Direct3D 12

개발 일지

학번: 2019182023

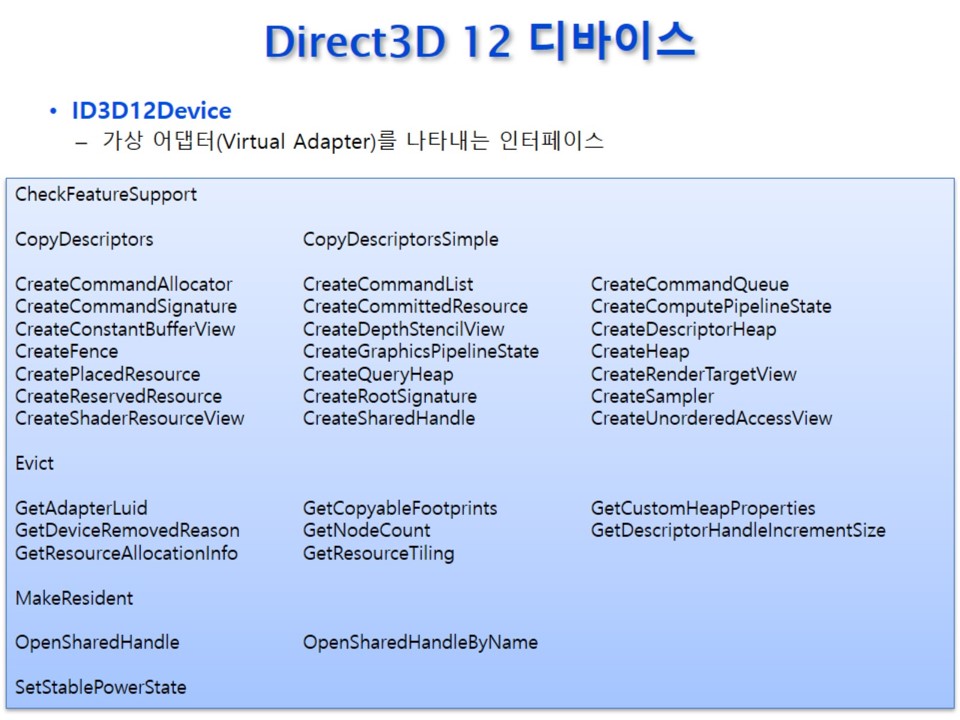
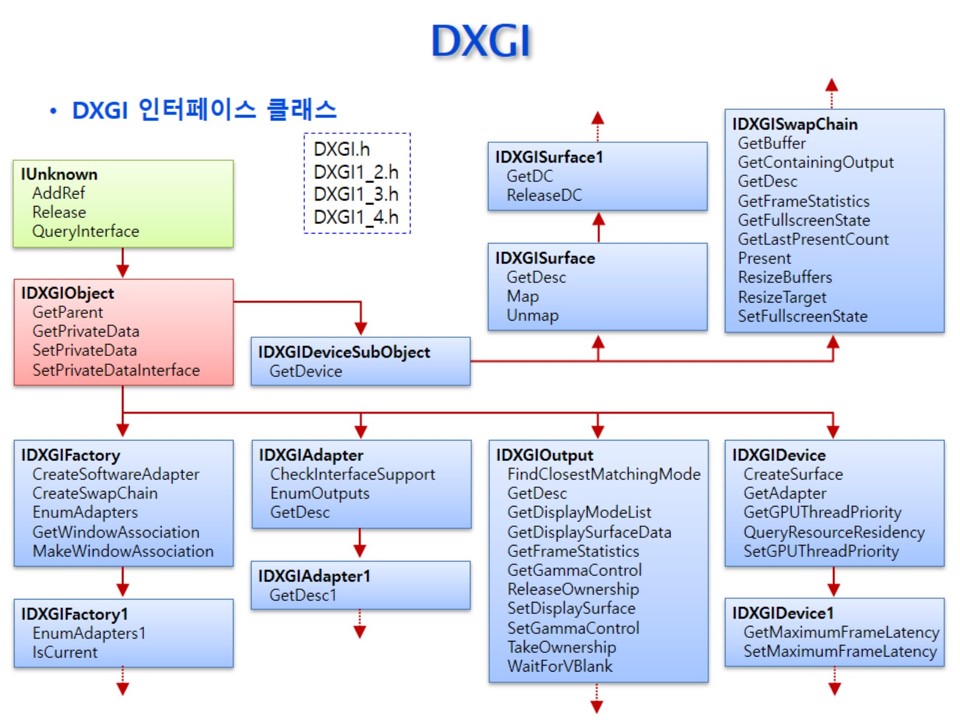
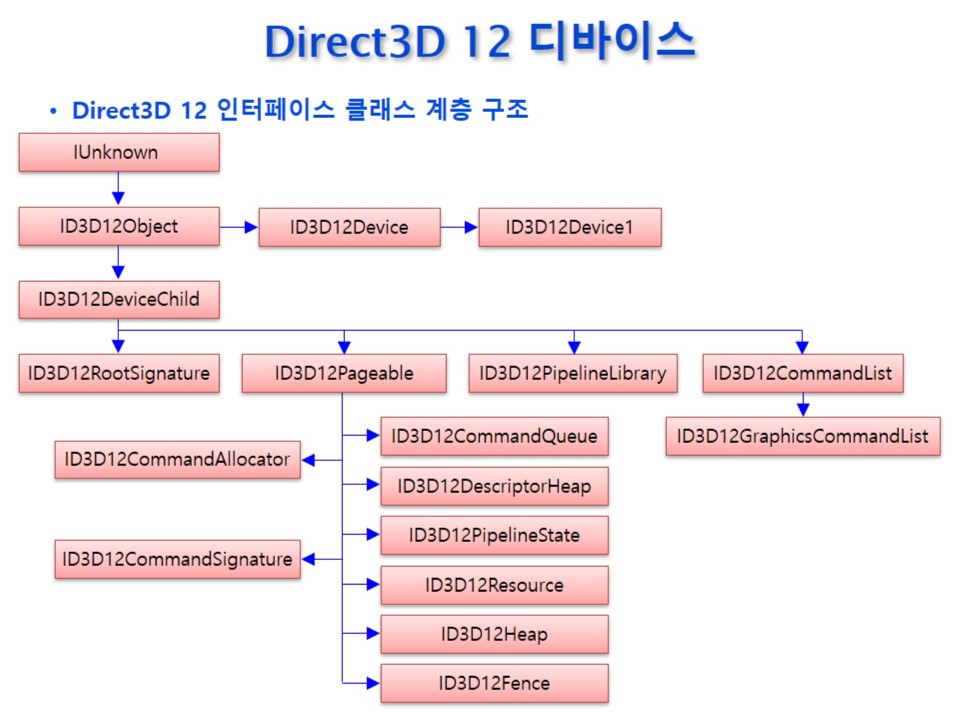
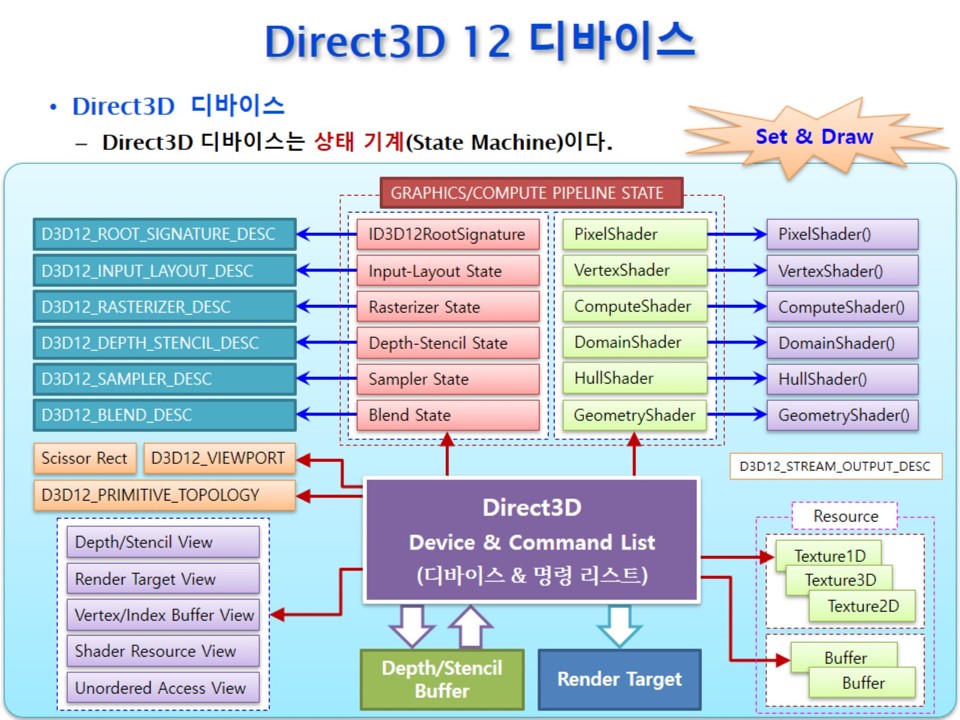
학과: 게임공학과

이름: 양정우

해당 개발 일지는 Direct3D를 배우며 알아가는 내용, 정리, 디버깅과정 교수님과의 질의 응답 등을 기록하기 위함이다.

더 나아가 교수님께서 제공하신 코드를 위주로 하였으며, 따라하기를 다시 해보며 그래픽스 파이프라인의 개념을 더 자세하게 앎을 목적으로 복습을 진행하는 것이다.

먼저 교수님께서 중요하니 외우라고 하신 사진 4장을 먼저 올리고 시작하겟다.



사전과 기능정리 및 굵직한 함수들 정리

1. 윈도우 시작부분(프로그램이 종료할때까지 영원히 돌아간다)

int APIENTRY wWinMain(\_In\_ HINSTANCE hInstance,

\_In\_opt\_ HINSTANCE hPrevInstance,

\_In\_ LPWSTR lpCmdLine,

\_In\_ int nCmdShow)

1. 윈도우창의 메뉴를 없애고 싶다면

wcex.lpszMenuName = NULL;로 바꾸면 된다.

1. 윈도우의 사이즈를 조절하고싶다면

DWORD dwStyle = WS\_OVERLAPPED | WS\_CAPTION | WS\_MINIMIZEBOX | WS\_SYSMENU | WS\_BORDER;

RECT rc = {0, 0, 640, 480}; // left, top, right, bottom

AdjustWindowRect(&rc, dwStyle, FALSE);

위 코드를 추가하고

HWND hWnd = CreateWindowW(szWindowClass, szTitle, dwStyle,

CW\_USEDEFAULT, CW\_USEDEFAULT, rc.right - rc.left, rc.bottom - rc.top, nullptr, nullptr, hInstance, nullptr);

위 코드로 변경 해준다.

1. 변수 이름에 대한 약어

m\_pdxgiFactory와 같이 코드의 내용을 더 빠르게 이해할 수 있는 약어

1. 픽셀의 형식을 설정 CreateSwapChain 내에 존재하는 DXGI\_SWAP\_CHAIN\_DESC1형식의 인자 중 format부분.
2. \_\_uuidof(ID3D12Device), &pd3dDevice 를 더 간단하게 줄이는 방법,

IID\_PPV\_ARGS(&pd3dDevice)를 이용하자.

1. ComPtr을 이용해서 자동으로 Release()를 처리한다.
2. MSAA
3. DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT
4. DXGI\_SWAP\_EFFECT\_FLIP\_DISCARD
5. DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH
6. Present, GetBuffer, GetContainingOutput
7. ResizeBuffers, ResizeTarget
8. SyncInterval, Flags
9. CreateSwapChainForHwnd
10. GetFullscreenState, SetFullscreenState
11. OnResizeBackBuffers()
12. Descriptor Heap
13. CD3DX12\_CPU\_DESCRIPTOR\_HANDLE, CD3DX12\_GPU\_DESCRIPTOR\_HANDLE
14. Heap이 필요없는 Descriptor들
15. CreateRenderTargetView(LabProject) - RTV만들기
16. CreateDepthStencilView(LabProject) - DSV만들기
17. CreateCommittedResource – 리소스 만들기
18. CreatePlacedResource와 CreateReservedResource – 리소스를 만드는 다른 함수들
19. D3D12\_RESOURCE\_STATES
20. Rasterizer Stage 설정
21. CONSERVATIVE\_RASTERIZATION
22. ReMapping
23. Viewport를 이용하여 UI를 만들 수 있다.
24. 정점 버퍼를 사용하지 않고 랜더링 하기
25. Primitive Topology
26. Root Signature
27. 게임 오브젝트와 프레임워크의 상관관계
28. HLSL - 진입점 이름
29. HLSL - 최적화 사용에 대한 여부와 디버깅 정보 사용 여부
30. HLSL - 셰이더 형식과 셰이더 모델
31. ReadCompiledShaderFile – Shader를 미리 파일로 만들어서 읽어오는 함수
32. Resource의 모든 것
33. IA(Input Assembly) 설정
34. Map과 Unmap
35. CopyResource
36. 정점 버퍼(Vertex Buffer)는 생성할 때 Format이 무조건 UNKNOWN이여야만 한다
37. LabProject9와 LabProject10을 비교해본다면 오브젝트를 랜더링 하는 방법의 차이를 확인할 수 있다.
38. 순환 include 오류
39. Depth는 Size가 존재하지 않는다, 대신 깊이 범위설정만 존재한다.
40. ㄴ
41. ㄴ
42. ㄴ
43. ㄴ
44. ㄴ
45. ㄴ
46. ㄴ
47. ㄴ
48. ㄴ
49. ㄴ
50. ㄴ
51. ㄴ
52. ㄴ
53. ㄴ
54. ㄴ

3D게임프로그래밍 2차

1. Block Compression(블록 압축)
2. Mip Map - 밉맵
3. D3D12CalcSubresource
4. Texture Array – 텍스쳐 배열
5. ㄴ
6. ㄴ
7. ㄴ
8. ㄴ
9. ㄴ
10. ㄴ
11. ㄴ
12. ㄴ

따라하기를 진행하다가 생긴 궁금증에 대한 교수님의 답변

1. 텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

   자동 생성된 설명

위 사진처럼 GameFramework.h에선 SwapChain에 대한 인터페이스에 대한 버전을 3으로 작성하셨는데, 왜 .cpp에선 버전을 1로 낮추었는지?

1. 저 함수랑 내가만든 함수는 다른함수이다. dxgi.h 내에서 제공하는 함수이다.
2. 상태 객체를 “설정한다” 와 상태 객체를 “랜더링한다”의 차이

따라하기01 – 기본 분석

먼저 따라하기부터 시작하겟다.

첫 따라하기의 주요 내용은 윈도우 창이 만들어지고 띄워지는 코드를 분석하였다.

대략적인 메인 코드의 흐름은 메인함수 시작 -> 프로그램의 시작 설정을 읽어옴 -> 무한으로 루프를 진행하며 윈도우 창을 출력한다.

따라하기02 – 윈도우 사이즈 조절

이번 따라하기는 윈도우의 사이즈를 조절하는 방법이다.

