DirectX12 Chapter4 폴리곤 표시

여기까지 화면묘화의 준비를 계속했습니다만, 본 장에서 조금씩 재밌어집니다. 폴리곤을 표시하고, 결국에는 간단한 세이더를 쓰는 것이 되겠습니다.

* 초점버퍼
* 초점버퍼뷰
* 인덱스버퍼
* 초점세이더
* 픽셀세이더
* 루트시그니처
* 초점레이아웃
* 파이프라인스테이트
* 뷰포트

같은 개념입니다. 많습니다만, 힘내봅시다.

4.1 그래픽스 파이프라인의 복습

그러면, 여기부터는 처음에 말한 [그래픽스 파이프라인]이 중요하기 때문에, 가볍게 복습해봅시다.

2.1장에서 기술한 대로, 그래픽스 파이프라인이란, 정보의 바구니릴레이 같은 것이며, 우선 초점정보를 입력하면서 넘기면서, 거기에서

\* IA->VS->HS->TS->DS->GS->RS->PS->OM

이며, 각각 [전의 스테이지에서 넘겨진 정보가 가공되고 다음의 스테이지에 넘겨진다] 라는 것입니다.

여기부터는, [파이프라인스테이트]를 사용하고 이 파이프라인을 구축해가는 것입니다만, 그를 위해서는 물론,

* 입력이 되는 초점정보의 정의
* 초점세이더와 픽셀세이더의 정의

가 최소한 필요가 되겠습니다. 거기에서 그것들의 정의를 진행해봅시다.

4.2 초점정보의 작성

우선은 초점정보로서, 가장 기본적인 삼각형을 정의해봅니다.

당연합니다만, 삼각형을 만드는데, 초점은 3개 필요합니다. 그러면, 1가지의 초점을 정의하기 위해서 필요한 정보로는 뭘까요

4.2.1 좌표정보

우선은 최소한으로서, 좌표의 정보가 필요합니다. 3D 좌표가 되기 위해 x, y, z라는 3가지의 변수를 float으로 갖게 한 것이라 생각합니다.

Struct Vector3

{  
 float x, y, z;

}

이고, 여기까지 쓴 것으로, 일단 이 4행을 지워버립니다. 사실은, 수학함수나 구조체같은 것을 직접 실장하는 것은 매우 수고가 들어갑니다.

\*DirectXMath 라이브러리

본서에서는 DirectXMath라는 라이브러리를 사용해봅시다. 우선, 필요한 헤더파일을 인클루드합니다.

#include <directxmath.h>

\*\*directxmath는 링크필요없음

또한, directxmath의 이용에 있어서, lib파일의 링크는 필요 없습니다.

DirectX SDK 2010 June이라는 버전 이전의 SDK를 사용해서 DirectX프로그래밍을 처리한 사람은, D3DX라는 라이브러리에 포함된 각종 수학함수나 구조체를 이용하고있을지도 모릅니다. 그런데 현재, DirectX SDK가 Windows SDK에 포함되도록 하는 중에 D3DX는 폐지됐고, 대신에 DirectXMath라는 라이브러리가 그 역할을 담당하고 있습니다.

\*XMFLOAT3 구조체

DirectXMath에는, 아까 정의한 Vector3 같은 것으로, XMFLOAT3같은 이름의 구조체가 정의되어 있습니다.

\*\*XMVECTOR구조체

또한,XMVECTOR 라는 구조체도 정의되어있습니다만, 사용법이 조금 어렵기 때문에, 번서에서는 XMFLOAT3를 사용합니다.

DirectXMath라이브러리의 이름공간에는 DirectX:: 입니다.

//using namespace DirectX

로 해도 좋고, 혹시 코딩룰같은것에 의해 using namespace을 사용하지 않는 경우에는

DirectX::XMFLOAT3 vertices[3]

로 해도 상관없습니다. 본서에서는 인쇄 공간의 관계도 있어서, 이후에는 using namespace DirectX; 을 선언하고 있는 전제로 기술해갑니다.

\*\*영단어 vertices

Vertices란, [초점]을 의미하는 영단어 vertex의 복수형입니다.

\*3가지의 초점의 정의

그러면, 3가지 초점을 정의해봅시다. 초점의 위치입니다만, 여기서는 화면(백버퍼)의 좌우(x좌표)와 상하(y좌표)가 각각 -1~1 이 되어 있다고 해봅시다. 그림으로 보면 4.1처럼 됩니다.

(-1, 1) 1 (1, 1)

-1 0 1

(-1, -1) -1 (1, -1)

z좌표는, 우선 0으로 해둡시다.

이처럼 3가지의 초점좌표를 정의해갑니다만, 여기서 주의점이 있습니다. 초점의 순서는 시계방향이 되도록 해주세요.

\*\*초점을 시계방향으로 합니다

이것은 앞으로 설명할 [뒷면컬링] 과 관계있습니다.

코드로는, 다음과 같습니다. 좌표의 수치는, 표시가 되어있고 여러가지 바꿔가 봅시다.

//XMFLOAT vertices

4.3 초점버퍼

초점정보를 정의했습니다만, 이 정보를 GPU에 넘기려고 해도, 이대로라면 GPU는 받아주질 않습니다. 받아주길 위한 버퍼 영역(리소스)를 GPU쪽에 준비시키고, 거기에 메모리 복사해서 흐르게 할 필요가 있습니다.

4.3.1 ID3D12Resource 오브젝트

DirectX12에서는, GPU쪽의 메모리영역을 확보하기 위한 ID3D12Resource 오브젝트를 사용합니다. CPU쪽이라는 new(보다는 malloc()함수)와 가까운 것이라 이미지해주세요.

ID3D12Resource오브젝트는, ID3D12Device::CreateCommittedResource()메소드를 사용해서 생성합니다.

\*\*다른 방법

ID3D12Device::CreateCommittedResource()메소드만이 아닌, ID3D12Device::CreateHeap()메소드와 ID3D12Device::CreatePlacedResource()메소드를 사용해서 생성하는 방법도 있습니다만, 실장이 어렵기 때문에 본서에서는 사용하지 않습니다.

ID3D12Device::CreateCommittedResource()메소드의 인수는 다음과 같이 정의되어 있습니다.

virtual HRESULT STDMETHODCALLTYPE CreateCommittedResource(

\_In\_ const D3D12\_HEAP\_PROPERTIES \*pHeapProperties, //힙설정구조체의 주소

D3D12\_HEAP\_FLAGS HeapFlags,//특히 지정하지 않기 때문에 d3d12\_heap\_flag\_none으로

\_In\_ const D3D12\_RESOURCE\_DESC \*pDesc,//리소스설정구조체주소

D3D12\_RESOURCE\_STATES InitialResourceState,//GPU에서 불러오기 전용으로 GENERIC\_READ

\_In\_opt\_ const D3D12\_CLEAR\_VALUE \*pOptimizedClearValue, //사용하지 않기 때문에 nullptr

REFIID riidResource,

\_COM\_Outptr\_opt\_ void \*\*ppvResource) = 0;

한번 본것만으로는 알기 어렵네요. 특히 D3D12\_HEAP\_PROPERTIES 구조체와 D3D12\_RESOURCE\_DESC 구조체의 2가지는, [힙의 설정]과 [리소스의 설정]이라 하고 어느쪽도 메소드를 표시하고 있고, 이해하기 위해서도 자세히 볼 필요가 있을것같습니다.

4.3.2 초점힙의 설정

우선, 초점힙의 설정을 지정하는 D3D12\_HEAP\_PROPERTIES 구조체에서 봅시다.

typedef struct D3D12\_HEAP\_PROPERTIES

{

D3D12\_HEAP\_TYPE Type; //힙의 종류(map할 필요가 있기 때문에 UPLOAD)

D3D12\_CPU\_PAGE\_PROPERTY CPUPageProperty; //CPU의 페이징 설정

D3D12\_MEMORY\_POOL MemoryPoolPreference; //메모리 풀이 어딘지 가리킨다

UINT CreationNodeMask;//단일 어댑터

UINT VisibleNodeMask; //단일 어댑터

} D3D12\_HEAP\_PROPERTIES;

\*\*초점힙설정은 매우 이해하기 어렵습니다

초점힙 설정은, 매우 이해하기 어려운 데이터입니다. 정확히 말하면, 필자도 정확히, 완벽히 설명할 자신이 없습니다. 따라서, 이하의 내용은 [공식사이트와 함께, 필자가 해석했다]라는 것으로 생각해주세요.

