, \_cmdAlloc, \_cmdList 3총사를 다 이런 식으로 만들어주었어.

함수이름 자체가 뭘 하고 있는지 잘 나타내주고 있어.

새부적인건 나중에 다시 볼 때 찾아보고 일단은 넘어가자.

CommandQueue와 동기화. CPU와 GPU사이에 동기화 하기 위한 펜스도 같이 만들어져 있는 거 알 수 있었어.

void WaitSync(); // 펜스를 이용해 기다려주는 함수

기본적으로 어떤 값을 줘서 해당 부분을 끝날 때 까지 기다리려주고 있는 부분

다시 Engine으로 돌아가서 CommandQueue도 \_cmdQueue→Init해줌.

**3. SwapChain**

문제는 매 프레임마다 계속 외주를 해줘야 함.

특수 종이는 이미 화면을 그리는데 사용중

다음 프레임을 계산할 때는 어떻게야 할까?

게임 화면이 섞이게 될거야. 지금 프레임과 다음 프레임의 화면이.

그렇게 하지 말고 특수 종이 2개 만들어서 하나는 그리고 용도, 하나는 외주를 맡길 때 활용 한다.

번갈아 가면서 교체하면서 쓰는게 스왑체인. 꼭 2개일 필요는 없지만 일반적으로 2개로 충분하니까 2개로 활용하는 거.

옛날 영화 보면 소총 연속해서 쏠 쑤 있는게 아니라 앞줄 쏘면 뒷줄 장전하고, 뒷줄 쏘면 앞줄 장전하는 그런 영화가 많았어. 그런 느낌. 앞줄은 전선에서 싸우면 뒤에 애들 준비하고, 앞줄 준비할 땐 뒤에 애들이 싸우는 식으로. SwapChain이 그걸 만들어주는 작업

특수 종이를 렌더 타겟이라 부를거야. 그릴 대상

// 말 그대로 버퍼를 여러개 만들어 줘가지고 왔다 갔다하는 부분이 SwapChain의 핵심 역할이라고 볼 수 있어.

다시 Engine.cpp로 돌아가서

swapChain->Init(info, \_device->GetDXGI(), \_cmdQueue->GetCmdQueue()); 해줘.

4. DescriptorHeap

view라고도 불러

GPU한테 뭔가를 요청할 때 리소스 자체를 통으로 넘기는 개념이 아니라 그 리소스를 한번 더 래핑을 해서 그 리소스는 무엇인지를 적은 기안서를 건내준다고 보면 됨.

그걸 위해서 한단계를 더 거쳐서 만들어주는 과정이라고 볼 수 있어.

// 예전에는 각기 리소스마다 각각의 뷰가 있었어. 다 따로관리하고 있었는데 DX12부터는 DescriptorHeap을 이욯해서 모든 애들을 다 묘사하고 관리하게 된다는 차이점 생김.

// 그리고 결국에는 우리가 리소스 자체를 swapChain에서 관리 했었어. \_renderTargets라는 애가 우리의 리소스라고 볼 수 있어.

// 그것을 묘사하는 기안서를 하나 더 만들고 있다고 보면 됨. 쓸데 없는 일 같긴 한데 그렇게 규격이 정해져 있음.

// 요약: SwapChain자체에 두개의 종이를 만들어 놨는데 그 종이 자체는 리소스야. 리소스를 GPU한테 바로 건내주는게 아니라

// 기안서를 만들어서 View라는 걸 건내줘야 하는데 RenderTargetView를 따로 만들어 줬으니까 그걸이용해서 BackBuffer의 View를 (RenderTargetview를) 반환해주고 있어.

// DescriptorHeap은 별거는 아니었고 GPU한테 요청을할 때 GPU가 알아먹게끔 만들어주기 위한 일종의 매핑되는 클래스라고 이해해줄 수 있어.

다시 Engine으로 돌아와서 Engine.cpp에 \_descHeap->Init(\_device->GetDevice(), \_swapChain); 추가

이렇게 하면 외주를 할 기본 준비는 거의 끝난거.

Engine에서 Render를 할 때 무엇을 해줄 것이냐?

RenderBegin(), RenderEnd()를 만들어줘.

둘다 \_cmdQueue→ 에서 작업하기에 commandQueue에 RenderBegin(), RenderEnd() 추가해준다.

// 일련의 함수가 호출되면 백버퍼 인덱스가 0이었으면 1로 바뀌게 될거고 그 다음에서 RenderBegin이 똑같은 코드가 실행이 되면 1번이 앞에 가고 0번이 뒷단에 가고 바꿔치기가 되는 코드가 실행이 되는거.

// RenderBegin과 RenderEnd를 세트로 기억을 해줘야 한다.

// commandQueue가 작업하고 있는 부분에서 SwapChain도 맞물려서 동작을 하고 있다고 보면 되는거고

// GPU에게 백버퍼를 알려줄 때 \_descHeap 즉 우리가 만들어준 View를 이용해 건내주고 있음을 알 수 있어.

// 기존에 만들었던 여러가지 기능들이 맞물려서 세트로 동작하고 있다고 볼 수 있어.

// 최종 기능을 CommandQueue에 대표로 넣어놓긴 했는데 걔를 다시 엔진에서 꺼내서 활용하고 있는 걸 알 수 있어.

이렇게 엔진 코드가 거의 완료가 되었고 테스트 해봄. 빌드하고 통과가 되었으면

클라이언트 쪽에다가 Game::Init에는 GEngine→Init(), Update에는 GEngine→Render(); 를 호출해주면 됨.

Game.cpp에 Engine.h를 추가해줘야 됨.

Init()에 windowInfo 건내줘야 함. Init(const WindowInfo& info);

Client.cpp에 가서도 전역변수 WindowInfo GWindowInfo;를 추가함. 그리고 정보를 넣어줌.

그리고 game->Init()에 GWindowInfo참조값을 넘겨줌. game->Init(GWindowInfo);

한가지 빠진건 핸들을 받아주어야 하는데 Client.cpp에 InitiInstance가 있어.

// hWnd 자체가 스택에다가 활용했다가 끝내주는데 이걸 전역 변수에 저장해주면 마무리 됨.

GWindowInfo.hwnd = hWnd;

빌드하는 습관 들이기

Client가 시작 프로그램으로 설정되어 있는지 확인한 후 f5로 실행

딱봐도 크래시 안나야 할 곳에서 크래시 나면 nullptr 크래시를 의심해야함.

장치초기화를 하는 부분 만들어놨고, 이걸 기반으로 이런저런 기능들 추가해줄거야.

RenderBegin(), RenderEnd()사이에 끼워넣어서 추가적으로 CommandList에 요청을 넣어주면 걔도 딸려서 실행됨을 알 수 있어.

창치초기화 첫 단추 끼우는게 오래 걸렸는데 운영체제나 다이렉트 x라거나 하드웨어에 접근해서 뭔가를 하는 건 대부분 초기화 하는게 오래 걸려.