InitInstance함수 내에 아래 코드를 추가하면 된다.

DWORD dwStyle = WS\_OVERLAPPED | WS\_CAPTION | WS\_MINIMIZEBOX | WS\_SYSMENU | WS\_BORDER;

RECT rc = { 0, 0, 640, 480 };

AdjustWindowRect(&rc, dwStyle, FALSE);

1. **DWORD dwStyle = WS\_OVERLAPPED | WS\_CAPTION | WS\_MINIMIZEBOX | WS\_SYSMENU | WS\_BORDER;**
   * 이 줄은 **DWORD** 형식의 **dwStyle** 변수를 정의합니다. **DWORD**는 32비트 부호 없는 정수를 나타냅니다.
   * **dwStyle** 값을 비트 OR (**|**) 연산자를 사용하여 다양한 창 스타일의 조합으로 설정합니다.
     + **WS\_OVERLAPPED**: 이 스타일은 타이틀 바와 테두리가 있는 겹치는 창을 생성합니다.
     + **WS\_CAPTION**: 이 스타일은 타이틀 바를 포함하며 일반적으로 창의 제목과 제어 버튼을 포함합니다.
     + **WS\_MINIMIZEBOX**: 이 스타일은 창을 최소화하는 버튼을 포함합니다.
     + **WS\_SYSMENU**: 이 스타일은 창 타이틀 바에 시스템 메뉴를 포함합니다.
     + **WS\_BORDER**: 이 스타일은 창 주위에 얇은 테두리를 추가합니다.
2. **RECT rc = { 0, 0, 640, 480 };**
   * 이 줄은 **RECT** 구조체를 정의하고 초기화합니다. **RECT** 구조체는 창의 위치 및 크기를 나타내는 데 사용됩니다. 이 경우, 창의 왼쪽 위 모퉁이는 (0, 0)이고 폭과 높이가 각각 640과 480으로 설정됩니다.
3. **AdjustWindowRect(&rc, dwStyle, FALSE);**
   * **AdjustWindowRect** 함수를 호출하여 창의 크기 및 위치를 조정합니다. 이 함수는 **rc** 구조체를 창 스타일 및 여분의 스타일 정보를 기반으로 조정하여 창의 실제 크기를 계산합니다. 이때 **FALSE**는 창이 크기 변경을 허용하지 않음을 나타냅니다.

라는 기능을 수행한다.

저거만 추가하면 변하는게 없으니

hWnd 부분의 변수를 알맞게 바꾸어주자.

WS\_OVERLAPPEDWINDOW변수는 방금 만든 dwStyle로 바꾸어 주고.

rc.right - rc.left, rc.bottom - rc.top

여기서 아까 rc변수는 left top, right, bottom 순으로 변수가 입력된다.

따라하기03(LabProject02) – 게임 프로그램의 골격 작성

해당 챕터에서는 본격적으로 랜더링 골격을 짜는 작업을 거칠것이다.

일단 Direct3D 디바이스의 요소인 그래픽 디바이스, 스왑체인, 렌더타겟을 생성할 것이고

다음으로 해당 요소들의 소멸자를 만들것이다

키보드와 마우스 클릭에 대한 입력처리도 진행할것이며

게임에서 사옹되는 객체(Game Object)를 생성 및 소멸을 진행할 것이다.

애니메이션 처리

해당 그래픽 랜더링관리에 사용될 상태정보도 설정할것이며

그 장면을 그려 화면에 출력하는 것도 진행할것이다

그러기위해선

Direct3D라이브러리 헤더를 포함해서

그걸로 객체변수를 선언해주고

위에서 언급한 함수들을 선언해주고

그 함수들을 포함시킨 새로운 메시지 루프를 변경해준다.

또한 윈도우 크기도 변경해주고

그림을 그리기, 즉 렌더 타겟 지우기와 그리기를 추가할 것이고

최종적으로 완성된 디바이스와 시왑체인을 생성하는 함수를 추가할것이다.

그럼 LabProject03을 만들고 시작해보자.

// DXGI Factory 인터페이스에 대한 포인터

IDXGIFactory4\* m\_pdxgiFactory;

// Swap Chain 인터페이스에 대한 포인터, 주로 Display를 제어하는데 사용

IDXGISwapChain3\* m\_pdxgiSwapChain;

// Direct3D Device 인터페이스에 대한 포인터, 주로 Resource를 생성하는데 사용

ID3D12Device\* m\_pd3dDevice;

여기서 위와 같은 코드가 나왔는데

문득 위에 선언한 형식에 써진 숫자가 궁금했다.( IDXGIFactory4, IDXGISwapChain3)

찾아보니 해당 숫자는 버전이라고 한다.

그래서 가장 높은 숫자를 사용하는것이고

추가적으로 m\_pdxgiFactoy라고 변수 이름을 선언한 이유도 궁금하였다.

해당 내용의 의미를 찾아보니 m\_는 멤버변수라는 접두어(My와 같은 느낌)이고 p라는 것은 포인터 형식이라는 내용을 내포하고 있다고 한다.

좋은 걸 알아갔다. 이걸 이용해서 코드 분석을 할 때 도움을 더 받을 수 있을 것 같다.

추가적인 p와 같은 약어는 아래 추가하겟다

1. **p**: "포인터"를 나타내는 관행적인 접두어입니다. 변수 이름에서 **p**가 사용되면 해당 변수가 포인터 변수임을 나타냅니다. 예를 들어, **int\* pValue;**는 정수형 포인터를 나타내는 변수입니다.
2. **n**: "숫자"를 나타내는 관행적인 약어입니다. 변수 이름에서 **n**이 사용되면 해당 변수가 정수나 숫자와 관련된 값을 나타내는 변수임을 나타냅니다. 예를 들어, **int nCount;**는 정수형 숫자를 나타내는 변수입니다.
3. **b**: "부울"을 나타내는 관행적인 약어입니다. 변수 이름에서 **b**가 사용되면 해당 변수가 부울 (불리언) 값, 즉 **true** 또는 **false** 값을 나타내는 변수임을 나타냅니다. 예를 들어, **bool bIsEnabled;**는 부울 값이 활성화되었는지 여부를 나타내는 변수입니다.
4. **sz**: "문자열 크기"를 나타내는 관행적인 접두어입니다. 변수 이름에서 **sz**가 사용되면 해당 변수가 문자열의 크기를 나타내는 변수임을 나타냅니다. 예를 들어, **size\_t szLength;**는 문자열의 길이를 저장하는 변수입니다.
5. **i**, **j**, **k**: 정수형 변수에 자주 사용되는 한 글자 약어입니다. 일반적으로 반복문에서 사용되며, **i**는 첫 번째 루프, **j**는 두 번째 루프, **k**는 세 번째 루프 등을 나타내는 관행적인 변수 이름입니다.

Framework내에 변수에 대해 항목별로 만들어놨다.

요약하자면 OnCreate일 때 모든 변수들을 초기화 시켜서 만드는 과정을 가진다.

OnDestroy일때는 모든 메모리 할당을 Release해준다.

Swap Chain을 만들 때(CreateSwapChain() ) 픽셀의 형식을 설정하는 부분이 존재한다.

현재 따라하기에선 **DXGI\_FORMAT\_R8G8B8A8\_UNORM**이라는 Format을 지정하였다.

이는 색과 투명도가 8비트의 값으로 할당되는 것을 의미한다고 한다.

다른 자주 사용하는 것은 아래와 같다

1. **DXGI\_FORMAT\_R8G8B8A8\_UNORM**: 이미 이전에 언급한 형식으로, 8비트로 각 채널을 표현하고 정규화되지 않은 형식입니다.
2. **DXGI\_FORMAT\_R8G8B8A8\_UINT**: 8비트로 각 채널을 표현하며, 정수 형식으로 표현된 형식입니다. 정규화되지 않고 정수 형식으로 데이터를 저장할 때 사용됩니다.
3. **DXGI\_FORMAT\_R16G16B16A16\_FLOAT**: 16비트로 각 채널을 표현하고 부동 소수점 형식으로 저장되며, 고해상도 색상 및 투명도 데이터를 나타내는 데 사용됩니다.
4. **DXGI\_FORMAT\_D32\_FLOAT**: 32비트 단일 채널 형식으로, 깊이(depth) 정보를 부동 소수점 형식으로 저장할 때 사용됩니다. 주로 깊이 버퍼(depth buffer)에 사용됩니다.
5. **DXGI\_FORMAT\_BC1\_UNORM**, **DXGI\_FORMAT\_BC2\_UNORM**, **DXGI\_FORMAT\_BC3\_UNORM**: BC 형식은 블록 압축 형식으로 압축된 텍스처 데이터를 나타냅니다. 이러한 형식은 텍스처 압축 및 저장 공간을 절약하는 데 사용됩니다.
6. **DXGI\_FORMAT\_R32G32B32A32\_FLOAT**: 32비트로 각 채널을 표현하고 부동 소수점 형식으로 저장되며, 고해상도 컬러 및 투명도 데이터를 나타내는 데 사용됩니다.
7. **DXGI\_FORMAT\_R8\_UNORM**: 8비트로 단일 채널을 표현하고 정규화되지 않은 형식으로 사용되며, 그레이스케일 이미지를 저장할 때 사용됩니다.
8. **DXGI\_FORMAT\_R16G16\_SINT**: 16비트로 각 채널을 표현하고 부호 있는 정수 형식으로 사용됩니다.

Framework에 대한 주석을 전부 작성하고 코드를 분석하다가 몇가지 버그가 발생하여서 디버깅하는데 시간이 좀 걸렸다.

버그의 원인은 제대로 된 인자를 작성하지 않은 것과 Dsv의 포인터 위치를 바꿀 필요가 없음에도 바꾼것이다.

추가적으로 이용희교수님께 온 답장을 보자면, 내가 착각을 한것이였다.

당연하게도 내가 만든 CreateSwapChain은 파라메터가 존재하지도 않는데 이상한 착각을 한것이다. 내가만든것과는 다르게 dxgi.h내에 존재하는 제공함수였던것이다. 밥안먹고 잠을 안자서 판단이 흐려진것같다.

따라하기04(LabProject03) – 프레임 레이트 출력

이제 1초에 몇 번 그림을 그리는지에 대한 FPS정보를 화면에 출력해보자.

더 나아가 게임의 장면들을 관리하기위한Scene클래스도 추가적으로 구현한다.

Tick 함수 구현 도중 신기한 부호를 발견했다. /= 라는 부호인데

이건 A /= B같이 사용하고, 뜻은 A를 B로 나누고 그 값을 A에다가 대입하는거라고 한다.

또 GetFrameRate함수 내에 \_itow\_s라는 함수도 있었다.

이는 정수를 유니코드 문자열로 변환해주는 함수라고 한다. 여기서 접미사 \_s는 안전한 버전이라버 하는 것을 보아 Safty인것같다.

다음으로 wcscat\_s라는 함수도 존재하였다.

이는 문자열을 연결시켜주는 함수로 내가 작성한 코드에선 FPS를 연결해주는 역할을 수행하는것으로 보인다.

프레임레이트 출력 부분에서 문제가 발생하였다.

LabProject(까지만 출력되고 프레임워크 FPS부분이 출력이 되지 않았다.

30분간 코드를 들여다본 결과 GameFramework부분 어딘가에 문제가 발생했다고 판단했다.

정답을 알았다.

처음 프레임워크 변수를 초기화할 때에\_tcscpy\_s(m\_pszFrameRate, \_T("LapProject(")); 라고 작성하였는데.

FrameAdvance()함수 내 마지막을 보자면

m\_GameTimer.GetFrameRate(m\_pszFrameRate + 12, 37);

::SetWindowText(m\_hWnd, m\_pszFrameRate);

즉 m\_pszFrameRate +12부터 프레임을 작성하도록 되어있는데

위에 초기화 값으로는 한칸이 부족해진다.

그래서 띄어쓰기를 위한 스페이스바 한번을 눌러주면 정상적으로 출력한다.

결국 숫자계산을 못해서 발생한 해프닝이였다. 에휴….

m\_ppd3dSwapChainBackBuffers에 대한 항목이 없었는데 m\_ppd3dRenderTargetBuffers라고 이름지었었다. 그래서 여기서부터 이름을 바꾸었다.

여기서 알아둘 중요한 점은

ChangeSwapChainState 함수에서 CreateRenderTargetView를 호출하기 때문에 더 이상 OnCreate에서 RenderTargetView를 생성할 필요가 없다.

1차 정리

이번 챕터 이후부터는 본격적으로 삼각형을 그리기 시작할것이다. 그전에 지금까지 했던 작업을 돌아보며 정리하는 시간을 가지려고 한다.

화면에 그림이 그려지는 과정

전면 버퍼(Front Buffer)와 후면 버퍼(Back Buffer)로 구성되어 있고

전면 버퍼가 모니터에 출력되는 버퍼이고

후면 버퍼는 응용프로그램이 랜더링을 해야되는, 그러니까 그림을 그려줘야 하는 대상이다

그 후면 버퍼에 그린 그림을 전면 버퍼로 옭기는 행위를 Presentation이라 하며, 그 방법에는Flipping과 Blit가 있다.

이렇게 프레임 버퍼들(전면과 후면들)이 순차적으로 연결된 것을 Swap Chain이라고 한다.

결국 크게 보자면 Rendering과 Present를 아주 빠르게 무한히 번갈아 가며 하는 것이 화면에 그림이 그려지는 과정인것이다.

\_\_uuidof(ID3D12Device), &pd3dDevice 를 IID\_PPV\_ARGS(&pd3dDevice)로 바꿔 쓸 수 있다는 사실도 추가적으로 복습하면서 알았다.

추가적으로 ComPtr을 이용하면 자동으로 Release()를 해준다.

사용방법은 ID3d12Device \*pd3dDevice처럼 만들던 것을, ComPtr<ID3D12Device> pd3dDevice와 같은 형태로 만들면 되는것이다.

되게 예제는 고급 스킬보단 정석 적인 방법으로 프로그래밍을 한다고 여기서 느꼈다.

결국 DXGI의 역할은 하드웨어 디바이스 열거, 모니터 열거, Present, Full Screen Mode 전환등을 한다.

여기부터는 Swap Chain의 생성 방법에 대한 대략적인 설정방법들을 간단하게 정리했다.

MSAA는 멀티샘플 안티앨리어싱의 약자로, Direct3D에서 제공하는 계단현상 제거 기법이다.

DXGI에서 후면버퍼를 Render Target이라고 부를것이다.

즉 DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT은 내가 만든 버퍼의 설정을 Render Target의 OutPut을 위해 사용할것이라고 설정하는 것이다. 그래서 Swap Chain을 만들 때 저걸 사용하는 것이다.

다음으로 SXGI\_SWAP\_EFFECT 설정 중, DXGI\_SWAP\_EFFECT\_FLIP\_DISCARD가 있는데

여기서 커다란 옵션이 두 개 존재한다. DISCARD와 SEQUENTIAL이 있는데

DISCARD는 버퍼의 내용을 폐기, 즉 버리는것이다.

그 다음으로 FLIP의 여부인데, FLIP은 전면버퍼와 후면버퍼를 빠르게 교체한다는 의미이다.