\*힙종류

D3D12\_HEAP\_PROPERTIES 구조체의 멤버 중, 힙의 종류를 가리키는 D3D12\_HEAP\_TYPE 열거체의 정의는 이하와 같습니다.

enum D3D12\_HEAP\_TYPE

{

D3D12\_HEAP\_TYPE\_DEFAULT = 1, //CPU에서 엑세스 안됨(맵 안됩니다)

D3D12\_HEAP\_TYPE\_UPLOAD = 2, //CPU에서 엑세스 가능(맵 가능)

D3D12\_HEAP\_TYPE\_READBACK = 3, //CPU에서 불러오기 가능

D3D12\_HEAP\_TYPE\_CUSTOM = 4 //커스텀 힙(복잡하기 때문에 자세한건 후술)

} D3D12\_HEAP\_TYPE;

폴리곤을 표시하기 위해 사용되는 힙은, D3D12\_HEAP\_TYPE\_DEFAULT, D3D12\_HEAP\_TYPE\_UPLOAD 가 대부분입니다. 거기서 우선은, 이 2가지에 대해 설명합니다.

우선, D3D12\_HEAP\_TYPE\_DEFAULT 는 [CPU에서 엑세스 안된다] 힙을 표시하고 있습니다.

이 때문에, 이 지정으로 힙을 만든 경우, 후술하는 ID3D12Resource::Map() 메소드에서 힙에 엑세스하려고 하면 실패합니다. 이 힙은 밴드폭이 최대로 넓은(즉 엑세스가 빠르기 때문에), GPU쪽만 다루기 위한 힙으로서 매우 유용합니다.

다음은 D3D12\_HEAP\_TYPE\_UPLOAD 입니다. 이 열거치에서 나타나는 힙은, CPU의 데이터를 GPU에 업로드하기 위해 사용합니다. 그 때문에 이 지정으로 만든 힙에는 CPU에서 엑세스 가능하고, ID3D12Resource::Map() 메소드에서 내용을 바꿔쓰는 것이 가능합니다. 편리한 것이지만, CPU / GPU 쌍방에서 엑세스가 D3D12\_HEAP\_TYPE\_DEFAULT의 힙과 비교하면 늦기 때문에, [CPU에서 1회 한해서 입력][GPU에서 1회 한해서 입력]을 하는 것 같은 경우에 추천됩니다.

아까 말한 대로, 일반적인, 초점에 대해서는 이상의 2가지의 열거치를 사용하게 됩니다만, 다른 2가지도 가볍게 소개해보겠습니다.

D3D12\_HEAP\_TYPE\_READBACK 에서 지정된 힙은, 다시 읽기 전용 힙입니다. 즉, CPU쪽에서 엑세스 가능한 힙입니다만, 밴드폭은 넓지 않습니다(비교적 느리다). GPU에서 가공, 계산한 데이터를 CPU쪽에서 활용하기 위한 것이라 할 수 있습니다.

마지막의 D3D12\_HEAP\_TYPE\_CUSTOM 입니다만, 이것을 이해하는 것은 조금 힘듭니다. D3D12\_HEAP\_TYPE\_CUSTOM 이외의 설정(D3D12\_HEAP\_TYPE\_DEFAULT같은) 경우, 후술의 페이징 설정이나 메모리풀 설정에 UNKNOWN을 지정한다면 좋고, 간단하고 편리합니다만, D3D12\_HEAP\_TYPE\_CUSTOM에서는 이 페이징 설정이나 메모리풀을 바르게 설정하지 않으면 안됩니다.

초점버퍼작성시에는, 힙의 종별로서 D3D12\_HEAP\_TYPE\_DEFAULT나 D3D12\_HEAP\_TYPE\_UPLOAD를 사용합니다. 이 때문에, D3D12\_CPU\_PAGE\_PROPERTY 와 D3D12\_MEMORY\_POOL의 열거치는 각각 D3D12\_CPU\_PAGE\_PROPERTY\_UNKNOWN과 D3D12\_MEMORY\_POOL\_UNKNOWN을 지정해둔다면 괜찮습니다.

\*CPU의 페이징 설정

\*\*본 목은 초심자에는 어려울지도 모릅니다

기본적으로, 이후의 얘기는 힙종류가 D3D12\_HEAP\_TYPE\_CUSTOM 밖에는 관계없기 때문에, 초심자는 4.3.3목까지 건너뛰어도 상관없습니다.

통상 (힙 종이 D3D12\_HEAP\_TYPE\_CUSTOM이외의 때)은 D3D12\_CPU\_PAGE\_PROPERTY\_UNKNOWN을 사용한다면 좋습니다만, 흥미가 있는 독자를 위해, 일단 뭔가 다른 것이 있는지를 봅시다.

enum D3D12\_CPU\_PAGE\_PROPERTY

{

D3D12\_CPU\_PAGE\_PROPERTY\_UNKNOWN = 0, // 생각 안해도 좋습니다

D3D12\_CPU\_PAGE\_PROPERTY\_NOT\_AVAILABLE = 1, //CPU에서 엑세스 불가

D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_COMBINE = 2, //라이트 컴바인

D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_BACK = 3 //라이트 백

} D3D12\_CPU\_PAGE\_PROPERTY;

[CPU에서 엑세스 불가]는 아직 알수 있다고 해도, [라이트컴바인]과 [라이트 백]은 뭘까요? 자세히 봅시다.

우선, D3D12\_CPU\_PAGE\_PROPERTY\_UNKNOWN 입니다. 이것은 힙 종류가 D3D12\_HEAP\_TYPE\_CUSTOM이외에 사용하는 설정입니다. 특히 설정하지 않아도 각각에 맞춰서 설정을 해줍니다. 역으로 말하면, D3D12\_HEAP\_TYPE\_CUSTOM으로 하면 이 지정은 허가되지 않고, 자신이 필요한 설정을 하는 것이 될 뿐입니다.

계속해 D3D12\_CPU\_PAGE\_PROPERTY\_NOT\_AVAILABLE 란, 읽는 글자 그대로 [CPU에서 엑새스 불가] 인 것을 나타냅니다. 즉, [이 힙은 GPU안에서의 계산에 한정하고 사용된다] 라는 것입니다. 힙 종류는 D3D12\_HEAP\_TYPE\_DEFAULT 때와 같은 상태가 된다고 말할 수 있습니다.

다음으로 D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_BACK 입니다만, CPU의 캐쉬에 있는 [라이트백방식]을 이미지한다면 알기 쉬울지도 모릅니다. 통상 CPU는 연산을 처리하고, 그 결과를 메모리에 다시 쓴다는 것 뿐입니다만, 이 때 [메모리와 캐쉬에 동시에 입력한다]는 것을 라이트스루 방식이라 하고, 이것에 대해 CPU의 연산결과를 [우선은 캐쉬에 입력, 중요한 타이밍에 메모리에 입력한다]는 것을 라이트백 방식이라 합니다. 이 열거치가 나타내는 것도 그것과 같이, CPU에서 GPU까지 전송하는 경우에 [CPU의 메모리를 캐쉬로서 사용한다]또는 [CPU쪽의 캐쉬메모리를 캐쉬로서, 시간의 공백이 생기자 마자, 순차 GPU쪽의 메모리에 전송한다]는 방식이라 생각할 수 있습니다.

\*\*D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_BACK에 대해

라이트백방식에 대해서는 [자신을 갖고 이거다!]라고 말할 수 없습니다.