FLIP을 하고나서 해당 내용을 수행할것이다 라는것이다.

즉 저 내용은 FLIP을 하고나서 후면 버퍼의 내용을 버리는 것이다.

그리고 DISCARD옵션을 사용한다면 속도가 빨라서 대부분 사용한다.

SEQUENTIAL은 후면버퍼가 여러 개일때, 후면버퍼들의 내용을 순차적으로 옮길 때 사용한다.

즉 2개면 DISCARD 3개 이상이면 SEQUENTIAL을 사용하는것이다.

Swap Chain에서 Flag에 대한 내용에서 DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH에 대한 내용만 간단하게 알아갔다.

해당 내용은 모니터의 디스플레이 모드를 변경할 수 있게 만들어주는것이다.

더 쉽게 말해서 윈도우 바탕화면의 해상도를 변경해주는 것이다.

이는 전체화면모드와 윈도우모드 사이의 전환을 할 때, 혹은 바탕화면의 해상도를 바꿔야만 할 때 사용하는 것이다.

생성한 다음 MakeWindowAssociation이란 함수가 존재한다.

그중 인자로DXGI\_MWA\_NO\_ALT\_ENTER라는 내용을 집어넣는다.

이는 Alt + Enter를 사용하지 못하게 막겟다라는 것이다.

왜 막느냐? 이걸 호출 안하고 Swap Chain을 만들고 Alt Enter를 누른다면, 윈도우 출력에서 Swap Chain의 모드가 전체화면 모드와 윈도우 모드가 Toggle로 전환이되도록 자동으로 DXGI가 제공을 해버리기 때문이다.

여기까지가 Swap Chain을 생성하는 방법이다.

d이렇게 Swap Chain을 만든다면, 우리는 이제 Swap Chain Interface Pointer를 나의 응용프로그램이 가지게 된다(m\_pdxgiSwapChain).

저 인터페이스 포인터를 이용해서 여러가지 멤버 함수들을 드디어 사용을 할 수 있다.

이렇게 만든 Swap Chain은 Frame Buffer들의 리스트 형식이다. 이를 DXGI는 Surface라는 용어로 표현한다.

대표적인 멤버함수가 바로 Present이다. 이름 그대로 후면 버퍼의 내용을 전면 버퍼로(모니터로)옮겨줘라는 명령이다.

다른것으로는 GetBuffer로 후면버퍼들을 가져오는것이고.

GetContainingOutput은 현재 스왑체인에 연결된 윈도우 클라이언트 영역의 대부분(Output)을 출력해달라는 것이다.

말이 어려운데, 듀얼모니터를 쓸 때 윈도우 창을 정 가운데에 두고, 오른쪽으로 좀 더 옮기면 대부분 오른쪽에서 출력이 일어나면 그 출력이 나오는 Output을 리턴 받는 것이다. 그래서 예제코드에는 해당 내용이 존재하지 않는다.

다른 Get함수들은 통계정보, 현재 스왑체인의 출력모드가 윈도우인지 전체화면인지를 물어보는 BOOL함수이다.

SetFullscreenState함수를 사용하면 전체화면 모드로 강제적으로 변환이 된다.

ResizeBuffers, ResizeTarget함수는 좀 자세히 다뤄보겟다.

ResizeBuffers 함수는 후면버퍼의 개수, 크기 format을 필요에 따라 바꿔주는 함수들이다.

ResizeTarget 함수는 디스플레이 모드(바탕화면)의 Format이나 크기 등을 바꾸는 것이다.

근데 이러한 디스플레이 모드들을 바꾸기 위해서는 Swap Chain을 만들 때에 Flag 내용에 위에서언급했던 ALLOW\_MODE\_SWITCH라는 설정을 꼭 해줘야 한다.

Present에서는 SyncInterval이라는 사용안해본 내용이 있는데, 이는 수직동기화와 관련된 개념이다. 숫자를 집어넣어 그 숫자만큼의 수직회귀를 기다리고 Present하라는 기능이다.

당연하게도 0을하면 Present가 가장 빠르게 작동한다.

Flags는 자주 사용해보았는데, 값으로는 0을 사용하거나, DXGI\_PRESENT\_(DO\_NOT\_SEQUENCE / DO\_NOT\_WAIT / RESTART)가 존재한다.

0은 시퀀싱(Sequencing)을 하는 것, 각 후면버퍼들의 내용들을 순차적으로 앞의 버퍼를 옮기는 것이다. 이를 일반적으로 가장 많이 사용할것이다.

CreateSwapChainForHwnd함수에서

DXGI\_SWAP\_CHAIN\_DESC1의 내용은 그냥 SwapChainDesc의 내용과 거의 유사하다.

다만 다른 점이 존재하는데, 바로 Stereo설정이다.

이는 해당 내용에만 있는데. 이 설정은 입체 출력을 하는것이다. 입체처럼 착시효과를 일으키는 것이다.

또한 FullScreen에 대한 Swap Chain모드를 세부적으로 설정 할 수 있는 부분이 존재한다.

또한 ALPHA\_MODE도 설정할 수 있다.

CreateSwapChain이나, CreateSwapChainForHwnd함수를 사용하여도 똑같이 Swap Chain이 만들어지지만. 추가적인 설정을 할 수 있는것이다. 이는 버전 번호에 따라서 달라질 수 있는것이다.

전체화면 모드(Fullscreen Mode)와 윈도우 모드(Window Mode)는

Alt + Enter를 Disable을 안하면(아까 위에서 설정한거) DXGI가 알아서 해준다.

근데 우리는 Alt + Enter를 Disable을 해도 전환을 할 수 있다.

GetFullscreenState와 SetFullscreenState를 사용하면 가능해진다.

GetFullscreenState는 Swap Chain의 현재 화면모드를 가져온다.

SetFullscreenState는 전체화면 모드와 풀스크린모드를 이 함수를 호출함으로 바꾸어주는 기능을 한다.

윈도우 클라이언트의 크기가 변경이 발생하면

윈도우 운영체제는 WM\_SIZE라는 메시지를 발생한다.

윈도우의 바뀐 크기에 따라서 후면버퍼의 크기를 바꿔줘야 할것이다.

그 후면버퍼의 크기를 바꿔주는 함수는 위에서 말했던 ResizeBuffers라는 함수이다.

근데 이 후면버퍼 크기를 바꿀라 하면, 이전에 만든 그 후면버퍼의 모든 참조들을 Release, 즉 해제해줘야 한다. 이렇게 되면 바꾼 ResizeBuffers함수를 이용해 새로 만들어줘야 하는것이다.

근데 사실상, 게임프로그램을 만들 때 클라이언트의 크기, 후면버퍼의 크기를 바꿀 필요가 없다.

왜? 리소스에 따라서 크기가 정해져있기 때문에 바꿀 필요가 없게 되는것이다.

화면 모드 전환은 DXGI가 강제적으로 정해진 바탕화면크기로 윈도우의 크기를 키우는 것이다. 근데 이렇게 하면 후면버퍼가 전면버퍼의 크기와 달라서 Present를 할 수 없게 된다.

그래서 출력 윈도우 크기가 자동으로 변경이 되게 하면, 나의 바탕화면 크기를 후면버퍼 크기로 바꿔버려서 전면과 후면의 크기가 같게 만드는것이다.

즉, 후면버퍼 크기에 맞게 바탕화면 크기를 바꾸고 모드의 전환을 하는것이다. 그걸 위해서 ALLOW\_MODE\_SWITCH라는 플래그로 SwapChain을 만들고 ResizeTarget함수를 사용해서 바탕화면 크기를 바꾼다음에 전체화면 모드로 바꾸는것이다.

여기서 중요한점이 ResizeTarget은

Swap Chain이 윈도우 모드일때는 윈도우 크기를 변경해주고

Swap Chain이 전체화면 모드일때는 바탕화면 모드를 변경해준다는것이다.

여기서 OnResizeBackBuffers()라는 함수가 존재하는데 이에 대해서 교수님께서 설명하지 않으셨다.

나중에 코드에 추가해볼까 한다.

DescriptorHeap은 서술자 힙이라고도 하고, 모든 리소스들을 Descriptor로 표현하기 위해 필요한 메모리 블록(배열형식)이다.

그 디스크립터 힙의 주소를 표현하려고

D3D12\_CPU\_DESCRIPTOR\_HANDLE과 D3D12\_GPU\_DESCRIPTOR\_HANDLE이라는 구조체를 사용한다.

근데 여기서 이걸 직접써도 되고

C++형식으로 사용하기 편하도록

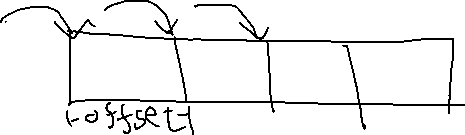
CD3DX12\_CPU\_DESCRIPTOR\_HANDLE과 CD3DX12\_GPU\_DESCRIPTOR\_HANDLE 형태를 제공한다.

위 구조체는 생성자와 비교연산자를 제공한다.

일단 우리 따라하기는 위 구조체를 사용하지 않는다.

그치만 위 구조체에서 가장 중요한 요소는 Offset인데

Offset은 내부에서 다음원소의 위치를 자동으로 설정해준다



위 그림같이 offset값만큼 띄워서

그다음 원소의 위치를 알게 해주는것이다.

우리는 직접 설정해서 사용할것이다.

여기서부터는 Descriptor Heap에 대해 자세히 다룰것이다.

Descriptor Heap을 무조건 써야지만 Descriptor를 사용할 수 있는 것은 아니다.

상수 버퍼 뷰 – CBV : Constant Buffer View

무순서화 접근 뷰 – UAV : Unordered Access View

쉐이더 리소스 뷰 – SRV : Shader Resource View

샘플러 – Sampler(Sampler는 그 자체로도 View이다)

렌더 타겟 뷰 – RTV : Render Target View

깊이 스탠실 뷰 – DSV : Depth Stencil View

위의 뷰들은 모두 Heap이 필요한 반면

정점 버퍼 뷰 – VBV : Vertex Buffer View

인덱스 버퍼 뷰 – IBV : Index Buffer View

스트림 출력 뷰 – SOV : Stream Output View

위 3종류는 Heap이 없어도 만들 수 있는 뷰들이다.

위에 Heap이 필요한 뷰들을 만들 때의 순서는 이렇게 된다.

해당 속성에 해당하는 Descriptor Heap을 만들고, 만든 Heap의 위치를 정한 다음, 그곳에다가 Descriptor를 만들어 줘 라고 알려주어야 한다.

더 나아가 UAV, CBV, SRV 세 종류는 같은 타입의 Descriptor Heap으로 만들 수 있다.

추가적인 설명은 그림 자료와 함께 충당하겟다.

View를 Set하려면 View를 Create해야하고, View를 Create하기위해선 Resource가 필요하다.

왜? View는 Resource를 설명해주기 위한 기능이기 때문이다.

그 Resource를 설명하기 위한 View를 만들려면 View를 만들기 위한 메모리가 필요하다.

그 메모리가 바로 Descriptor Heap인것이다.

가장 먼저 해야되는 것은 Resource를 만드는것이다.

만드는 방법은 CreateCommitedResource와 같은 함수로 Resource를 만들어야 한다.

다음으로 만든 Resource로 View를 만드는것이다. 이중 Heap이 필요한 뷰를 만든다고 한다면

Descriptor Heap을 만드는것이다.

이때 Resource의 타입에 따라서 Descriptor Heap을 따로 만들어줘야 한다.

앞서 이야기했듯, Descriptor Heap은 Descriptor를 저장하기 위한 배열인데. 그래서 Heap을 만들 때 몇 개의 원소를 가질지, Descriptor가 몇 개인지를 확인하고 Heap을 만드는 과정이 필요하다.

세번째는 View를 만들 때, Resource에 해당하는 Descriptor를 가리키는 Create View 인터페이스 포인터가 필요하다. 그리고 그 포인터는 만든 Heap의 시작주소를 최종적으로 가리킨다.

친필, 아동 미술, 텍스트, 그림이(가) 표시된 사진

자동 생성된 설명

저렇게 Resource에 대해 View를 만들고, 그 View에 대한 Descriptor는 저기다가 넣어줘 라는 의미이다.

그렇게 만들었다면 우리는 Descriptor Heap의 주소를 가지고 있을 것이다.

그렇다면, 이제 Set작업을 해줄것이다.

그 Set작업은 방금 말한 Descriptor Heap의 주소를 넘겨주면 된다.

이렇게까지가 Resource를 최종적으로 View로 Set해주는 방법이다.

만약 또 다른 Resource를 이용해서 또다른 View를 만들려고 한다면 아까와는 또 다른 위치의 Heap에 저장하고, 같은 방법으로 View를 Set해주면 된다.

친필, 아동 미술, 그림, 스케치이(가) 표시된 사진

자동 생성된 설명

위처럼 처음거와 다른 위치의 Heap을 가리키고 그곳의 시작 주소를 건네 주는 방식으로 여러가지 Resource를 Set 해줄 수 있다.

그리고 이러하게 Resource를 저장한 곳을 알려주고 건네 주는 등의 작업을 하는 것이 바로 Pipeline이 하는 것이다.

그 Pipeline은 10개의 단계가 존재하며, 고정파이프라인 5개와, Shader 5단계로 이루어져 있다.

그 Pipeline에서 방금 Set한 View들을 읽어와서 Resource가 어떤 타입인지 뭐가 있는지를 알 수 있는 것이다.

View의 개수가 정해져있지 않듯, Heap의 개수도 정해진 형식과 개수가 없다.

키포인트, Resource를 만드는 함수, Heap을 만드는 함수, View를 만드는 함수는 모두 Device 내에 존재한다.

그와 반대로 View를 Set하는 함수는 CommandList 함수 내에 존재한다.

이것이 의미하는 바는.

Set View를 하는 CommandList는 1초동안 60번 호출하는데 반해, Resource, Heap, Create View들은 미리 만들어 둘 수 있다는 이야기이다.

CreateRenderTargetView를 먼저 보자면

Render Target은 Output Merger에 연결을 해야한다. 실제로는 Render Target을 연결하는 것보단. Render Target을 가리키는 View를 연결하는 것이다. 이는 최대 8개까지 연결을 해줄 수 있다.

View라고 하는 것은 Resource를 설명하는 내용이다. 즉 Resource를 어떤 식으로 사용할건지에 대해 말한거라고 할 수 있다.

근데 그말은? 다르게 볼 수도 있다는 뜻이다.

가령 1.0f라는 값을 준다면 그걸 색으로도 볼수있고, 좌표로 볼수있고, 투명도로도 볼수있다.

그걸 어떤 의미로 볼지를 결정한다는게 View이다.

그럼 하나의 Resource로 하나 이상의 View를 만드는것도 가능하다는것이다(필요에 따라서)

예시로, Resource에 대해서 View를 RTV와 SRV 두가지로 만드는 것이다. 이 말 뜻은 Rendering의 대상으로 볼수도 있고, Shader를 만들 때의 정보로도 볼 수 있다는 것이다.

CreateDepthStencilView 또한 앞의 RTV와 비슷하다.