마지막으로, D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_COMBINE입니다. COMBINE(결합) 이라는 의미에서 추측되는것처럼, 보내야하는 데이터가 있는 정도의 크기의 데이터에 모아서 전송하는 방식을 나타냅니다. 이 [모아서 전송한다]는 것을 [버스트 모드]라고 말하기도 합니다. 하지만, 모아서 보내는 때에는 순서가 고려되지 않기 때문에, 사용하는 때에는 자신있게 붙이는 것이 좋아보입니다.

D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_BACK과 D3D12\_CPU\_PAGE\_PROPERTY\_WRITE\_COMBINE 이라는 두 가지의 지정은, CPU에서 메모리에 데이터를 입력하고 GPU쪽에 전송합니다. 라는 흐름이 공통입니다만, 전송의 방식에 차이가 있다고 할 수 있습니다. 메모리 풀과의 맞춤은 정해지지 않았기 때문에, 나중에 표 4.1에서 소개하겠습니다.

\*메모리 풀 설정

다음은 메모리풀의 설정입니다. D3D12\_MEMORY\_POOL 열거체는 이하처럼 정의되어 있습니다.

enum D3D12\_MEMORY\_POOL

{

D3D12\_MEMORY\_POOL\_UNKNOWN = 0, //CUSTOM 이외의 때에는 이것으로 좋습니다

D3D12\_MEMORY\_POOL\_L0 = 1, //시스템메모리(어댑터가 UMA일 떄, GPU밴드폭은 좁음)

D3D12\_MEMORY\_POOL\_L1 = 2 //비디오메모리(어댑터가 UMA일 때, GPU밴드폭은 넓음)

} D3D12\_MEMORY\_POOL;

[힙 종류가 D3D12\_HEAP\_TYPE\_CUSTOM이외일 떄는 UNKNOWN이라도 상관없음]이라는 것은, CPU페이지의 설정과 같습니다.

D3D12\_MEMORY\_POOL\_L0는 시스템 메모리(CPU쪽의 메모리)를 나타냅니다만, 어댑터(그래픽스보드)에 따라 움직임이 바뀌는 것 같습니다. 공식 사이트의 해설을 보는 대로, 온보드GPU(UMA)의 경우에는, 이 열거치밖에 선택지가 없는 것 같습니다. 혹시, NVIDIA나 AMD같은 디스크리트그래픽스보드를 참조하고 있는 경우에 이 열거치를 고르면, CPU밴드폭이 넓고, GPU밴드 폭이 좁게 됩니다.

D3D12\_MEMORY\_POOL\_L1 은 비디오메모리를 나타냅니다. 이것은 온보드GPU(UMA)의 경우에는 사용이 안되는 파라미터 입니다. NVIDIA나 AMD의 디스크리트 그래픽스보드를 참조하고 있는 경우만 사용가능합니다. 이 설정을 하면 GPU용의 밴드 폭이 넓어집니다만, CPU쪽에서 엑세스 접근이 안되게 됩니다.

\*쿠미아와세(조합)

여기까지를 표로 종합해봅시다. 필자의 환경에서는, 이것 이외의 조합에서는 INVALIDARG가 돌아오기에 에러가 발생합니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 힙 종류  (D3D12\_HEAP\_TYPE\_ | 페이지 설정(D3D12\_CPU\_PAGE\_PROPERTY) | 메모리풀  (D3D12\_MEMORY\_POOL\_) | RESOURCE\_STATE | 맵의 가능, 불가능 |
| DEFAULT | UNKNOWN | UNKNOWN | GENERIC\_READ | 불가 |
| UPLOAD | UNKNOWN | UNKNOWN | GENERIC\_READ | 가 |
| READBACK | UNKNOWN | UNKNOWN | COPY\_DEST | 가 |
| CUSTOM | NOT\_AVAILABLE | L0/L1 | GENERIC\_READ | 불가 |
| WRITE\_BACK | L0만 | GENERIC\_READ | 가 |
| WRITE\_COMBINE | L0/L1(?) | GENERIC\_READ | 가 |

\*\*필자의 검증상황

유감이지만 표4.1은 NVIDIA의 그래픽스보드에서밖에 검증되지 않았습니다. 또한, 전부의 조합 검증을 행한 것도 아닙니다. AMD 등의 움직임이나, 용도에 맞춘 검증은 각자 행해주면 좋을것입니다.

또한, 필자의 환경에서는 D3D12\_CPU\_PAGE\_PROPERTY\_COMBINE 과 D3D12\_MEMORY\_POOL\_L1 의 조합 때의 동작은 불안정합니다.

또한, 표 4.1에서는 [맵의 가, 불가]라는 항목이 있습니다. 맵에 대해서는 다음 항목에서 설명합니다만, 간단히 말하면, 이 버퍼를 통상의 CPU쪽의 메모리엑세스처럼 다루기 위한 처리를 맵이라 하고, 이것이 가능한지 아닌지를 나타냅니다.

\*이번의 초점 힙의 설정

긴 부분동안 초점 힙의 설정 항목에 대해 해설했습니다만, 우선 이번에는, [초점데이터는 CPU쪽에서 설정하는 것이므로, 맵에서 설정한다] 로서 D3D12\_HEAP\_TYPE\_UPLOAD 를 지정해서 초점힙을 작성합니다.

D3D12\_HEAP\_TYPE\_UPLOAD 는 다른 지정에 비해 퍼포먼스가 나쁘기 때문에, 빈번하게 엑세스가 있는 경우에는 다른 설정을 검토해야 합니다. 그렇다고 해서, 제 7장에 나오는 모핑(블렌드세이프)까지 초점데이터는 매 프레임 1회정도 밖에 참조하지 않고, 현 시점에서는 다소 대역이 좋아도 문제 없습니다.

나중에 퍼포먼스를 고려하거나 용도에 맞추거나 해서 이것들의 버퍼의 작성법을 선택해 가는 것이 됩니다만, 우선은 단순히 항목을 GPU에 보내기 때문에, 이번에는 D3D12\_HEAP\_TYPE\_UPLOAD 를 사용합니다.

4.3.3 리소스 설정 구조체

그러면, 초점버퍼의 설정으로 돌아가봅시다. 초점 힙의 설정 다음에, 리소스의 설정을 처리합니다. 리소스의 설정에는 D3D12\_RESOURCE\_DESC 구조체를 사용합니다.

typedef struct D3D12\_RESOURCE\_DESC

{

D3D12\_RESOURCE\_DIMENSION Dimension; 버퍼로 사용하기 때문에 BUFFER를 지정

UINT64 Alignment; //0으로

UINT64 Width; //폭으로 전부 충당하기 때문에 sizeof(전 초점) 으로 합니다

UINT Height; //폭으로 표현하고 있기 때문에 1로 합니다

UINT16 DepthOrArraySize; //1로

UINT16 MipLevels; //1로

DXGI\_FORMAT Format; //화상이 아니기 때문에 UNKNOWN 으로 좋습니다

DXGI\_SAMPLE\_DESC SampleDesc; //SampleDesc.Count = 1 로합니다

D3D12\_TEXTURE\_LAYOUT Layout; //D3D12\_TEXTURE\_LAYOUT\_ROW\_MAJOR 로합니다

D3D12\_RESOURCE\_FLAGS Flags; //NONE으로

} D3D12\_RESOURCE\_DESC;

멤버 중에, Width나 height 는 텍스쳐의 경우 화상의 폭과 높이를 나타냅니다. 그런데 이번에는 텍스쳐가 아닌, 단지 초점정보의 집합체이므로, 코멘트에 쓴 대로의 지정이 됩니다. 또한 Format멤버도 같습니다.

또한, SampleDesc 멤버는 안티엘리어싱을 사용할 때의 파라미터입니다. 이번에는 초점정보만 있고 안티엘리어싱을 사용하지 않게 때문에, SampleDesc.Count = 1 로 해도 상관 없습니다. 본래적으로, SampleDesc.Count = 0 으로 하고 싶습니다만, 그렇게 하면 데이터가 없는 것으로 되버리기 때문에, 1을 넣습니다.