근데 타입이 다르다.

D3D12\_CPU\_DESCRIPTOR\_HANDLE d3dRtvCPUDescriptorHandle = m\_pd3dRtvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();

m\_pd3dDevice->CreateRenderTargetView(m\_ppd3dSwapChainBackBuffers[i], NULL, d3dRtvCPUDescriptorHandle);

위는 CreateRenderTargetView에 대한 LabProject내의 코드 일부이다.

D3D12\_CPU\_DESCRIPTOR\_HANDLE d3dDsvCPUDescriptorHandle =

m\_pd3dDsvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();

m\_pd3dDevice->CreateDepthStencilView(m\_pd3dDepthStencilBuffer, NULL, d3dDsvCPUDescriptorHandle);

아래는 CreateDepthStencilView의 예시코드이다.

둘은 Descriptor Handle의 인자가 다르다. 즉 따로 다른형태로 만들어주어야만 한다는 것이다.

추가적으로 DepthStencilView 의 Format의 형식은 D혹은 S로 시작하는 타입만 사용이 가능한것이다. (예시 코드 d3dResourceDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT; )

Flag 설정으로 Depth만 읽을것인지, Stencil만 읽을것인지, 아님 둘다 쓸것인지에 대한 설정도 가능하다.

그럼 여기서 물음이 발생할것이다.

근본적으로 우리는 왜 Depth Buffer가 필요한가?

윈근투영변환을 하여서 이미지가 찍히는 오브젝트들의 Pixel 에 대한 Z값(카메라와의 거리)값들을 저장할 것이기에 Depth Buffer인것이다.

이렇게 저장한 Depth Buffer는 화면을 랜더링하거나 앞으로 조명의 그림자를 계산하거나, 여러모로 쓰이기 때문에 당연히 만들어야 하는 것이다.

좀 전문적으로 하자면 Painter’s Algorithm을 이용하기위해선 오브젝트의 멀고 가까움에 대한 정렬(Sorting)이 필요하다. 근데 오브젝트들을 정렬해서 한 프레임에 수만개의 Pixel을 그린다는 것은 불가능하다고 할 수 있다. 그래서 Depth값만 따로 저장하도록 하는것이다.

이때 최종적으로 화면에 그려지는 Pixel들을 보이도록 Depth 값들을 저장하는 곳을 DXGI에서는 Frame Buffer라고 하고, Direct3D는 Render Target이라고 한다.

이때 Depth Buffer들을 1.0으로 초기화를 진행한다. 1.0의 의미는 절두체에서 0과 1사이만 오브젝트 들이 렌더링이 되는데 1.0으로 초기화 해서 더 가까운 오브젝트들이 생기면 그 오브젝트의 Depth값을 읽어서 1.0에서 그 값으로 갱신해주는것이다.

다시 Depth로 돌아와서 Flag 형식을 보자면

DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT라는 예제가 있었다.

여기서 D24는 Depth값으로 24비트를 사용하고 0과 1 사이 값으로 Unsigned Normalize되어 있으며, Stencil은 8비트로 사용할것이고 UINT형식으로 사용한다는 의미이다.

다른것도 마찬가지 의미로 해석하면 된다.

이렇게 RTV와 DSV를 만들었는데, Resource를 안만들어두면 의미가 없게 된다.

그럼으로 우리는 Resource를 생성해야 하는데, 그 함수가 바로 CreateCommittedResource이다.

이 함수를 설명하자면, 가상메모리(Virtual Memory)를 할당하면서 동시에 물리 메모리(Physical Memory)도 같이 할당해서 Resource를 만든다.

이때 다른 리소스 만들기 함수로는 CreatePlacedResource와 CreateReservedResource가 존재한다.

두 함수는 물리메모리와 쌍으로 할당하는게 아니라서 메모리공유나 Virtual메모리로 Resource를 사용할 수 있다는 장점이 존재하지만. Heap을 만든다거나 물리메모리로 맵핑을 해야된다라는 번거로운 단점이 존재한다.

그래서 우리는 다루지 않을 것이다.

최종적으로 Resource만들고, View만들고 셋해서 RTV와 DSV를 만들면

그 만든 것들을 Pipeline에 연결을 하기 위해서, Output Merger에 연결을 할 수 있다.(예시 함수 OMSet…..)

이때 RTV는 색을 리턴하고(Frame Buffer이니까), DSV는 깊이값을 리턴한다.

그럼 OM에 연결된건 View가 알고있는 해당 View의 주소, Descriptor Heap의 주소이다. OM은 그 View로 가서 해당 Resource를 읽어서 출력하는 것이 된다. 그럼 최종적으로 이미지가 옮겨지게 되는것이다.

그렇다면 Resource를 만드는 과정에 대해 좀 더 자세히 분석해보도록 하자.

Heap을 만들 때에 먼저 D3D12\_HEAP\_PROPERTIES를 설정하게 되는데.

Type의 종류는 Default, Upload, ReadBack, Custom이 존재한다.

먼저 Default는 GPU가 메모리를 읽고 쓰는 것이 가능하게 해주는 타입이다.

친필, 그림, 아동 미술, 스케치이(가) 표시된 사진

자동 생성된 설명

Upload HEAP TYPE은 위에서 CPU에서 Resource의 데이터를 쓸 수(WRITE) 있고, GPU는 Resource를 당연하게 쓸 수 있다.

왜 이러한 UPLOAD힙이 필요한가 하면, 대부분 게임 리소스는 응용 프로그램이 가지고 있다.

그 데이터를 비디오 메모리한테 옮겨줘야 한다. 그 비디오 메모리는 GPU가 읽고 쓸 수 있는 권한을 가지고있다.

그런데 여기서 Default를 하면 CPU가 가지고 있는 리소스 데이터를 못쓴다.

그래서 Upload힙을 만들어야 하는 것이다. 그 다음에 Upload의 내용을 Default Heap에다가 Copy해서 GPU한테 건네주는 것이다.

그림, 스케치, 아동 미술, 도표이(가) 표시된 사진

자동 생성된 설명

다음은 READBACK인데, CPU에서 리소스 데이터를 읽어오는 것을 의미한다.

대부분의 데이터 흐름은 CPU의 게임 리소스들 🡪 Default Heap 🡪 GPU가 그 내용을 읽음

의 흐름으로 진행이 되게 하기 위해 이런 통로를 많이 만드는 것이다. 그래서 엄청 빠르다.

그래서 그걸 거스르는, GPU가 읽는 화면에 그릴 내용을 CPU가 읽어갈 작업은 거의 없다. 그래서 엄청 느리다.

그래서 ReadBack HEAP은 최대한 사용하면 안된다.

다음인자로 HEAP FLAG는 그냥 NONE를 쓰면 된다.

다음으로 HEAP을 만들 때 필요한 설정이 ClearValue인데.

Resource를 최적의 방법으로 초기화 하기 위한 함수이다. 이를 이용하면 빠르게 클리어를 할 수 있는것이다.

이를 정해주지 않으면 속도가 많이 느려질 수 있게 된다.

D3D12\_RESOURCE\_STATE설정은 Resource들은 사용하다보면 바뀔 수도 있는데, 초기값을 세팅해줘야 한다. 그 초기상태를 설정해주는 것이 이 파라메터이다.

VERTEX\_AND\_CONSTANT\_BUFFER는 Vertex, Constant형태의 버퍼이면 이 상태를 사용하는것이다.

마찬가지로 INDEX\_BUFFER 또한 Index 형식이라면 이 상태로 설정해줘야 하며

RENDER\_TARGET, UNORDERED\_ACCESS도 마찬가지이며

Depth Stencil은 DEPTH\_READ와 DEPTH\_WRITE 설정이 존재한다.

PIXEL\_SHADER\_RESOURCE, STREAM\_OUTPUT 등등 많이 존재한다.

나머지는 직접 확인

typedef enum D3D12\_RESOURCE\_STATES {

D3D12\_RESOURCE\_STATE\_COMMON = 0,

D3D12\_RESOURCE\_STATE\_VERTEX\_AND\_CONSTANT\_BUFFER = 0x1,

D3D12\_RESOURCE\_STATE\_INDEX\_BUFFER = 0x2,

D3D12\_RESOURCE\_STATE\_RENDER\_TARGET = 0x4,

D3D12\_RESOURCE\_STATE\_UNORDERED\_ACCESS = 0x8,

D3D12\_RESOURCE\_STATE\_DEPTH\_WRITE = 0x10,

D3D12\_RESOURCE\_STATE\_DEPTH\_READ = 0x20,

D3D12\_RESOURCE\_STATE\_NON\_PIXEL\_SHADER\_RESOURCE = 0x40,

D3D12\_RESOURCE\_STATE\_PIXEL\_SHADER\_RESOURCE = 0x80,

D3D12\_RESOURCE\_STATE\_STREAM\_OUT = 0x100,

D3D12\_RESOURCE\_STATE\_INDIRECT\_ARGUMENT = 0x200,

D3D12\_RESOURCE\_STATE\_COPY\_DEST = 0x400,

D3D12\_RESOURCE\_STATE\_COPY\_SOURCE = 0x800,

D3D12\_RESOURCE\_STATE\_RESOLVE\_DEST = 0x1000,

D3D12\_RESOURCE\_STATE\_RESOLVE\_SOURCE = 0x2000,

D3D12\_RESOURCE\_STATE\_RAYTRACING\_ACCELERATION\_STRUCTURE = 0x400000,

D3D12\_RESOURCE\_STATE\_SHADING\_RATE\_SOURCE = 0x1000000,

D3D12\_RESOURCE\_STATE\_GENERIC\_READ,

D3D12\_RESOURCE\_STATE\_ALL\_SHADER\_RESOURCE,

D3D12\_RESOURCE\_STATE\_PRESENT = 0, // RESOURCE\_STATE\_COMMON과 같음, 후면버퍼의 내용을 전면버퍼에 옮기는것

D3D12\_RESOURCE\_STATE\_PREDICATION = 0x200,

D3D12\_RESOURCE\_STATE\_VIDEO\_DECODE\_READ = 0x10000,

D3D12\_RESOURCE\_STATE\_VIDEO\_DECODE\_WRITE = 0x20000,

D3D12\_RESOURCE\_STATE\_VIDEO\_PROCESS\_READ = 0x40000,

D3D12\_RESOURCE\_STATE\_VIDEO\_PROCESS\_WRITE = 0x80000,

D3D12\_RESOURCE\_STATE\_VIDEO\_ENCODE\_READ = 0x200000,

D3D12\_RESOURCE\_STATE\_VIDEO\_ENCODE\_WRITE = 0x800000

} ;

D3D12\_HEAP\_TYPE\_UPLOAD의 초기상태는 D3D12\_RESOURCE\_STATE\_GENERIC\_READ로 설정해줘야 한다. copy연산으로 위에서 말했던것처럼 Upload힙에서 Default 힙으로 옮겨줘야 하기 때문에

D3D12\_HEAP\_TYPE\_RAEDBACK의 초기상태는 D3D12\_RESOURCE\_STATE\_COPY\_DEST로 설정해줘야 한다

텍스쳐는 CPU의 접근을 위해 D3D12\_RESOURCE\_STATE\_COMMON이라는 타입으로 설정해줘야 한다.

마지막 파라메터로 Resource Description은 Heap에 대한 속성이 아닌 Resource의 속성을 설정해주는 부분이다.

나머지는 LabProject코드에서 void CGameFramework::CreateDepthStencilView()에 대한 함수 내용을 참고하자. 모두 주석을 달아놨다.

추가적으로 중요한 점은, Upload Heap에는 Texture를 생성할 수 없다.

그래서 Texture는 반드시 Default Heap에 만들어야만 한다.

이렇게 Resource를 Create 했다면,

CreateRtvAndDsvDescriptorHeaps로 RTV와 DSV의 HEAP을 만들고

CreateRenderTargetViews로 RTV를 만들면 된다.

여기서 좀 중요한 내용은

void CGameFramework::CreateRenderTargetViews()

{

// 스왑체인의 각 후면버퍼에 대한 랜더 타겟 뷰를 생성한다.

D3D12\_CPU\_DESCRIPTOR\_HANDLE d3dRtvCPUDescriptorHandle = m\_pd3dRtvDescriptorHeap->GetCPUDescriptorHandleForHeapStart(); // Descriptor Heap의 시작 주소를 가져온다.

for (UINT i = 0; i < m\_nSwapChainBuffers; i++) // Swap Chain의 개수만큼

{

m\_pdxgiSwapChain->GetBuffer(i, \_\_uuidof(ID3D12Resource), (void\*\*)&m\_ppd3dSwapChainBackBuffers[i]); // 여기서 백버퍼의 내용을 가져와서

m\_pd3dDevice->CreateRenderTargetView(m\_ppd3dSwapChainBackBuffers[i], NULL, d3dRtvCPUDescriptorHandle); // RTV가 그려야되는 내용을 알려준다. (리소스에 대한 인터페이스 포인터, View에 해당하는 Description, Descriptor Heap의 주소)

d3dRtvCPUDescriptorHandle.ptr += m\_nRtvDescriptorIncrementSize; // 다음 RTV를 위한 디스크립터 핸들을 준비하기 위해 포인터의 위치를 다음으로 넘긴다.

}

}

코드의 내용중 포인터 이동이 존재하는데

텍스트, 라인, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

위 그림을 보면 왼쪽에 서술자 힙이 Swap Chain의 개수(2개)만큼 만들어 진 것이 보인다.

처음 위치에 처음 후면버퍼를 가져다가 리소스를 만들고

다음 위치로 포인터이동해서 다음 위치에 그린다.

후면버퍼를 RTV로 만든것이다

만약 현재의 Swap Chain의 인덱스가 0, 1이고, 0번의 후면버퍼 내용을 OM에 Set하려면, DescriptorHeap의 시작주소로 가서 Clear하고 OM에 SET한다

그 과정을 보여주는 곳이

// 현재의 Render Target에 해당하는 Descriptor의 CPU Handle(주소)을 계산

D3D12\_CPU\_DESCRIPTOR\_HANDLE d3dRtvCPUDescriptorHandle = m\_pd3dRtvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();

d3dRtvCPUDescriptorHandle.ptr += (m\_nSwapChainBufferIndex \* m\_nRtvDescriptorIncrementSize); // 계산하고나서 다음으로....

// 원하는 색상으로 Render Target View를 지운다.

float pfClearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f }; // RGBA, 해당 색상은 Azure이다.

m\_pd3dCommandList->ClearRenderTargetView(d3dRtvCPUDescriptorHandle, pfClearColor, 0, NULL);

// Depth Stencil Descriptor의 CPU Handle(주소)을 계산

D3D12\_CPU\_DESCRIPTOR\_HANDLE d3dDsvCPUDescriptorHandle = m\_pd3dDsvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();

// 원하는 값으로 Depth Stencil View를 지운다.

m\_pd3dCommandList->ClearDepthStencilView(d3dDsvCPUDescriptorHandle, D3D12\_CLEAR\_FLAG\_DEPTH | D3D12\_CLEAR\_FLAG\_STENCIL, 1.0f, 0, 0, NULL);

// Render Target View와 Depth Stencil View를 OM(출력-병합 단계)에 연결

m\_pd3dCommandList->OMSetRenderTargets(1, &d3dRtvCPUDescriptorHandle, TRUE, &d3dDsvCPUDescriptorHandle);

해당 코드이다.