가장 골때리는 것은 Layout() 멤버입니다. D3D12\_TEXTURE\_LAYOUT 열거체의 정의를 보면 D3D12\_TEXTURE\_LAYOUT\_UNKNOWN가 적절하다고 생각할 수도 있습니다만, 이번에는 D3D12\_TEXTURE\_LAYYOUT\_ROW\_MAJOR 을 사용합니다.

마지막의 ID3D12Device:CreateCommittedResource() 메소드의 반환치로 S\_OK가 돌아오는가를 확인해 봅시다. 혹시 S\_OK 이외가 돌아왔을 경우에는, 어딘가 설정이 잘못된 것입니다. 고쳐주세요.

4.4 초점정보의 복사 (맵)

4.4.1 ID3D12Resource::Map() 메소드

여기까지 초점버퍼를 만들었습니다만, 실제로는 이 버퍼에 초점정보를 복사하지 않으면 GPU쪽에 초점을 넘기는 것이 안됩니다. 거기에서, ID3D12Resource::Map() 메소드를 이용해서 먼저의 버퍼에 초점정보를 복사해갑니다.

ID3D12Resource::Map() 메소드로는, 버퍼의 (가상)주소를 취득하기 위한 함수입니다. CPU쪾에서 이 주소상의 메모리에 대해 변경을 한다면, 그것이 GPU쪽에 전달한다는 것을 이미지 하는 것이 좋을겁니다.

ID3D12Resource::Map() 메소드의 인수는, 이하처럼 정의되어 있습니다.

HRESULT Map(

UINT Subresource, //미프 맵 같은 것은 아니기 때문에 0으로 좋습니다

const D3D12\_RANGE \*pReadRange, //범위지정. 전 범위이므로 nullptr로 좋습니다

void \*\*ppData) = 0; //받기 위한 포인터 변수의 주소

제 1 인수는, 리소스배열이나 미프맵의 경우, 서브리소스 번호를 넘깁니다. 다만, 이번에는 초점 버퍼 하나이므로 번호는 0가 됩니다.

제2인수의 D3D12\_RANGE 구조체는, 맵하고 싶은 범위를 특정하기 위한 것입니다. 버퍼의 일부만을 갱신하고 싶은 경우에 설정합니다만, 이번에는 초점 전부를 갱신하기 위한 nullptr 로서 전 범위를 넘깁니다.

제3인수는 [언제인 것]라는 느낌의, 포인터의 포인터라는 것입니다만, 포인터의 주소입니다. 보통의 포인터 변수를 선언하고 그 변수에 &를 붙여서 넘긴다면 좋습니다.

실제의 코드는 다음과 같이 됩니다.

//XMFLOAT3

맵한 vertMap에 대해, 사전에 만들어 둔 초점 데이터를 입력해 줌으로써, 버퍼상의 항목정보를 갱신하고 있습니다. 또한, 마지막의 ID3D12Resource::Unmap() 메소드는 [이미 맵을 해제해도 좋습니다] 라는 명령(언맵) 입니다.

그렇다고 해서, 이대로라면 실제로 초점을 넘겼는지 아닌지 모르기 때문에, 한번에 표시까지 처리합시다.

4.4.2 초점버퍼뷰의 작성

초점정보를 이용하기 위해서는 초점 버퍼뷰(D3D12\_VERTEX\_BUFFER\_VIEW 구조체)가 필요합니다. 이것은 [몇 바이트의 데이터가 존재하는가] [1초점 정도는 몇 바이트인가] 등을 알려주기 위한 데이터입니다.

구조체의 정의는, 이하 대로입니다.

typedef struct D3D12\_VERTEX\_BUFFER\_VIEW

{

D3D12\_GPU\_VIRTUAL\_ADDRESS BufferLocation;

UINT SizeInBytes;

UINT StrideInBytes;

} D3D12\_VERTEX\_BUFFER\_VIEW;

그러면 실제로 내용을 입력해봅시다.

//D3D12\_VERTEX\_BUFFER\_VIEW

ID3D12Resoucre::GetGPUVirtualAddress() 메소드는, 그 이름대로 GPU상의 버퍼의 가상 주소를 취득하기 위한 것입니다. 이 어드레스를 넘기는 것에 따라, GPU쪽은 어느 버퍼에서 데이터를 어느 정도 본다면 좋은가 알 수 있습니다.

나중에, 이 만든 초점 버퍼뷰를

\_cmdList->IASetVertexBuffers(0, 1, &vbView);

처럼 GPU에 전달해주는 것입니다만, 그것은 좀더 나중의 얘기입니다. 다음은 드디어 [세이더]를 써 가는 것이 되겠습니다.

4.5 처음 세이더

그러면, 드디어 기다려왔던 세이더를 써봅시다. 드디어, CPU쪽의 프로그래밍과는 다른 GPU쪽의 프로그램을 쓰는 것이 되겠습니다. 본서의 독자의 모두는 이미 DirectX나 Unity 에서 세이더를 쓴 적이 있을 지도 모릅니다만, 그렇지 않은 분을 위해서도 물론 해설해보겠습니다.

세이더란, HLSL라는 언어(C언어와 닮은 기법의 언어)로 쓰고 있습니다. 그 때문에, 통상의 C++프로그램과는 문법이 약간 다릅니다. 주의하면서 기술해주세요.

4.5.1 초점세이더의 기술

우선은 초점세이더를 기술해봅시다. Visual Studio 의 프로젝트 탐색기를 우클릭하고, 추가->새로운 항목을 선택합니다. 그리고, 여기까지는 평소처럼입니다만, 이번에는 HLSL->초점세이더파일 을 선택해서 HLSL 프로그램(확장자: .hlsl)을 만듭니다.

그림4,2

파일명은, 최초에 만드는 기본적인 초점세이더 라는 것으로 BasicVertexShader.hlsl 로 합니다.

BasicVertexShader.hlsl 의 안을 보면, 초점세이더 프로그램으로서, 이미 다음처럼 main() 함수가 쓰여져 있습니다.

//Float4 main

이번에는, 함수명만을 BasicVS 로 바꿔서, 이하의 내용으로 합니다

//float4 BasicVS

이 초점 세이더(BasicVS()함수)는 1초점으로 불리고, 초점데이터는 pos라는 인수로 넘겨집니다. CPU쪽의 초점데이터는 XMFLOAT3, 즉 float가 3개분의 데이터였습니다만, 여기서는 float4로 되어 있습니다. 그렇다고 해서, 4번째에는 특별한 의미가 있는 수치가 들어가는 것이 아닌, 또 다음의 파라미터로 지장이 있는 것도 아니기 때문에, 안심해주세요.

다음으로 POSITION이나 SV\_POSITION이라는 단어입니다만, 여기서는 2.2.3 항에서 기술한 [세먼틱스]가 있고, 각각 pos 나 BasicVS() 함수를 반환치인 float4가 어떻게 의미를 갖고, 어떻게 다뤄지는가를 기술한 것입니다. 먼저 기술한 대로, 파이프라인에서 바구니릴레이를 하기 위한 이 세먼틱스가 필요하게 됩니다. 예를 들면 반환치의 float4에 들어가는 SV\_POSITION는, [좌표]인 것을 의미하고 있고, 래스터라이저로 넘겨져 픽셀세이더로 보내져갑니다.

\*비쥬얼 스튜디오의 설정

어쨌든 이것으로 초점세이더를 쓴 것이 되었습니다만, 컴파일을 통하게 하기 위해서는 조금 더 필요합니다.

우선, 비주얼 스튜디오 상의 세이더(여기서는 (BasicVertexShader.hlsl))을 우클릭하고, (프로퍼티)를 선택합니다.

한다면, 그림4.3처럼 화면이 나오므로, 이하의 조작을 합니다.

* 엔트리포인터명을 [BasicVS]로 변경
* 세이더모델을 [Shader Model 5.0(/5\_0)] 이후로 한다

4.5.2 픽셀세이더의 기술

먼저 BasicVertexShader.hlsl 과 같이, 픽셀세이더로서 BasicPixelShader.hlsl을 작성하고, 프로젝트를 추가해봅시다. 그리고 이하처럼, 함수명을 BasicPS() 로 변경해주세요.

//float4 BasicPS

이것이 픽셀 세이더입니다. 초점세이서에서 바구니 릴레이로 전달된 좌표정보를 인수로서 받고, 렌더타겟으로 입력하기 위한 (색) 정보를 넘깁니다. 반환치의 세먼틱스인 (SV\_TARGET)은, 렌더타겟에 입력하기 위한 정보인 것을 가리키는 것입니다. 또한, BasicPS() 함수의 내용은 [백색](float4(1, 1, 1, 1))을 렌더타겟에 입력한다는 것이 됩니다.

BasicVertexShader 과 같이, BasicPixelShader 도 엔트리포인터 명의 설정(BasicPS)와, 세이더 모델의 설정을 진행하고, 컴파일 에러가 나오지 않는 것을 확인해주세요.

문제가 없으면, 초점세이더도 픽셀세이더도 최소한의 준비는 되었습니다. 다음은 이것들을 이용하는 DirectX쪽의 코드를 입력해봅시다.

4.6 세이더의 입력과 생성

다음으로, 쓴 세이더를 읽어오고, 세이더오브젝트를 만들어봅시다. 최종적으로, 그것들의 세이더 오브젝트를 렌더링 파이프라인의 일부로서 파이프라인스테이트 에 세트하는 것으로 세이더가 그 역할을 다하게 되었습니다.

4.6.1 ID3DBlob 형

세이더오브젝트는 ID3DBlob라는 형의 데이터입니다. 그렇다고 해서, ID3DBlobsms 세이더 오브젝트 이외에도 여러 용도로 사용되는 형 이고, [투영형]이라 할수 있습니다. 실제로, ID3DBlob 오브젝트는 [뭔가의 데이터의 덩어리]를 가리키기 위한 포인터와 그 사이즈를 갖고 있는, 그것이 어떤 데이터인가는 사용하는 쪽이 결정합니다.

\*\*BLOB란

덧붙여 BLOB란 (Binary Large Object)의 약어로, 큰 데이터의 덩어리를 가리키는 말입니다.

이번의 세이더 데이터는 크지도 않습니다만, 어쩄든 부정형의 데이터에 대해 이 명칭이 사용됩니다. 또한, [blob] 라는 영단어에는 또 다른 한가지, 판타지작품에 나오는 [슬라임]같은 끈적끈적한 젤리상태의 부정형의 물체라는 의미도 있어, 통틀어 이중 의미라 할 수 있습니다.

그러면, 세이더 오브젝트를 유지하기 위한 ID3DBlob\*형의 변수를 선언해봅시다.

//ID3DBlob\*

4.6.2 필요한 인클루드와 링크 지정

다음은 바로 세이더의 읽기를 진행해보려합니다만, d3d12.h 에서는 세이더의 컴파일 기능이 제공되지 않습니다. 거기에서, 컴파일에 필요한 헤더파일의 인클루드와, .lib 파일의 링크 설정을 기술할 필요가 있습니다.

우선, 필요한 헤더파일인 <d3dcompiler.h>을 인클루드 합니다.

//include<d3dcompiler

그리고, 다음처럼 d3dcompiler.lib의 링크를 기술합니다.

#pragma comment(lib, “d3dcomplier.lib

4.6.3 D3DCompileFromFile() 함수

그러면, 준비는 됐기 때문에, 바로 읽기를 해봅시다. 읽어오기 위한 함수는D3DCompileFromFile() 이라는 이름입니다. 이 함수의 인수는 다음처럼 정의되어 있습니다.

typedef HRESULT (WINAPI \*pD3DCompile)

(LPCVOID pSrcData,

SIZE\_T SrcDataSize,

LPCSTR pFileName, //파일명

CONST D3D\_SHADER\_MACRO\* pDefines, //세이더매크로오브젝트(nullptr)

ID3DInclude\* pInclude, //인클루드 오브젝트(후술)

LPCSTR pEntrypoint, //엔트리포인트(불러오는 세이더명)

LPCSTR pTarget, //어느 세이더를 할당할것인가(vs, ps등)

UINT Flags1, //세이더컴파일옵션(후술)

UINT Flags2, //이펙트컴파일옵션(0으로 권장)

ID3DBlob\*\* ppCode, //받기 위한 포인터의 주소

ID3DBlob\*\* ppErrorMsgs); //에러용 포인터의 주소

이 중에서 중요한 것은, 제 1인수의 파일명, 제4인수의 엔트리포인트, 제5인수의 세이더 타겟입니다만, 여기서는 하나씩 봅시다.

제1인수의 pFileName은 파일명입니다만, 와이드 문자열인 것에 주의해봅시다. 다음에 가리키는 것처럼, 와이드 문자열리터럴은, 선두에 L을 붙이는 것을 나타냅니다.

LPCWSTR str = L”와이드문자열”;

한편, 다른 인수인 엔트리포인트나, 타겟은 [와이드가 아닌] 문자열입니다.

\*\*char\*나 std::string에서 와이드 문자열로의 변환

와이드 문자열 리터럴을 사용하지 않고, char\*나 std::string 에서 와이드 문자열로 변환하는 방법은, 나중에 (8.4절에서) 해설합니다.

제2인수의 pDefines는, 이번에 사용하지 않기 때문에 nullptr을 지정합니다만, C언어에서 #define을 배열로 지정하기 위한 인수입니다.

제3인수의 pInclude에는, 뒤에 HLSL파일안에 인클루드를 처리하기 위한 D3D\_COMPILE\_STANDARD\_FILE\_INCLUDE 매크로를 지정해두둡니다. 이것에 의해, 세이더 안에 #include 문이 쓰여져 있는 경우, 인클루드 파일의 디렉토리로서, current directory(현재 디렉토리)를 참조하도록 됩니다. #include 문을 사용하지 않는 경우에는 nullptr 을지정하는 것도 가능합니다.

\*\*currentdirectory 이외에서 인클루드하는 경우

혹시 커렌트 디렉토리 이외를 인클루드 디렉토리에 하는 경우에는 프로그래머가 사전에 ID3DInclude에서 계승한 클래스를 만들고, 그 오브젝트를 넘길 필요가 있습니다만, 본서의 범위를 넘기 때문에 양해부탁드립니다.

제4인수의 pEntrypoint는, 통틀어 엔트리포인트입니다. 세이더 안의 어느 함수를 초점세이더 또는 픽셀세이더의 엔트리포인트로서 할당하는가를 지정합니다. 이번의 예라면 “BasicVS” 혹은 “BasicPS”가 됩니다.

제5인수의 pTarget는, 세이더의 종류와 버전을 지정하는 문자열입니다. 예를 들면, 초점세이더라면 vs, 픽셀세이더라면 ps가 되고, 그 다음 \_에 이어서 버전을 지정합니다. 이번에 사용하는 것은 초점세이더도 픽셀세이더도 버전 5.0이므로, 각각 “vs\_5.0” 과 “ps\_5.0”라 지정합니다.

제6인수 (Flags1)는 세이더컴파일옵션입니다. 컴파일옵션에는 매우 많은 종류가 있고, 그것들을 OR연산자( | )로 맞춰서 지정합니다. 종류가 꽤 많기 때문에 그 전부는 쓰지 않습니다만, 일부만 소개하겠습니다.

* D3DCOMPILE\_DEBUG : 디버그용
* D3DCOMPILE\_SKIP\_VALIDATION : 세이더의 검증(하지 않습니다)
* D3DCOMPILE\_SKIP\_OPTIMIZATION : 세이더의 최적화(하지 않습니다)
* D3DCOMPILE\_PACK\_MATRIX\_ROW\_MAJOR : 행렬은 행 우선합니다
* D3DCOMPILE\_PACK\_MATRIX\_COLUMN\_MAJOR : 행렬은 열 우선합니다.