다 만들고나서 필요한 순간에 Clear를 해야한다.

정확히는 View를 클리어 하는 것이 아닌 View에 있는Resource 내용을 Clear하는 과정이다.

다음 설정은 바로 Rasterizer Stage 설정이다.

먼저 RSSetViewports와 RSSetScissorRects이다.

뷰포트와 시저랙트의 설정이고

RSSetViewports는 렌더링을 할 때의 Render Target의 영역을 설정해주는것이다.

Command List가 Reset 될 때마다 이 Viewport를 다시 Set해줘야 한다.

그걸 해주는 함수가 RSSetViewports이다

RSSetScissorRects는 렌더링 할 때 제거하지 않을 영역을 설정해준다.

해당 함수에 있는 사각형 외의 공간은 렌더링되지 않는다.

이 또한 Command List가 Reset 될 때마다 이 ScissorRect를 다시 Set해줘야 한다.

따라하기06(LabProject05) – 삼각형 그리기

CONSERVATIVE\_RASTERIZATION(보수적 레스터라이제이션)이란

다각형을 그릴 때, 그 다각형에 조금이라도 걸쳐져 있는 픽셀들을 전부 그릴것인지에 대한 여부를 설정하는 역할을 한다.

그리지 않는 것이 Rasterization의 속도적인 면에서 좋은 성능을 발휘할 수 있다.

하지만 보수적 레스터라이제이션을 사용한다면, 픽셀쉐이더에서 조명처리를 할 때 더 정확한 값을 얻을 수는 있다.

그림자 처리를 하기 전까지는 OFF로 설정해둔다.

Rasterizer 중 Viewport 설정에서 MinDepth, MaxDepth에 대한 설정을 ReMapping 할 수 있다.

그렇다면 Viewport에서 OM로 바뀐 픽셀이 넘어갔기에 다른 Depth값으로 Depth test를 할 수 있다.

또한 Viewport를 이용하여 UI를 만들 수 있다. 특정한 부분에다가만 랜더링을 하는 것으로, 특정위치에서만 랜더링을 하게 Viewport를 Set해주면 된다.

정점 버퍼를 사용하지 않고 랜더링 하기에 대해 설명하자면.

현재 따라하기 6번에 있는 내용을 참고하면 되는데,

먼저 IA(입력 조립기)라는 첫 파이프라인 단계에서는, 몇개의 정점을 사용해서 삼각형들을 그려라 라고 알려주면, 그래픽스 루트 시그너쳐도 Set하고, PipelineState도 Set했을 때, Scene을 Render하는 곳에

D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST라고하는 Primitive Topology를 Set하는 부분이 존재한다.

이는 시계방향의 Winding Order를 주면 알아서 그리게 해주는것이다.

Primitive Topology는 IA에서 정점들의 데이터를 가지고 어떤걸 조립할지를 알려준다.

예로 나온 것은 정점 3개를 주면 삼각형 한 개를 그리라고 하는 것이다.

그 값을 DrawInstanced에서 확인을 한다.

여기서 Shaders.hlsl내의 코드를 확인해보자면

float4 VSMain(uint nVertexID : SV\_VertexID) : SV\_Position

{

float4 output;

// Primitive(삼각형)를 구성하는 정점의 인덱스(SV\_VertexID)에 따라 정점을 반환한다.

// 정점의 위치 좌표는 변환이 된 좌표(SV\_Position)이다. 즉, 투영좌표계의 좌표이다.

if (nVertexID == 0)

output = float4(0.0, 0.5, 0.5, 1.0);

else if (nVertexID == 1)

output = float4(0.5, -0.5, 0.5, 1.0);

else if (nVertexID == 2)

output = float4(-0.5, -0.5, 0.5, 1.0);

return (output);

}

여기서 VS내에 존재하는 0에 해당하는 uint nVertexID의 값을 보내줄것이다.

이때의 VS가 보내는 좌표계는 Projection 좌표계이다.

Root Signature를 사용할 것인데, 왜 쓰는것인가 하면

Shader들을 Pipeline에 연결하기 위함이다.

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

해당 그림은 게임 오브젝트와 프레임워크의 상관관계를 나타낸다.

Game Object들은 Mesh에 대한 포인터와 Shader에 대한 포인터를 가지고 있다.

해당 내용을 이용하면 여러가지 오브젝트를 만들 수 있을것이다.

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

Shaders.hlsl은 프로젝트 빌드에서 제외를 해주어야 한다. 일반항목에서 빌드제외 항목을 “예”로 하여 컴파일 대상에서 제외된다.

다음으로 빌드를 할 때 Visual Studio에서 컴파일을 진행할 때,

HLSL컴파일러에는 진입점 이름이라고 하는 내가 만든 Shader의 함수 이름을 지정할 수 있는데 이것을 무조건 해 주어야 한다.

또한 최적화 사용에 대한 여부와 디버깅 정보 사용 여부도 존재한다. 이는 Release모드로 컴파일 할 때 설정해주어야 하는 항목이며

셰이더 형식과 셰이더 모델에 대한 설정도 무조건 해 주어야 한다.

예제 진행 에서는 꼭짓점 셰이더에 5.1로 지정해주였다.

이렇게 설정해주면 진입점 이름으로 된 CSO파일을 만들어준다. 또한 VS와 PS에 대한 파일을 따로 만들어줄 수 있는것이다.

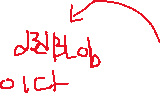
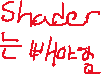
출력파일 항목에 보면 이를 확인해 줄 수 있다. 이걸 커스텀 해주면 내맘대로 여러 개를 만들 수도 있다.

이렇게 하면 이점은 나중가면 아주 많은 Shader 코드들이 생기는데 이를 하나하나 컴파일 할 필요 없이 바로 읽기만 하면 되니 속도가 아주 빨라지는 이점을 가질 수 있는것이다.

그 함수가 바로 ReadCompiledShaderFile이다

텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명



파일을 오픈하고(wfopen\_s)

파일의 시작부분으로 파일 포인터를 옮기고(fseek)

파일 전체 사이즈를 받아다가(ftell)

바이트를 할당하고(new BYTE)

파일전체 내용을 읽어들여서(fread, nFileSize)

내용을 저장한다(fread, pByteCode)

파일은 닫고(fclose)

그걸 블록으로 만들기 위해 d3dByteCode를 만들고

만약 ppd3dBlob을 만들었으면 거기다가 넣을거고

없으면 바이트코드와 바이트 수를 넘긴다면 Shader로 바이트 코드를 만들어내는것이다.

마치 C스타일로 파일을 오픈하는 것이다.

여기부턴 IA와 VS PS에 대해 살펴보고자 한다.

Resource의 모든 것

개념은 게임월드를 구성하고 그리기 위한 여러가지 형태(기하쉐이더, 텍스쳐, 쉐이더 데이터 등등)의 데이터이다.

GPU가 빠르게 접근하는 비디오 메모리에 저장되야 하고, 미리 작성된 파일형식

타입은 Buffer와 Texture로 나뉠 수 있다.

Buffer는 더 세부적으로 나누면, 정점 버퍼, 인덱스 버퍼, 상수 버퍼, 버퍼 이렇게 4가지로 나눌 수 있다.

방금 GPU에서 Shader를 읽고 쓴다고 했는데, 이게 엄청 빠르다.

근데 이 비디오메모리를 CPU의 응용프로그램도 읽거나 쓰는과정도 있겟지만 엄청 느리다.

그래서 CPU에서 비디오 메모리쪽으로 서로 움직이는 Resource의 양을 줄여야 한다.

위 작업을 사용하는 대표적인 함수가 Set함수들이다.

이렇게 단적으로 하는 방법이 삼각형 회전을 하는 따라하기 08번을 하면서 하는 DiffusedVertex관련한 것이다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

원래는 왼쪽과 같이 표현을 하였지만 오른쪽으로 표현하게 되는것이다.

이렇게 그리기 위해선 IA단계에서 정점, 인덱스 버퍼들이 IA에 연결이 되어있어야 하며, 그 IA가 넘겨주는 Primitive가 구성하는 정점들을 Transform작업을 수행해서 RS(Rasterization)에게 건네준다음 색칠하고 OM에게 건네주는 과정이 거쳐진다.

텍스트, 스크린샷, 디스플레이, 도표이(가) 표시된 사진

자동 생성된 설명

그러기 위해서는 IA(Input Assembly)단계에서 먼저 설정이 필요하다.

먼저 CreateCommittedResource() 함수를 호출해서 입력버퍼(정점, 인덱스) 객체를 생성한다. 리소스니까 메모리를 할당하기 위함이다. 그 만든 리소스에다가 정점 데이터, 인덱스 데이터를 집어넣는다

리소스를 만들었으니까 그거에 대한 버퍼의 뷰를 생성해야 한다. 그 함수는 D3D12\_VERTEX\_BUFFER\_VIEW 혹은 D3D12\_INDEX\_BUFFER\_VIEW라고 한다. 여기서 Vertex Buffer와 Index Buffer는 Descriptor Heap을 사용하지 않고도 View를 만들 수 있다. 그래서 방금 말한 저 두 구조체에의 내용에다가 방금 만든 Resource를 연결만 해도 View들을 만들 수 있다.

이렇게 해서 만든 정점 데이터에다가 정점 하나의 데이터를 입력 레이아웃으로 표현해줘야 한다. 그러므로 우리는 입력 레이아웃을 생성해야 하며 그 구조체는 D3D12\_INPUT\_ELEMENT\_DESC라고 한다.

그렇게 만든 입력 레이아웃을 Pipeline State에다가 위 그림처럼 연결을 해주며 그걸위한 것이 D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC::InputLayout이다. 그리하여 정점의 구조를 Pipeline이 알 수 있게 된다. 이걸 위해선 우리는 D3D12\_INPUT\_ELEMENT\_DESC와 D3D12\_INPUT\_LAYOUT\_DESC를 알아야 한다.

여기까지 한 다음, 이 정점버퍼들의 Primitive의 유형들, 선을 그릴지, 삼각형을 그릴지, 뭘 그릴 것인지에 대한 정보를 입력 조립기 단계에서 알려줘야 한다. 그것에 대한 함수가 IASetPrimitiveTopology()라는 함수이다.

그리고 실제 Buffer View들을 입력 조립기 단계에다가 Set을 해준다. 이때 쓰는 함수들이, IASetVertexBuffers() 와 IASetIndexBuffer()라는 함수이다.

여기까지 다 연결 한 다음 그리기 함수를 호출해주면 IA단계에서 Resource에 대한 내용을 읽고, 그것을 어떻게 처리할지에 대한 정보들도 읽어서 Vertex Shader단계(VS)에게 넘겨주게 된다. 이때 사용하는 함수가 DrawInstanced()와 DrawIndexedInstanced() 함수이다.

이처럼 랜더링, Graphics Pipeline이 시작되는 것은 위의 Draw함수로부터 시작이 되는것이다.

여기까지가 IA단계의 설정과정이였다

Resource라고 하는 것은 위에 나온 것 중 첫째로 하는 CreateCommittedResource로 Resource에 대한 인터페이스 포인터를 얻으면, 그 Resource들을 Map, Unmap이 가능하다.

여기서 Map이라는 함수의 기능은 Resource의 포인터를 넘겨주는 역할을 한다. 아까 알려줫듯 CPU와 GPU의 액세스 관계는 매우 번거롭다. 이 과정의 의미는 CPU 응용 프로그램에게 GPU의 비디오 메모리의 주소를 리턴해주는 것이다.

주소를 사용하고 있다면, 또다른 쓰레드가 이 리소스를 Map할 수 없기에, 얻어오자마자 Unmap을 해줘야지 다른 프로그램이 이 리소스의 접근을 할 수 있게 된다.

즉 Map과 Unmap은 쌍을 이루어서 사용되어야 한다. 흐름은 포인터가 필요해-> Map을 써서 준다-> 다썼다. -> 다른애들도 쓸 수 있게 Unmap. 이런 흐름이여야 한다.

Map의 파라메터는(서브리소스 인덱스, CPU가 접근할 메모리 영역, 서브리소스에 대한 포인터(시작주소))와 같은 형식이다. 여기서 NULL은 전체서브리소스를, CPU가상주소를 캐싱하는 의미로 사용된다.

Unmap의 파라메터는(서브리소스 인덱스, CPU가 수정한 메모리 영역)이다.

여기서 중요한 키 포인트는, Heap 타입에 따라 상황이 다르다라는 것이다.

Default힙일때는 Map과 Unmap을 사용할 수 없게 된다. 왜냐하면 Default Heap에 만들어진 Resource는 CPU의 접근을 막는 상태로 만들어지기 때문이다. 그래서 예제코드를 보면 Switch Case문으로 조건을 달고, Default일때는 Heap 타입을 Upload로 바꿔서 사용하는것이다. 해당 내용은 Map, Unmap혹은 Default를 검색해보면 확인할 수 있을것이다.

Upload는 CPU가 write하겟다고 하는것과 같은 뜻이다.

Readback은 CPU가 read하겟다고 하는 것이다.

근데 여기서 Readback은 Map을 하고 Unmap을 해줘야 하는 반면, Upload Heap은 Map만 하고 Unmap안해도 계속 사용할 수 있다.

또한 CopyResource라고 하는 Resource 전체를 복사하는 함수도 존재한다.

세부적인 파라메터를 보자면(복사한걸 담아둘 리소스(목표 리소스), 복사할 리소스(소스 리소스))와 같은 형식이고,

그 소스 리소스의 형식은 D3D12\_RESOURCE\_STATE\_COPY\_SOURCE의 형식이여야 하고

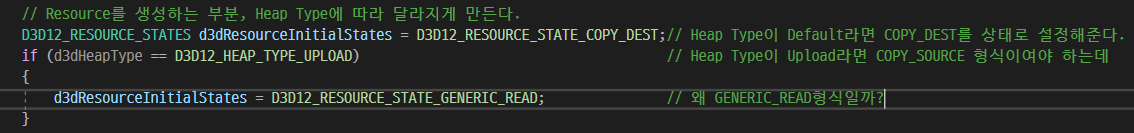
목표 리소스의 형식 또한 D3D12\_RESOURCE\_STATE\_COPY\_DEST와 같은 형식이여야 한다.

또한 두 리소스의 유형과 크기가 같아야 한다.

말고도 또 여러가지 리소스 복사 함수들이 존재한다.(8주차 2차시 18분쯤)

추가적으로 정점 버퍼(Vertex Buffer)는 생성할 때 Format이 무조건 UNKNOWN이여야만 한다.

의문점



왜일까?

Default Heap에다가 정점 버퍼를 만드는 경우에는, 반드시 Upload Heap이 필요하다. 그 Upload힙을 Copy명령을 한 다음에 Release를 해주면 안되기 때문에. 그 Upload Heap에 대한 인터페이스 포인터의 포인터로, 함수를 호출한 쪽의 인터페이스 포인터 값을 넘겨준다.