그 외, 조금 조사해본바 20종류 이상의 옵션이 있었습니다. 이번에 처음 접하시는 분들도 많기 때문에, 디버그용으로 해둡시다. 또한, 최적화도 제어해두고 있기 때문에, 이 2가지를 조합한 파라미터(D3DCOMPILE\_DEBUG | D3DCOMPILE\_SKIP\_OPTIMIZATION)을 사용합니다.

제7인수의 Flag2는, 세이더 파일의 경우 0으로 하는 것이 추천되고 있습니다. 이번에도 0 으로 해둡시다. 0 이외의 수치에는 D3DCOMPILE\_EFFECT\_CHILD\_EFFECT와 D3DCOMPILE\_EFFECT\_ALLOW\_SLOW\_OPS 가 있습니다만, 여기서는 특별히 설명하지 않습니다. 마지막의 인수 ppErrorMsgs는, 에러가 일어났을 때, 에러내용을 받는 포인터를 지정합니다. 혹시 에러가 일어나지 않는다면, nullptr이 반환됩니다.

각 인수의 의미를 알았기 때문에, 실제로 초점세이더의 읽기 코드를 써봅니다.

//result = D3DCompileFromFile

실행하고, 반환치가 S\_OK가 된 것을 확인했다면, 픽셀세이더의 읽기코드도 써봅시다.

//result “”

\*에러가 일어난 경우

초점세이더와 픽셀세이더의 어느쪽도 반환치가 무사히 S\_OK 라면 좋습니다만, 에러가 나왔을 때 대처도 소개하겠습니다.

혹시 읽기의 때에 에러가 발생한 경우, 대강 세이더의 내용이 틀렸기 때문에, 내용을 우선 확인해봅시다. 세이더의 내용이 틀렸을 경우, ppErrorMsg에 세이더 컴파일러에서의 [매우 상세한] 메시지가 문자열로서 들어갑니다.

ppErrorMsg 안의 메시지를 보여주기 위해, 이하의 가리키는 2가지의 ID3DBlob의 메소드를 사용합니다.

* ID3DBlob::GetBufferSize() : 버퍼의 사이즈를 반환(메시지 문자열의 바이트 수)
* ID3DBlob::GetBufferPointer() : 버퍼의 주소를 반환(메시지 문자열의 주소)

이것들을 사용한다면, 다음 처럼 에러 메시지를 디버그윈도우에 표시가능합니다. 또한, 디버그윈도우로의 표시는 OutputDebugStringA() 함수를 사용하고 있습니다.

std::string errstr; //수치 받는 용 string

errstr.resize(errorBlob->GetBufferSize()); // 필요한 사이즈를 확보

//데이터 복사

std::copy\_n( (char\*)errorBlob->GetBufferPointer(),

errorBlob->GetBufferSize(),

errstr.begin());

OutputDebugStringA(errstr.c\_str()); // 데이터 표시

이외에도, 파일명이 틀리지는 않았는지 체크해야합니다. 파일명이 틀릴 경우, HRESULT\_FROM\_WIN32(ERROR\_FILE\_NOT\_FOUND)가 반환됩니다. 또한, 이 경우는, ppErrorMsg 오브젝트는 아무것도 들어가있지 않습니다(nullptr). 파일명을 다시 한번 봅시다.

그러면, 초점세이더 읽기에서 에러체크까지를, 종합해서 써보겠습니다. 혹시 세이더에 에러가 나오는 경우에는, 디버그윈도우를 확인하고, 해당 부분을 수정해주세요.

4.7 초점 레이아웃

여기까지 세이더 오브젝트를 작성했습니다. 그런데, [넘겨진 초점 데이터를 어떻게 해석할 것인가] 를 GPU쪽에 가르쳐주지 않으면, 거기까지는 초점으로서 사용할 수 없습니다. 지금까지는 아직 그저 바이트 데이터 덩어리입니다.

초점 데이터를 만든 프로그래머는 그것이 [XMFLOAT3가 3가지] 라는 것을 알고 있습니다만, GPU쪽에서 그림 4.4처럼 그저 데이터 덩어리로밖에 보이지 않습니다.

00 00 00 등

그림 4.4 초점 데이터도 그저 데이터 덩어리

솔직히 이것을 보고, 뭔가 어떻게 정렬되어있는가 판별불가능이겠지요. GPU도 똑같기 때문에, 가르쳐주지 않으면 안됩니다. 거기에서, [초점데이터를 어떻게 해석할것인가] 를 나타내는 초점레이아웃이라는 데이터를 이용합니다.

DirectX12에서는, 초점레이아웃은 D3D12\_INPUT\_ELEMENT\_DESC 구조체의 배열로서 정의합니다. 그렇다고 해서 이번에 초점좌표만의 데이터이므로, 배열요소수는 1입니다.

4.7.1 D3D12\_INPUT\_ELEMENT\_DESC 구조체

우선, D3D12\_INPUT\_ELEMENT\_DESC 구조체의 정의를 봅시다.

typedef struct D3D12\_INPUT\_ELEMENT\_DESC

{

LPCSTR SemanticName; //세먼틱스 이름

UINT SemanticIndex; //같은 세먼틱스명의 때에 사용하는 인덱스 (0으로 좋습니다)

DXGI\_FORMAT Format; //포맷(원소수와 비트수로 형을 나타냅니다)

UINT InputSlot; //입력슬롯인덱스(0으로 좋습니다)

UINT AlignedByteOffset; //데이터의 오프셋 위치, (D3D12\_APPEND\_ALIGNED\_ELEMENT 로 좋습니다)

//INPUT\_CLASSIFICATION\_PER\_VERTEX\_DATA로 좋습니다

D3D12\_INPUT\_CLASSIFICATION InputSlotClass;

UINT InstanceDataStepRate; //한번에 묘화하는 인스탄스의 수

} D3D12\_INPUT\_ELEMENT\_DESC;

그러면 D3D12\_INPUT\_ELEMENT\_DESC 구조체의 멤버인

* 세먼틱스 명: SemanticName
* 포맷 : Format
* 슬롯 관련 : InputSlot, AlignedByteOffset, InputSlotClass
* 인스탄싱관련 : InstanceDataStepRate

\*세먼틱스명

세먼틱스명(SemanticName) 은, 세이더의 해설 중에 나온 세먼틱스와 같은 것이므로, 데이터의 의미를 문자열로 기술한 것입니다. 이번에는 좌표이므로 “POSITION” 이 됩니다만, 그거 이외의 세먼틱스에 대해서는 앞으로 설명하겠습니다.

\*포맷(DXGI\_FORMAT 열거형)

포맷(Format) 에는 DXGI\_FORMAT 열거형을 지정합니다.

[DXGI\_FORMAT] 라는 키워드를 검색해서 나오는 공식사이트의 해설을 보면 알 수 있듯이, 포맷의 종류는 무수히 존재합니다. 그래도 본서 안에서 그 모두를 소개할 수는 없습니다만, 대략적인 구분법을 설명하겠습니다.

예를 들면, 이번의 좌표데이터라면 DXGI\_FORMAT\_R32G32B32\_FLOAT 를 지정하게 됩니다만, 이것은 크게 [R32G32B32] 와 [FLOAT]으로 나눠 생각하면 알기 쉽습니다.

우선 R32G32B32는, R과G와B라는 3가지의 요소가 있는 것을 가리킵니다. 이것이 4 요소의 경우에는 R32G32B32A32 라는 지정이 되고, 2요소밖에 없는 경우에는 R32G32 가 됩니다. 또한, RGB 각각의 뒤에 붙어 있는 수치는, 그 요소의 비트수가 됩니다. 다음으로 FLOAT입니다만, 이것은 문자대로 float형을 나타내고 있습니다.

이번에는, XMFLOAT3 로 좌표정보를 넘기고 있기 때문에, float형 (32비트) 3개분의 데이터입니다. 그 때문에, DXGI\_FORMAT\_R32G32B32\_FLOAT 이 되게 됩니다.

FLOAT 외에도, 사용빈도는 높은 데이터 형으로서는, 이하와 같은 것을 들 수 있습니다.

* SINT: 부호있는 정수형(비트수는 32 라 할 수 없음)
* UINT: 부호없는 정수형(비트수는 32라 할 수 없음)
* SNORM: -1.0f ~ 1.0f 로 정규화된 부동소수점(비트 수는 32라 할 수 없음)
* UNORM: 0.0f ~ 1.0f 로 정규화된 부동소수점(비트수는 32라 할 수 없음)
* TYPELESS: 형태 없음(여기서 형태를 특정하지 않고, 다른 요인에 따라 결정)

SNORM, UNORM 에 있는 [정규화됐다] 라는 것은, 각각 지정된 비트수로 -1.0f~1.0f, 0.0f~1.0f의 수를 표현하고 있다는 의미입니다. 예를 들면, 앞으로 등장할 [화소치], [심도치], [법선벡터] 등을 다룰 때에 사용합니다. 비트수는 32만이 아닌 8이나 16, 경우에 따라 10이나 2도 사용됩니다. 실제의 데이터의 정의에 따라, 적절한 지정을 처리해봅시다.

물론, DXGI\_FORMAT 열거형에 정의되지 않은 조합은 사용할 수 없습니다. 공식 사이트를 보면서 적절한 것을 골라봅시다.

\*슬롯관련 멤버

슬롯은, GPU가 초점데이터를 보는 [뺴는 구멍] 같은 것이라 이미지하면 좋을겁니다. DirectX쪽에서 ID3D12GraphicsCommandList::IASetVertexBuffers() 메소드를 사용해서 슬롯과 실제의 버퍼와 관련을 지어봅시다.

매우 단순한 케이스의 경우 슬롯은 하나로도 좋습니다만, 복수의 초점 데이터를 합쳐서 하나의 초점 데이터를 표현하는, 즉 복수의 슬롯을 나눠 데이터를 다루는 경우도 많습니다.

이러한 슬롯의 번호를 지정하는 것은, InputSlot 멤버입니다.

복수의 슬롯에 초점데이터가 입력되는듯한 양상일 떄, [이 데이터와 이 데이터는 슬롯 n부터] [이 데이터는 슬롯 m부터] 라는 형태의 배치를 합니다. 하지만, 이번에는 초점데이터는 하나로 모아져있기 때문에 0번 슬롯만 사용합니다. 이처럼, 데이터를 한번에 흘려집어넣는 [인터리프]라 불립니다. (그림 4.5 인터리프 이미지)

초점 데이터 GPU

(초점데이터로서 데이터를 본다)-> 슬롯 0 (ID3D12CommandList::IASetVertexBuffers() 메소드에 의한 지정) -> VRAM

슬롯1

슬롯2

물론 경우에 따라 예를 들면 [좌표 데이터]와 [법선], [칼라]처럼 슬롯을 나누는 것도 있습니다. 이처럼 데이터의 버퍼를 나누는 것을 [비인터리프]라 부릅니다.

초점 데이터 GPU

(초점데이터로서 데이터를 본다)-> 슬롯 0 -> VRAM

(법선과 초점컬러데이터로서 데이터를 본다)->슬롯1 -> VRAM

슬롯2

다음은 AlignedByteOffset 입니다. 이 수치는, [그 데이터의 장소]를 가리킵니다. 예를 들면, 좌표데이터의 다음에 법선 데이터가 오면서, 그 때의 법선데이터의 장소는 32비트 (4바이트)의 float 3개분을 나중에 하면서, 12(바이트) 라 지정합니다. 그런데, 하나하나 데이터 사이즈를 계산하는 것도 귀찮기 때문에, 차례차례 데이터가 이어져 있는 경우는, 연속하고 있는 것을 나타내는 정수, D3D12\_INPUT\_CLASSIFICATION\_PER\_VERTEX\_DATA 를 지정합니다. 이 수치는 [데이터의 내용으로사, 1초점째로 이 레이아웃이 들어가 있다] 라는 것을 나타내고 있습니다. 외에도 D3D12\_INPUT\_CLASSIFICATION\_PER\_INSTANCE\_DATA 라는 열거치도 있습니다만, 이것은 인스탄싱 (13.2.2항에 후술)이라는 기술을 사용할 때 지정합니다.

\*\*인스탄싱관련 멤버

마지막의 멤버 InstanceDataStepRate는, 한번에 묘화하는 인스탄스의 수를 지정합니다. 그렇다고 해서, 이번에는 인스탄싱을 이용하지 않기 때문에 0으로 해둡시다.

혹시 복수의 인스탄싱을 처리하고, 또 이 데이터가 인스탄스째의 데이터인 경우, 이 파라미터가 2 이상이라면, 그 데이터는 그 수만 같은 데이터를 루프 사용하는 것이 됩니다. 자세한 것은 13.2.2항에서 인스탄싱을 소개할 때에 맞춰서 해설합니다.

\*실제의 기술

여기까지 왔고, 이번의 초점 레이아웃은 이하처럼 기술가능합니다.

//D3D12\_INPUT\_ELEMENT\_DESC

조금 깁니다만, 틀리지 않고 써봅시다. 여기까지 됐으면, 이 inputLayout 을 파이프라인스테이트의 적절한 부분에 할당하는 것이 됩니다.

4.8 그래픽스파이프라인 스테이트의 작성

그러면, 본래라면 이 다음은 루트시그니처를 해설하는 것이 자연스럽습니다만, 여기까지의 단계에서 [초점정보] [초점세이더] [픽셀세이더]를 만들고 있기 때문에, 우선 먼저 [그래픽스파이프라인 스테이트]를 만듭시다.

\*\*물론 실행은 실패합니다

루트시그니처를 작성하지 않기 때문에, 본절에서 일단 실행에 실패합니다. 그렇다고 해서, 그것도 상정 안이기 때문에, 신경쓰지 말고 나아갑시다.

여기까지라도, 파이프라인이라는 말은 몇번이고 나옵니다. 여기서 파이프라인이란, 물론 [그래픽스파이프라인 (렌더링파이프라인)] 이라는 의미입니다만, DirectX12에는 이것 이외에도 [컴퓨트 파이프라인] 이라는 파이프라인도 존재합니다. 그런데 지금은, 컴퓨트파이프라인은 일단 잊어버리고, 그림을 표시하기 위한 그래픽스 파이프라인을 설정합시다.

4.8.1 그래픽스 파이프라인이란

DirectX12에서는, 그래픽스 파이프라인의 설정을 정의하기 위한 그래픽스 파이프라인 스테이트라는 오브젝트 (ID3D12PipelineState\* 형)을 사용합니다. 앞서 설명했습니다만. 세이더나 그 외의 [그래픽스 파이프라인에 관련된 설정]을 하나로 모아서 효율화를 시키기 위해 태어난 것이 이 그래픽스 파이프라인 스테이트입니다.

그래픽스 파이프라인 스테이트를 만들기 위해서는 ID3D12Device::CreateGraphicsPipelineState() 메소드를 사용합니다.

virtual HRESULT STDMETHODCALLTYPE CreateGraphicsPipelineState(

\_In\_ const D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC \*pDesc, //각 파라미터 설정

REFIID riid, //평소의 IID\_PPV\_ARGS로

\_COM\_Outptr\_ void \*\*ppPipelineState) = 0; //평소의 처리

마지막의 2인수, riid와 ppPipelineState는 예전의 IID\_PPV\_ARGS 매크로를 사용하면 좋습니다. 문제는 제 1인수의 pDesc (D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC\*형), 즉 각 파라미터의 설정입니다. 다음은 이 구조체에 대해 알아봅시다.