Copy명령이 다 된 것이 확인되고 Release를 해줘야 하기에 이러한 포인터의 포인터 형식이 되는것이다.

LabProject 08 – 직육면체 만들기와 카메라 위치에 대해

여기서 직육면체 큐브를 만들고 실행해본다면 카메라는 중앙에 존재하고, CullMode가 은면(뒷면)은 그리지 않아서 보이지 않는다.

그렇다면 카메라의 위치는 어디에 있는것일까?

m\_pCamera->GenerateViewMatrix(XMFLOAT3(0.0f, 0.0f, -2.0f), XMFLOAT3(0.0f, 0.0f, 0.0f), XMFLOAT3(0.0f, 1.0f, 0.0f));

GameFramework내에 존재하는 코드를 보면 카메라의 위치는 0, 0, -2에 존재한다.

그럼 직육면체 위치는 왜 정 중앙에 그려지는가?

그건 게임 객체가 월드변환행렬을 단위행렬로 초기화 하여서이다.

CCamera::CCamera()

{

m\_xmf4x4View = Matrix4x4::Identity(); // 단위 행렬로 초기화

m\_xmf4x4Projection = Matrix4x4::Identity(); // 단위 행렬로 초기화

m\_d3dViewport = { 0, 0, FRAME\_BUFFER\_WIDTH , FRAME\_BUFFER\_HEIGHT, 0.0f, 1.0f }; // Viewport 초기화

m\_d3dScissorRect = { 0, 0, FRAME\_BUFFER\_WIDTH , FRAME\_BUFFER\_HEIGHT }; // ScissorRect 초기화

}

Camera.cpp에 존재하는 코드를 보면 빨간색 칠한부분에 단위행렬로 초기화 하는 것을 볼 수 있다.

추가적으로 회전하는 이유는 // Rotate 부분을 확인하면된다.

직육면체를 확인하기 위해, 카메라의 위치를 위로 올리고 뒤로 빠지게 하기 위해 코드를,

m\_pCamera->GenerateViewMatrix(XMFLOAT3(0.0f, 12.0f, -25.0f), XMFLOAT3(0.0f, 0.0f, 0.0f), XMFLOAT3(0.0f, 1.0f, 0.0f));

위와 같이 바꾸어주었다.

여기서 이제 엄청 많은 오브젝트들을 만들 것이다.

원래 오브젝트는, 게임 객체(CGameObject)가 셰이더 객체(CShader)와 메쉬(CMesh)에 대한 포인터를 가지고 있는 구조이다

이에 대한 내용이

void CScene::BuildObjects(ID3D12Device\* pd3dDevice, ID3D12GraphicsCommandList\* pd3dCommandList)

{

// Graphics Root Signature를 만든다

m\_pd3dGraphicsRootSignature = CreateGraphicsRootSignature(pd3dDevice);

// 직육면체(Cube)

CCubeMeshDiffused\* pCubeMesh = new CCubeMeshDiffused(pd3dDevice, pd3dCommandList, 12.0f, 12.0f, 12.0f);

m\_nObjects = 1;

m\_ppObjects = new CGameObject \* [m\_nObjects];

// Object에 사용할 Mesh를 연결해주는 작업

CRotatingObject\* pRotatingObject = new CRotatingObject();

pRotatingObject->SetMesh(pCubeMesh);

// Object에 사용할 Shader를 연결해주는 작업

CDiffusedShader\* pShader = new CDiffusedShader();

pShader->CreateShader(pd3dDevice, m\_pd3dGraphicsRootSignature);

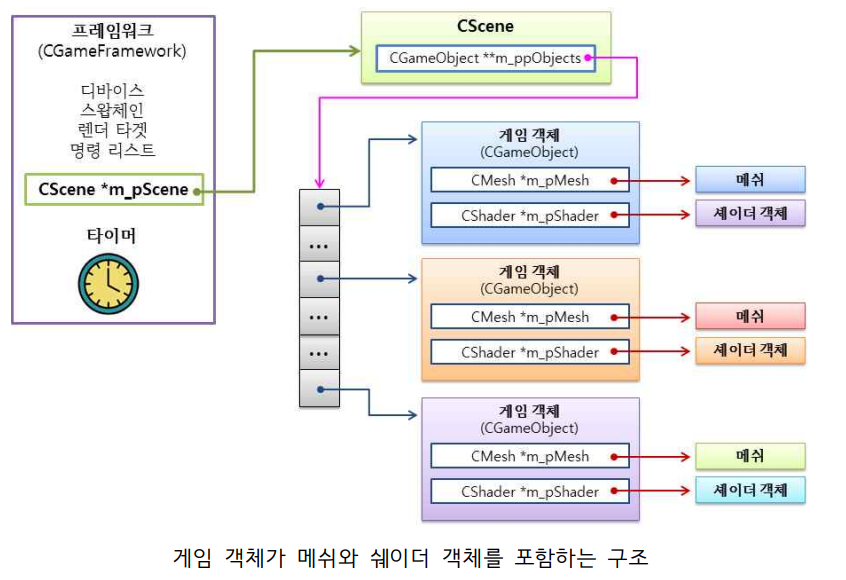
pShader->CreateShaderVariables(pd3dDevice, pd3dCommandList);

pRotatingObject->SetShader(pShader);

m\_ppObjects[0] = pRotatingObject;

}

위에 해당하는 내용이다.



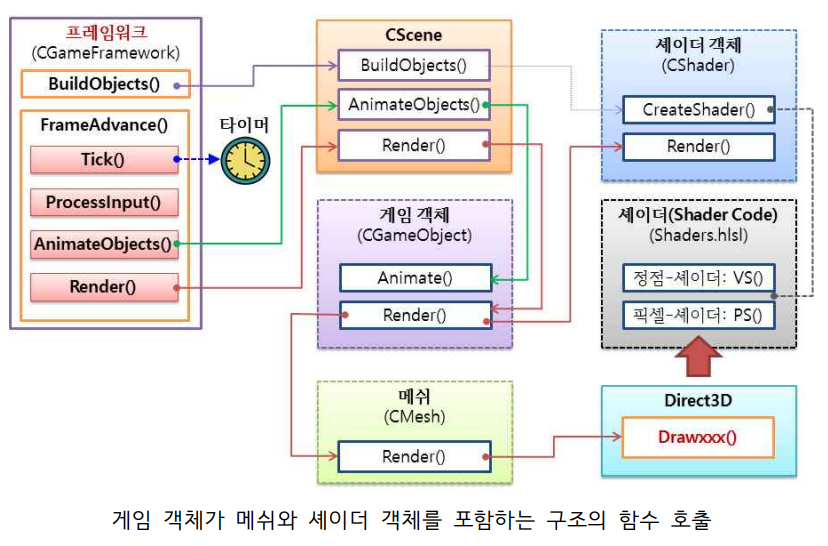
하지만 엄청 많은 오브젝트를 만들면 이러한 구조는

각 게임 객체를 렌더링하기 위해서 매번 Shader 객체를 렌더링(Shader 정보 - 정점 레이아웃, Vertex Shader, Pixel Shader설정)하고 Mesh 정보를 설정해야 한다.

이러한 구조에서는 각 객체를 렌더링하기 위하여 Direct3D 디바이스의 상태를 변경해야 한다.

Scene을 구성하는 객체가 아주 많은 경우 상태 변경을 위한 시간이 늘어날 수 있으므로 프로그램의 Framerate가 떨어질 수 있다.

그래서 구조를 바꾸고자 하는것이다.



Shader 객체가 게임 객체들의 배열을 가지고 있는 구조로 바꾸는 것이다. (Shader 객체가 Mesh에 대한 포인터를 가질 수도 있다).

그리고 Scene 객체는 Shader 객체들의 배열을 가지고 있으며 Scene을 렌더링하기 위해서 Shader 객체들을 순서대로 렌더링한다.

각 Shader 객체를 렌더링하기 위해서 Shader 정보를 한 번만 설정하고 게임 객체들의 배열에 포함된 모든 게임 객체들을 순서대로 렌더링하면 된다.

이러한 구조에서는 Scene을 렌더링하기 위하여 Shader 정보를 객체의 개수만큼 설정하지 않아도 된다. 결과적으로 같은 Shader를사용하여 렌더링하는 게임 객체들을 그룹화하기 때문에 Direct3D 디바이스의 상태 변화가 더 적게 될 수 있다. 이것으로 프로그램의 Framerate 향상될 수 있다.

즉 원래 Object(Mesh, Shader)의 구조를, Shader(Objects(Mesh))의 구조가 되게 하는것으로 요약할 수 있을 것 같다.

결국 Objects를 Object로 보느냐, 아니면 Shader로 보느냐라는 구성과 관점의 차이인것이다.

해당 내용은

LabProject9와 LabProject10을 비교해본다면 오브젝트를 랜더링 하는 방법의 차이를 확인할 수 있다.

버그가 있었는데 원인을 발견하였다.

1. Object.h에는 Shader.cpp와 Shader.h에서 사용하는 함수를 사용하기 위해 "Shader.h"라는 파일을 inclued 한 상태이다.

2. 그런데 여기서 위에서 언급했던 CGameObject\*\* m\_ppObjects = NULL; 부분이 오류가 발생하였다.

3. 해결방법은 Object.h에 include되었던 "Shader.h"를 Object.cpp에다가 include해서 정상적으로 작동하였다.

여기서 왜 이러한 오류가 발생하였나를 생각해보았는데.

사이클에서 문제가 발생하였다고 생각이 든다.

좀 찾아보니 이와 같은 현상을 “순환 include 오류”라고 한다.

그리고 LabProject를 만들면서 아주 많은 의문점이 발생하였다.

//m\_nSwapChainBufferIndex = m\_pdxgiSwapChain->GetCurrentBackBufferIndex();

// 이건 왜인지 모르겟는데 LabProject11에서 사라졌다.

//얘내는 안쓰는데 왜만듬?

void CPlayer::Move(ULONG nDirection, float fDistance, bool bVelocity)

{

}

void CPlayer::Move(const XMFLOAT3& xmf3Shift, bool bUpdateVelocity)

{

// bUpdateVelocity가 참이면

if (bUpdateVelocity)

{

// 플레이어를 이동하지 않고 속도 벡터를 변경한다.

m\_xmf3Velocity = Vector3::Add(m\_xmf3Velocity, xmf3Shift); // 플레이어의 속도 벡터를 xmf3Shift 벡터만큼 변경한다.

}

else

{

//플레이어를 현재 위치 벡터에서 xmf3Shift 벡터만큼 이동한다.

m\_xmf3Position = Vector3::Add(m\_xmf3Position, xmf3Shift);

//플레이어의 위치가 변경되었으므로 카메라의 위치도 xmf3Shift 벡터만큼 이동한다.

if (m\_pCamera) m\_pCamera->Move(xmf3Shift);

}

}

//얘내는 안쓰는데 왜 만듬?

void CPlayer::Move(float fxOffset, float fyOffset, float fzOffset)

{

}

또한 플레이어가 시원찮게 움직여서 확인해보니

//Move(xmf3Velocity, false); // 요거 잘못됐음

Move(m\_xmf3Velocity, false);

교수님의 샘플 코드에 오류가 있었다.

CCubeMeshDiffused::CCubeMeshDiffused(

ID3D12Device\* pd3dDevice,

ID3D12GraphicsCommandList\* pd3dCommandList,

float fWidth, float fHeight, float fDepth) : CMesh(pd3dDevice, pd3dCommandList)

{

m\_nVertices = 8;

m\_nStride = sizeof(CDiffusedVertex);

// m\_nOffset = 0; // 얘내는 왜 주석처리하는지는 모르겟다.

// m\_nSlot = 0; // 얘내는 왜 주석처리하는지는 모르겟다.

m\_d3dPrimitiveTopology = D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST;

저기도 왜 주석처리를 진행하는지 모르겟다

Instancing을 위한 LabProject12의 내용 CreateShaderVariables의 내용 중

// Vertex Buffer(Upload Heap)에 대한 포인터를 저장한다.

m\_pd3dcbGameObjects->Map(0, NULL, (void\*\*)&m\_pd3dcbGameObjects);

// Vertex Buffer 에 대한 View를 생성

m\_d3dInstancingBufferView.BufferLocation = m\_pd3dcbGameObjects->GetGPUVirtualAddress();

m\_d3dInstancingBufferView.StrideInBytes = sizeof(VS\_VB\_INSTANCE);

m\_d3dInstancingBufferView.SizeInBytes = sizeof(VS\_VB\_INSTANCE) \* m\_nObjects;

// 여기 왜 UnMap안해주지?

에 대한 내용이다. Map을 해주었으면 unmap을 해줘야 하는 것 아닌가? 데이터의 내용이 바뀔 위험이 없는 놈이라서 Unmap해줄 필요가 없는것인가?

25분 20초

할일 리스트, 1차 과제 시작, 최종 완료, 2복습 시작

과제 계획

1. 모델 불러오는 방법 불러서 존나 많은 네모 바꾸기

이 부분은 모델 로드 예제를 사용하자

1. 클릭 구현해서 오브젝트 소멸자와 연결하기

클릭구현은 따라하기의 17을 이용해서 만들고

오브젝트 소멸자와 연결은 옛날에 만들었던 탱크관련해서 확인해보자

1. 밉맵과 카메라를 연결해서 UI만들기

이건…..좀 많이 고생을 해봐야 할것이다.

여기까지만 하면 1차과제가 끝난다.

3D게임프로그래밍2에 대한 내용 정리본(1차 정리, 정갈하게 정리 안했음)

테셀과 기하 장단점 - 기하는 반복작업, 테셀은 로우폴리곤을 하이폴리곤으로 하는 작업

공통점은 프리미티브를 주면, 그 프리미티브들에 정점들을 추가해서 새로운 프리미티브들을 생성하는 것이다

차이점은 기하쉐이더는 쉐이더 단계로만 이루어져있어서, 정점추가나 어떻게 할지를 쉐이더에서 모든걸 처리한다.

테셀레이션은 파이프라인 함수로 이루어져있으니, 테셀레이션에서 쪼개지는 방법이 이미 정해져있다.

또 기하쉐이더는 추가할 정점의 개수에 제한이 걸려있지만, 테셀레이션은 아주 많은 정점을 생성해낼 수 있다는 차이점 또한 존재한다.

이러한 기하쉐이더의 장점으로는 쉐이더에 넘어가는 데이터의 양을 줄이는 이점이 있으며

이러한 테셀레이션의 장점은 그릴때만 적용이 되어서 로우폴리곤으로 충돌검사가 가능하다.

투명한 무언가(유리판같은거)를 그리기 위한 과정 - 먼저 블랜딩을 할 때는 Depth Test를 하면 안 된다.

즉 랜더타겟에 불투명한 색들을 먼저 그리고 그 다음에 투명한 오브젝트를 그리면서 블랜딩 작업을 해야 한다.

즉 멀리있는것부터 순서대로 그려야만 한다. 투명한건 그리는 순서를 정해줘야 한다.

그럼 어떻게 순서를 결정해주는가? 카메라의 거리에 따라 정렬을 한 다음 멀리있는 것부터 그리면 된다. 불투명한걸 다 그린다음에 투명한걸 랜더링 한다.

노멀매핑에 필요한 정보 – 법선맵에 대한 이미지로부터 픽셀에 따른 서로다른 법선정보가 필요하다. 또한 표면의 높이를 가진 높이맵을 읽어서 높이에 대한 정보도 필요할것이다. 더 나아가 텍스쳐가 적용되는 메쉬마다의 노멀벡터의 정보또한 필요하다. 왜냐하면 모든 메쉬에 모든 노멀을 적용할 수 없기때문이다.

뎁스와 스텐실 관계 – Depth는 깊이를 판단해서 화면에 최종적으로 그려지는 픽셀의 색을 결정결정하는 것, Stencil은 Render Target의 일부 영역을 랜더링 하지 않도록 설정하는 것이다. Depth를 성공해야하며, Stencil이 있을 때 그것도 성공해야 랜더링이 가능하다.

스텐실 검사가 종료되고 그 스텐실 값을 새로운 값으로 바꿔줘야 된다.(마치 움직임을 처리하는것과 같이)

Depth와 Stencil 검사결과가 True라면 깊이 버퍼가 갱신이 된다.

Stencil이 활성화가 되면 스탠실 버퍼가 갱신이 된다.

딱 이 두개의 조건으로 모든 상황이 커버가 된다.

거울 만드는 방법 - …

GPU 기반 파티클 시스템을 만드는 과정 – 파티클의 생성과소멸, 위치와방향을 처리해야 된다. 정점버퍼에 초기파티클 정보를 건네주고, 파이프라인과정에서 기하쉐이더에서 또다른 버퍼에 새로운 파티클들을 만들어서 그걸 입력조립기에 다시 집어넣는 과정을 거친다.

투영 텍스쳐 매핑 문제점3가지 – 게임 오브젝트에 해당하는 월드좌표계의 점을 변환하여서 텍스쳐의 UV좌표로 변환하는 것이다. 더 자세하게 말하면 흑백 텍스쳐그림을 오브젝트에 붙여버리는거다.

1. 빛의 특성상 상하좌우가 뒤바뀌며 원치 않는 곳에서 투영이 될것이다. 그래서 Pw가 음수일때는 투영을 하면 안되고
2. 은면도 투영이 되버리기 때문에 벡터를 확인해서 투영여부를 결정하는 것도 필요하다(p-v)\*n<0
3. 또한 투영 텍스쳐와 투영할 오브젝트 사이에 가려지는 상황에는 투영하지 말아야 한다. 이때도 Depth를 비교해서 그릴지여부를 판단해야된다.

원형 그림자와 평면 투영 그림자, 투영 텍스쳐 매핑과, 그림자 매핑의 차이를 설명하시오 –

1. 원형 그림자 - 메시라는 개념 없이 네모 하나 만들고 인스턴스 데이터로 포지션 벡터를 넘겨서 그 포지션에다가 그리는 형식이다.
2. 평면 투영 그림자 – 그림자가 투영되는 곳이 평면일 때 사용한다. XMMatrixShadow라는 함수이고 이때 방향성광원과 점광원을 구분하기위해 w부분의 값을 각각 0, 1로 표현해준다.
3. 투영 텍스쳐 매핑 - 게임 오브젝트에 해당하는 월드좌표계의 점을 변환하여서 텍스쳐의 UV좌표로 변환하는 것이다. 더 자세하게 말하면 흑백 텍스쳐그림을 오브젝트에 붙여버리는거다.

빛의 특성상 상하좌우가 뒤바뀌며 원치 않는 곳에서 투영이 될것이다. 그래서 Pw가 음수일때는 투영을 하면 안되고

은면도 투영이 되버리기 때문에 벡터를 확인해서 투영여부를 결정하는 것도 필요하다(p-v)\*n<0

또한 투영 텍스쳐와 투영할 오브젝트 사이에 가려지는 상황에는 투영하지 말아야 한다. 이때도 Depth를 비교해서 그릴지여부를 판단해야된다.

1. 그림자 매핑(여드름, 바이어스) – 여기서부터 평면에 그림자를 그리는 것이 아닌 어떠한 면에서도 그림자를 그리는것이다. 조명을 새로운 카메라라고 생각하고 그 조명카메라의 Rendering정보를 버퍼에 저장해서, 그림자가 있는가? 없는가에 대한 값을 저장해서 활용하는것이다. 이때 사용하는 함수가 있다. XMMATRIX XMMatrixPerspectiveForLH라고 하는 카메라 좌표계에 대한 원근투영변환행렬을 만들어주는 함수를 사용한다.(직교투영일때는 오쏘그래픽)
2. 그림자 여드름 - 이때 그림자 매핑에서 문제가 발생하는데. 그림자 맵의 해상도가 낮으면 그림자 여드름이 발생하게 된다. 이를 해결하려면 레스터라이저 설정에서 DepthBias에 해당하는 설정 3개를 해줘야 된다. 근데 바이어스를 키워주면 실제 그림자 투영거리보다 멀어져서 그림자가 점점 작아지는 현상이 발생한다. 상태 설정에는 픽셀에 더해질 바이어스 값과, 최대 바이어스 값과 기울기에 따른 바이어스 조절이 있다.

텍스쳐 노말 변위맵핑의 정의 – 텍스쳐 매핑은 텍스쳐에 대한 이미지 데이터를 메모리나 파일로 읽어서 메쉬에 이미지를 그리는 것이다. 이때 이미지에 따른 빛의 반사는 존재하지 않는다.

노말맵은 법선맵이라고 하며, 텍스쳐 이미지에서 노멀값, 즉 빛의 굴곡을 표현하기 위해 따로 만들어진 이미지 데이터이다.

변위맵은 각각의 픽셀을 높낮이로 저장해둔 것이며, 실제 텍스쳐에 높고 낮음의 처리를 하는 것이다.

위 세가지 과정은 테셀레이션과 기하 쉐이더 과정에서 다룬다. 기하쉐이더에선 추가된 정점을 높낮이를 설정하는것이고, 테셀레이션에선 곡면을 표현하는 단계를 만들 수 있다.

테셀레이션의 과정 – 프리미티브 정보 건네줘서, 어떤 도형(토폴로지)를 어떤 크기로 자를지를설정, 테셀레이션이 잘라줌, 도메인에서 CONTROL\_POINT\_PATCHLIST의 형식으로 건네줘야 한다.

간단 목차

텍스쳐

1. 블록 압축, DDS
2. 텍스쳐 배열, 쉐이더 리소스 뷰 & 루트 시그너쳐
3. 텍스처 필터링
4. 지형 텍스쳐 맵핑
5. 빌보드, 스카이박스
6. 후처리(Post Processing)
7. 샘플러 객체

블랜딩

1. 텍스쳐 블랜딩
2. 알파 블렌딩
3. OM 블렌딩
4. 색상 블렌딩
5. 다중샘플링, 알파 커버리지 다중 샘플링
6. 텍스쳐 스플래팅
7. 렌더링 순서(Depth)
8. 안개
9. 스텐실 검사
10. Depth Stencil 검사
11. 평면 거울(스텐실 표현)

매핑

1. 법선 매핑
2. 변위 매핑
3. 시차 매핑
4. 조명 처리
5. 지연 쉐이딩

기하 쉐이더

기하 쉐이더는 프리미티브를 구성하는 정점들의 배열형식의 입력을 받아서 그거에 대한 추가적인 정점을 만드는 쉐이더 단계이다. 더 나아가 그 입력받은 정점들을 프리미티브들의 출력을 레스터라이저에게 넘기는 함수이다

1. 아주 많은 나무 그리기
2. 스트림 출력 객체
3. GPU기반 파티클 시스템

테셀레이션

테셀레이션을 사용하는 가장 큰 이유는 어떠한 메쉬를 로우 폴리곤이여서 사각형들의 집합으로 되어 있는 것들을 하이 폴리곤화 시키는 것, 곡면화 시키는 작업을 위해 사용한다.

1. 헐 쉐이더
2. 테셀레이션 단계
3. 도메인 쉐이더
4. 베지어 곡면

그림자

1. 원형 그림자 - 메시라는 개념 없이 네모 하나 만들고 인스턴스 데이터로 포지션 벡터를 넘겨서 그 포지션에다가 그리는 형식이다.
2. 평면 투영 그림자 - XMMatrixShadow라는 함수이고 이때 방향성광원과 점광원을 구분하기위해 w부분의 값을 각각 0, 1로 표현해준다.
3. 투영 텍스쳐 매핑 - 게임 오브젝트에 해당하는 월드좌표계의 점을 변환하여서 텍스쳐의 UV좌표로 변환하는 것이다. 더 자세하게 말하면 흑백 텍스쳐그림을 오브젝트에 붙여버리는거다.

빛의 특성상 상하좌우가 뒤바뀌며 원치 않는 곳에서 투영이 될것이다. 그래서 Pw가 음수일때는 투영을 하면 안되고

은면도 투영이 되버리기 때문에 벡터를 확인해서 투영여부를 결정하는 것도 필요하다(p-v)\*n<0

또한 투영 텍스쳐와 투영할 오브젝트 사이에 가려지는 상황에는 투영하지 말아야 한다. 이때도 Depth를 비교해서 그릴지여부를 판단해야된다.

1. 그림자 매핑(여드름, 바이어스) - 조명을 새로운 카메라라고 생각하고 그 조명카메라의 Rendering정보를 버퍼에 저장해서, 그림자가 있는가? 없는가에 대한 값을 저장해서 활용하는것이다. 이때 사용하는 함수가 있다. XMMATRIX XMMatrixPerspectiveForLH라고 하는 카메라 좌표계에 대한 원근투영변환행렬을 만들어주는 함수를 사용한다.(직교투영일때는 오쏘그래픽)
2. 그림자 여드름 - 이때 그림자 매핑에서 문제가 발생하는데. 그림자 맵의 해상도가 낮으면 그림자 여드름이 발생하게 된다. 이를 해결하려면 레스터라이저 설정에서 DepthBias에 해당하는 설정 3개를 해줘야 된다. 근데 바이어스를 키워주면 실제 그림자 투영거리보다 멀어져서 그림자가 점점 작아지는 현상이 발생한다. 상태 설정에는 픽셀에 더해질 바이어스 값과, 최대 바이어스 값과 기울기에 따른 바이어스 조절이 있다.

기 하 쉐 이 더

기하 쉐이더는 프리미티브를 구성하는 정점들의 배열형식의 입력을 받아서 그거에 대한 추가적인 정점을 만드는 쉐이더 단계이다. 더 나아가 그 입력받은 정점들을 프리미티브들의 출력을 레스터라이저에게 넘기는 함수이다.

그리고 그 프리미티브가 몇번째 프리미티브인지에 대한 정보도 사용할 수 있다.

그래픽스 파이프라인 스테이트에서 버텍스 쉐이더나 픽셀 쉐이더를 연결하는 방법과 큰 차이가 없다라고 한다.

활용 예시 - 빌보드 나무 많이 그리기

빌보드 사각형의 포지션벡터 – 카메라의 포지션벡터를 해서 룩벡터를 구하고, 그걸로 빌보드 사각형을 만들어서 그걸 많이 만들면 숲이 만들어지는데

기하 쉐이더를 사용하면 포지션의 정보와 포지션 벡터만 있다면, 메쉬의 정점의 위치를 폭과 높이로 알아낼수있다. 이렇게하면 Shader에 넘어가는 정보의 양을 많이 줄여서 일일히 배열로 넘기는거보다는 효율적일것이다.

GSInstanceID는 실행되는 기하쉐이더의 인스턴스이고

어떤 렌더타겟, 어떤뷰포트로 이동할지에 대한 값을 지정하는 SV\_RTAI(렌더타겟어레이인덱스), SV\_VAI(뷰포트어레이인덱스)라는 시멘틱이 존재한다.

124944 18분….

테 셀 레 이 션

프리미티브->도메인

도메인은 도형을 일정한 크기로 쪼개는 것

테셀레이션을 사용하는 가장 큰 이유는 어떠한 메쉬를 로우 폴리곤이여서 사각형들의 집합으로 되어 있는 것들을 하이 폴리곤화 시키는 것, 곡면화 시키는 작업을 위해 사용한다.

그래서 프리미티브는 실제 정점 정보이지만, 도메인은 논리적인 정보이다.

즉 처음 입력조립기에 정점을 넣을 때 더 적은 데이터로 복잡한 구조를 만들 수 있는것이다.

헐 쉐이더

제어점들의 집합을 입력받고, 패치상수데이터를 출력한다.

테셀레이션 단계

간단하게 말하면 어떤 도형을 어떤 크기로 자를지를 받아서 그대로 자르는 과정이다. 여긴 내가 건들 부분이 아닌 글카가 알아서 하는 것

도메인 쉐이더

그 짜른 도메인 점들을 도메인쉐이더가 실행해서 SV\_Position이라는 시멘틱을 포함한 쉐이더 정보를 건네줘야 할것이다.

원래는 입렵조립기에서 프리미티브 토폴로지 트라이앵글 같은 과정을 했는데, 이젠 컨트롤 포인트 패치리스트의 형식으로 건네줘야 하는 것이 된다.

버텍스 쉐이더가 하는 작업과 같이 수행해서 레스터라이저한테 넘겨줘야 한다.

베지어

그 림 자

원형 그림자

텍스쳐를 읽어서 매핑하는 방식이다.

메시라는 개념 없이 네모 하나 만들고 인스턴스 데이터로 포지션 벡터를 넘겨서 그 포지션에다가 그리는 형식이다.

또한 파이프라인 스테이트에서 D3D12\_BLEND\_DESC의 설정에서 블렌드 관련 설정을 Set해주면 된다.

여기서 소스 블렌드를 ZERO, 데스티네이션 블렌드를 소스블렌드의 반대색상으로 설정하면 휜색 부분만 그려지는 형식으로 만들 수 있다.

월드 좌표계의 포지션을 읽어서 월드의 포지션을 기준으로 올리거나 내리는 작업을 하면 간단한 원형 그림자를 만들 수 있다.

뭐 굳이 원형이 아니여도, 활용에 따라 그림자의 모양을 미리 만들어서 시점에 따른 그림자를 불러들이는 방식도 가능할것이다.

근데? 이렇게 만들면 평면 그림자라서 굴곡진 바닥이거나 Y를 제대로 설정 안하면 그림자가 공중에 뜨거나 짤리거나 하는 문제가 발생하기도 한다.

교수님께서는 빌보드의 개념이기도 하다셨는데, 이건 싱글 1인칭 게임일 때만 가능한 말이고

3인칭이거나 다인용 게임일경우에는 빌보드로 했다간 다른사람 그림자가 이상하게 렌더링 될 것이라고 생각된다. 아 그냥 움직임 없는 오브젝트는 가능하고 플레이어끼리는 불가능할것같다.

평면 투영 그림자

그렇다면 그림자에 방향이 생긴다면?

해서 나오는게 평면 투영 그림자이다.

조명과 그림자, 평면은 같지만, 메쉬의 모양에 따라 그림자의 모양이 달라진다.

메쉬의 정점들과 빛의 직선 방정식을 이용해서 면에 투영하고, 이어서 그 속을 검게 만들어서 그림자를 만드는거다.

그러기 위해선 오브젝트를 지나는 광선의 도착점의 위치를 찾아야 하는데.

그 평면의 방정식과 광선의 직선 방정식이 만나는 점이니 둘의 방정식의 공통해를 찾는 과정을 가져야 한다. 이때 계산을 위해 모델좌표계로 만들어서 계산을 진행한다.(좌표계 변환은 변환행렬의 역행렬을 곱해주면 되니까 큰 어려움은 없다)

여기서 케이스가 갈리는데, 방향성 광원이냐, 점광원이냐에 따라서 좀 생각이 달라진다

방향성 광원일때는 그냥 직선의 방정식을 쓰는 반면, 점광원일때는 빛의 점 위치와 오브젝트의 직선의 방정식을 사용하여 계산하는 차이가 있다.

정점 Q를 위 식에 대입해서 그리면되는거지만…. 그말은 Q에서 변환을 하면 만들어지는게 Q’인 그림자의 정점데이터들이라는거다. 그걸 행렬로 표현해서 써먹자는 것이다.

근데 행렬로 만들 때 방향성 광원이랑 점광원이랑 최종적인 t계산식이 다르다.

그래서 이걸 통합하려면 조명벡터 L을 -L로 생각해서(그림자가 안그려지니까) 하나의 모양으로 합친다.

그걸 지원해주는 함수가 DirextX함수로 존재한다. XMMatrixShadow라는 함수이고 이때 방향성광원과 점광원을 구분하기위해 w부분의 값을 각각 0, 1로 표현해준다.

빛에 대한 값을 모델좌표계로 표현하고, 모든 오브젝트에 대해 루프처리로 전부 위의 함수를 적용시켜서 그림자를 그리라고 하면 된다(이때 색 쉐이더는 따로 그레이스케일로 만들어 준다)

투영 텍스쳐 매핑

지금까지 한건 평면에 그려지는 그림자를 표현하는 것이다.

근데 실제 세계는 투영되는 부분이 아무리 꾸겨져도 투영은 변하지 않는다.

그러니 여기서 절두체를 활용하는것이다.

게임 오브젝트에 해당하는 월드좌표계의 점을 변환하여서 텍스쳐의 UV좌표로 변환하는 것이다.

더 자세하게 말하면 흑백 텍스쳐그림을 오브젝트에 붙여버리는거다.

모델에따가 텍스쳐를 매핑하는 과정을 좀 더 자세하게 살펴보자면

조명에서 쏘는 광선에서 텍스쳐를 뚫고 오브젝트에 맞는 그 과정을 절두체로 투영하는 것이다.

이 과정에서 절두체를 쓴다는건, 조명을 카메라좌표계마냥 보겟다는 거니까.

월드좌표계에 해당하는 점은 투영좌표계의 특정 부분이 대체되는것이고, 이 투영좌표계를 텍스쳐좌표계로 바꾸어야 한다.

그 텍스쳐로 변환하는 과정은 0.5를 곱하고 0.5를 더하는 과정이 필요하다.

이렇게 하면 그림이 자연스럽게 메쉬에 따라 굴곡지게 그려지게 된다. 즉 이제 이미지를 동적으로 만들어둘 필요가 없다(위에 방법)

물체 사람 조명 순으로 진행될 때, 사람과 물체의 조명에대한 Depth를 진행해서, 갱신되면 그림자가 있다는 거니까. 갱신된 여부를 0과 1로 표현하여 배열형식으로 버퍼에 저장해서 그림자 여부를 판단할 수 있게 되었다.

하지만 문제점이 있다.

빛의 특성상 상하좌우가 뒤바뀌며 원치 않는 곳에서 투영이 될것이다. 그래서 Pw가 음수일때는 투영을 하면 안되고.

은면도 투영이 되버리기 때문에 벡터를 확인해서 투영여부를 결정하는 것도 필요하다(p-v)\*n<0

또한 투영 텍스쳐와 투영할 오브젝트 사이에 가려지는 상황에는(차폐) 투영하지 말아야 한다. 이때도 Depth를 비교해서 그릴지여부를 판단해야된다.

그림자 매핑

간단히 설명하면 투영 텍스쳐 매핑에서 오브젝트를 매핑하는거다.

조명을 새로운 카메라라고 생각하고 그 조명카메라의 Rendering정보를 버퍼에 저장해서, 그림자가 있는가? 없는가에 대한 값을 저장해서 활용하는것이다.

이때 사용하는 함수가 있다. XMMATRIX XMMatrixPerspectiveForLH라고 하는 카메라 좌표계에 대한 원근투영변환행렬을 만들어주는 함수를 사용한다.

여기서 조명이 많아지면 시간이 엄청 많이 걸린다.

이때 사용하는 것이 휴리스틱이다.

픽셀에 대한 조명을 계산할 때 밝은것부터 대략적인 계산을 한다.

이때 그림자 매핑에서 문제가 발생하는데. 그림자 맵의 해상도가 낮으면 그림자 여드름이 발생하게 된다. 이를 해결하려면 레스터라이저 설정에서 DepthBias에 해당하는 설정 3개를 해줘야 된다.

근데 바이어스를 키워주면 실제 그림자 투영거리보다 멀어져서 그림자가 점점 작아지는 현상이 발생한다. 상태 설정에는 픽셀에 더해질 바이어스 값과, 최대 바이어스 값과 기울기에 따른 바이어스 조절이 있다.

Block Compression(블록 압축)

쓰는 이유는 모델 데이터를 압축할라고 – 메모리 효율 때문에

이 블록 압축을 사용하면 바이트 수가 기존 사용 전보다 1/8정도 줄게 된다.

압축 형식은 DXGI\_FORMAT\_BC1\_UNORM이라는 형식을 사용하며

말고도 BC2, BC3이 있는데 숫자가 높아질수록 복잡한 방식과 디테일한 압축을 보여주며 압축률이 떨어진다. 숫자가 높아질수록 알파가 높아진다.

DXGI\_FORMAT\_BC1\_UNORM을 자세히 설명하면 알파를 고려 안하거나 알파를 투명Or불투명으로 판단하여 압축하는 방식이다.

BC2나 BC3를 알파를 4비트나 8비트로 따로 쓰는것이다. 즉 알파 정도를 설정해주는것이며 색 설정은 같다. 알파를 디테일하게 표현하는 것은 DXGI\_FORMAT\_BC3\_UNORM를 대표적으로 사용한다.

그래서 교수님은 BC1을 써서 압축률 최대로 만드셨다.

아주 적은 메모리를 써서 할 수 있다.

이는 다이렉트 X가 지원하는 그래픽카드 내부에서 압축하도록 지원을 한다.

압축을 해제하는건 크게 시간이 걸리지 않기 때문이다.

해당 코드의 내용을 보려면 당장 교수님 샘플프로젝트 7-9-3에서 위 형식을 검색하면 DDSTextureLoader파일 내에서 확인할 수 있다.

이러한 블록 압축 기능을 사용하려면 모든 이미지 파일의 형식을 “DDS”로 바꾸어야 할것이다.

Visual Studio에서 파일->새로만들기->그래픽->dds형식으로 만들고 거따가 이미지 집어넣으니까 잘 되었었다.

근데 밉맵은 2의 n승 형식이 아니여도 사용가능하지만 블록 압축은 무조건 2의 n승의 크기여야만 한다.

아니면 DirectXTex-Master라는 Github사이트 가서 텍스쳐에 관련된 것을 마소에서 올려놨다.

DDS텍스쳐 로더랑 WIC로더라고 jpg나 png파일들을 불러들이는 함수를 제공하는 함수이다.

DDSTexture의 세부 내용을 보자면

LoadDDSTextureFromMemory은 메모리에 있는걸 읽어오는것이고

LoadDDSTextureFromFile이라는 함수를 이용해서 파일에서 DDS텍스쳐를 로드 하여서 사용하여 텍스처를 넣을 수 있으며 이것을 주로 호출할것이다.

HRESULT DirectX::LoadDDSTextureFromFileEx(

ID3D12Device\* d3dDevice,

const wchar\_t\* fileName,

size\_t maxsize,

D3D12\_RESOURCE\_FLAGS resFlags,

unsigned int loadFlags,

ID3D12Resource\*\* texture,

std::unique\_ptr<uint8\_t[]>& ddsData,

std::vector<D3D12\_SUBRESOURCE\_DATA>& subresources,

DDS\_ALPHA\_MODE\* alphaMode,

bool\* isCubeMap)

들어간 파라메터를 보면

파일 이름과 파일 크기, 플래그 등등있고, 리소스에대한 인터페이스 포인터의 포인터를 인자로 받는다. 그 뜻은 이 파일을 받아서 비디오 메모리를 할당에서 인터페이스포인터로 리턴을 하는 것이다.

그 리소스에 대한 정보는 ddsData, subresources, isCubeMap에 저장되서 들어간다.

그래서 stdafx.cpp파일 내에서 CreateTextureResourceFromDDSFile라는 함수를 호출해서 실제 리소스를 만들어서 사용하도록 함수를 만들거나 Eclass에 올라온 샘플 프로젝트를 이용하면 된다.(LabProject 8-0에 존재한다.)

아래는 위에서 언급한 DDS함수 호출 함수이다.

ID3D12Resource \*CreateTextureResourceFromDDSFile(ID3D12Device \*pd3dDevice, ID3D12GraphicsCommandList \*pd3dCommandList, wchar\_t \*pszFileName, ID3D12Resource \*\*ppd3dUploadBuffer, D3D12\_RESOURCE\_STATES d3dResourceStates)

{

ID3D12Resource \*pd3dTexture = NULL;

std::unique\_ptr<uint8\_t[]> ddsData;

std::vector<D3D12\_SUBRESOURCE\_DATA> vSubresources;

DDS\_ALPHA\_MODE ddsAlphaMode = DDS\_ALPHA\_MODE\_UNKNOWN;

bool bIsCubeMap = false;

// 여기가 바로 아까 말한 DDS함수이다.

HRESULT hResult = DirectX::LoadDDSTextureFromFileEx(pd3dDevice, pszFileName, 0, D3D12\_RESOURCE\_FLAG\_NONE, DDS\_LOADER\_DEFAULT, &pd3dTexture, ddsData, vSubresources, &ddsAlphaMode, &bIsCubeMap);

D3D12\_HEAP\_PROPERTIES d3dHeapPropertiesDesc;

::ZeroMemory(&d3dHeapPropertiesDesc, sizeof(D3D12\_HEAP\_PROPERTIES));

d3dHeapPropertiesDesc.Type = D3D12\_HEAP\_TYPE\_UPLOAD;

d3dHeapPropertiesDesc.CPUPageProperty = D3D12\_CPU\_PAGE\_PROPERTY\_UNKNOWN;

d3dHeapPropertiesDesc.MemoryPoolPreference = D3D12\_MEMORY\_POOL\_UNKNOWN;

d3dHeapPropertiesDesc.CreationNodeMask = 1;

d3dHeapPropertiesDesc.VisibleNodeMask = 1;

여기서 Texture는 데이터를 카피해서 넣기위해선 Upload Buffer가 필요하다. 그래서 코드 진행을 보자면 먼저 Upload Buffer로 만드는 과정을 하는 것을 볼 수 있다. 나머지 끝까지 만드는 내용은 해당 코드를 참고하도록 하자.

그렇게 되면 DDS에서 읽은 파일의 내용이 온전히 Buffer에 저장되게 된다.

그 다음에 Resource State를 Transition하는 과정을 거치면 끝나게 된다.

이 과정을 함으로 GPU에서 사용이 가능한 상태로 전환이 이루어지게 된다.

아래는 해당 코드의 예제코드이다.

::UpdateSubresources(pd3dCommandList, pd3dTexture, \*ppd3dUploadBuffer, 0, 0, nSubResources, &vSubresources[0]);

D3D12\_RESOURCE\_BARRIER d3dResourceBarrier;

::ZeroMemory(&d3dResourceBarrier, sizeof(D3D12\_RESOURCE\_BARRIER));

d3dResourceBarrier.Type = D3D12\_RESOURCE\_BARRIER\_TYPE\_TRANSITION;

d3dResourceBarrier.Flags = D3D12\_RESOURCE\_BARRIER\_FLAG\_NONE;

d3dResourceBarrier.Transition.pResource = pd3dTexture;

d3dResourceBarrier.Transition.StateBefore = D3D12\_RESOURCE\_STATE\_COPY\_DEST;

d3dResourceBarrier.Transition.StateAfter = d3dResourceStates;

d3dResourceBarrier.Transition.Subresource = D3D12\_RESOURCE\_BARRIER\_ALL\_SUBRESOURCES;

pd3dCommandList->ResourceBarrier(1, &d3dResourceBarrier);

// delete[] pd3dSubResourceData;

return(pd3dTexture);

밉맵이 있다면 SubResource라는 단어를 필연적으로 사용하는데.

SubResource는 Index로 구분한다.

대부분은 Texture전체를 읽거나 할것이다.

D3D12CalcSubresource라고 하는 dxgi12내에 존재하는 구조체를 이용한다

배열로 자르면 Array Slice, 밉맵 레벨로 자르면 Mip Slice로 구분한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그런거 하기싫으면 위에있는 D3D12CalcSubresource이걸 호출하면 된다(이건 따라하기에 없는거같다…)

DDS파일들을 읽고 Resource 인터페이스 포인터를 가지고 있다면, 그 Resource를 Shader로 연결해야된다.

근데 그냥 연결은 안되니 View를 만들어서 연결을 해야된다.

리소스를 담을 그릇(메모리)를 할당하고, 그걸 바라볼 View를 만들고 그걸 연결해야되니까말이다.

그러므로 CreateShaderResourceView로 Shader에 연결을 진행해준다.

파라메터로는, Resource의 인터페이스 포인터, 리소스의 디스크립션, 어디에다가 만들지)를 인자로 받는다. 리소스는 힙이 있어야 하기에…

인터페이스 포인터는 DDS고대로 쓸것이다.

Texture Array – 텍스쳐 배열은 Shader 코드이다.

Texture2D라는 형식으로 배열을 만들수있다.

근데 Texture2D array라는 데이터 타입도 있는데, 텍스쳐 리소스 하나가 배열로 여러 개를 가지고 있는 것이다. 내가 기억하기론 이걸 이용해서 애니메이션의 배열도 만들 수 있던것으로 알고있는데 이건 확실하지 않다.

이런 텍스쳐의 배열을 만들고싶다면, 아까 본 제공파일에서 texassemble이라는 것을 이용해서 실행할때 옵션으로 array를 만들면 된다.

Shader Code를 만들어야 하는데

Texture2D에 해당하는 라는 변수를 선언하고

그 변수의 레지스터를 디스크립터에 연결해줄 때 몇번 디스크립터에다가 할지

그 Shader의 주소값은 어딘지를 BaseShaderRegister에 번호를 주면 된다.

샘플러 상태