4.8.2 D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC 구조체

바로, D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC 구조체의 정의를 봅시다. 그래픽스 파이프라인에 포함되어있는 요소로서, 다수의 멤버가 존재합니다.

typedef struct D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC

{

ID3D12RootSignature \*pRootSignature; //루트 시그니처

D3D12\_SHADER\_BYTECODE VS; //초점세이더

D3D12\_SHADER\_BYTECODE PS; //픽셀 세이더

D3D12\_SHADER\_BYTECODE DS; //도메인 세이더

D3D12\_SHADER\_BYTECODE HS; //헐 세이더

D3D12\_SHADER\_BYTECODE GS; //지오메트리 세이더

D3D12\_STREAM\_OUTPUT\_DESC StreamOutput; //스트리밍출력 버퍼 설정(불요)

D3D12\_BLEND\_DESC BlendState; //블랜드 파라미터

UINT SampleMask; //샘플마스크

D3D12\_RASTERIZER\_DESC RasterizerState; //래스터라이저의 설정

D3D12\_DEPTH\_STENCIL\_DESC DepthStencilState; //심도설정

D3D12\_INPUT\_LAYOUT\_DESC InputLayout; //초점 레이아웃

D3D12\_INDEX\_BUFFER\_STRIP\_CUT\_VALUE IBStripCutValue; //스트립때의 컴 수치 설정

D3D12\_PRIMITIVE\_TOPOLOGY\_TYPE PrimitiveTopologyType; //토폴로지 타입 삼각형으로 좋음

UINT NumRenderTargets; //렌더타겟 수

DXGI\_FORMAT RTVFormats[ 8 ]; //0번만을 R8G8B8A8\_UNORM 으로 설정

DXGI\_FORMAT DSVFormat; //뎁스치 포맷(이번에는 설정 불요)

DXGI\_SAMPLE\_DESC SampleDesc; //멀티샘플링의 설정

UINT NodeMask; //어댑터는 1개만 사용하기 때문에 0으로 합니다

D3D12\_CACHED\_PIPELINE\_STATE CachedPSO; //고속화에 관계하고 있지만 이번에는 설정하지 않음

D3D12\_PIPELINE\_STATE\_FLAGS Flags; //툴 디버그하는가 안하는가(이번에는 설정 불요)

} D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC;

이처럼, D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC 구조체에는 많은 멤버가 있습니다만, 이하처럼 종합하는 것이 가능합니다.

우선, 세이더 오브젝트는 이하 5개 입니다.

* 초점세이더: VS(필수, 작성완료)
* 픽셀세이더: PS(필수, 작성완료)
* 헐세이더: HS
* 도메인 세이더: DS
* 지오메트리세이더: GS

\*\*필수가 아닌 세이더도 있습니다.

일부의 세이더는 필수가 아닙니다. 제1장과 2장을 다시 확인하면 좋습니다.

그 외에도, 이하의 멤버도 필요하게 됩니다.

* 블랜드파라미터: BlendState(필수)
* 래스터라이저의 설정: RasterizerState(필수)
* 심도설정: DepthStencilState(필수)
* 초점레이아웃: InputLayout(필수, 작성완료)
* 토폴로지타입: PrimitiveTopologyType(필수)
* 그 외

그러면, 후술하는 루트시그니쳐와 관련지을 필요가 있기 때문에

* 루트 시그니쳐: pRootSignature(필수)

가 필요합니다. 루트시그니처에 대해 4.9절에서 해설합니다만, 이번에는 [텓ㄱ스쳐나 정수를 GPU로 넘기기 위해 필요한 것은 무엇인가]라 이미지해두면 좋습니다. 그렇다고해서, 이번에는 초점정보만이므로 빈 루트시그니처를 넘기는 것이 됩니다.

여기부터는, 실제로 이 멤버에 수치를 설정해갑니다.

4.8.3 세이더의 세트

우선, 세이더의 설정에서 진행합니다. D3D12\_SHADER\_BYTECODE구조체는, 멤버로서 세이더의 바이트코드의 포인터(pShaderBytecode) 로 사이즈 정보(BytecodeLength) 를 갖고 있습니다. 그것에, 이전 만든 ID3D12Blob 오브젝트에서 취득한 포인터와 사이즈를 대입합니다.

이번에는 초점세이더와 픽셀세이더만을 세트합니다.

4.8.4 샘플마스크와 래스터라이저 스테이트의 설정

다음으로, 샘플마스크(SampleMask) 와 래스터라이저(RasterizerState) 의 각 멤버를 설정합니다.

//gpipeline.Samplemask

CullMode (D3D12\_CULL\_MODE 열거형)는 뒷면 컬링을 처리하는가 아닌가 지정하기 위한 멤버입니다. 본서에서 다루는 모델 데이터 (PMD모델) 에는 뒤쪽에 폴리곤이 없는 경우도 많기 때문에, 컬링하지 않는 D3D12\_CULL\_MODE\_NONE 을 선택합니다. 또한, 이번에 폴리곤의 안을 칠하기 위해, FillMode(D3D12\_FILL\_MODE열거형) 에는 D3D12\_FILL\_MODE\_SOLID를 지정합니다.

또한 심도방향의 클리핑은 나중에 필요하게 되기 때문에 지금부터 true 로 합니다.

4.8.5 블렌드 스테이트 설정

다음으로, 블렌드 스테이트 (BlendState : D3D12\_BLEND\_DESC형) 의 설정을 처리합니다. D3D12\_BLEND\_DESC 구조체에는, 전부의 렌더타겟이 공통으로 설정하는 항목과 렌더타겟째로 설정하는 항목이 있고, 렌더타겟의 항목은, 경우에 따라, [사용하는 렌더타겟 만] 개별로 설정하지 않으면 안됩니다.

D3D12\_BLEND\_DESC 구조체의 정의는 이하처럼 되어 있습니다.

typedef struct D3D12\_BLEND\_DESC

{

BOOL AlphaToCoverageEnable; //지금은 false로 합니다 (후술)

BOOL IndependentBlendEnable; //지금은 false로 합니다 (후술)

D3D12\_RENDER\_TARGET\_BLEND\_DESC RenderTarget[ 8 ]; //렌더타겟의 설정

} D3D12\_BLEND\_DESC;

우선 AlphaToCoverageEnable 멤버의 해설을 하기 전에, 종래의 [a테스트] 에 대해 해설해봅시다.

a태스트란, 래스터라이저가 묘화하려하고있는 픽셀의 a치를 알아보고, [a가 0이라면 그 픽셀묘화자체를 파괴한다]는 것에 의해 묘화대상에서 벗어나는 것을 말합니다. 이것을 행하지 않으면. 완전히 투명한, 묘화할 필요가 없는 픽셀에도 심도가 입력되어버리기 때문에, 설령 텍스쳐의 일부분이 투명해도 그 폴리곤의 너머가 보이지 않게 되어버리는 것이 됩니다. (바로앞에 심도치가 입력되어버리면, 보다 멀리 있는 것이 묘화되지 않기 때문에)

그림 4.7 텍스쳐에 투명부분이 있는 폴리곤의 너머에 다른 폴리곤이 있는 상황

그 때문에, (묘화순서에 따라 다르지만) 혹시 바로앞이 먼저 묘화된 경우, a테스트없음 의 경우는 너머쪽의 폴리곤이 묘화되지 않게 됩니다.

또한 처리부담의 면에서도, 묘화할 필요가 없는데 픽셀세이더의 처리가 되어버리는 경우 아깝다고, 쓸데없다고 생각합니다. A가 0이라도, a테스트가 아닌 [a블렌딩]을 행하는 경우, 원래의 색과 묘화색을 블렌드한 색을 취득하고, 그것을 다시 쓰면서 그 나름대로의 비용이 드는겁니다.

이것을 회피하기 위해서, a테스트가 아닌, 픽셀세이더에 이하처럼 기술을 하는 것으로 실현가능합니다.

If(color.a == 0)

{  
discard;; //discard는 대상픽셀을 파괴 (심도치도 들어가지 않음)

}

하지만 이것은 장황한 프로그램이 되기 때문에, 전통적인 a테스트를 on으로 하는 것으로 이 문제를 회피 가능합니다.

그런데, DirectX12의 파이프라인 스테이트에는 AlphaTest 같은 멤버가 없습니다. 대신에 AlphaToCoverageEnable 이라는 멤버가 추가되었습니다. 그렇다고해서, 본서의 범위안에서는, 멀티샘플링(안티엘리어싱의 적용)이 아닌 경우, AlphaToCoverageEnable 의 수치가 그대로 a 테스트의 유무를 나타낸다고 생각하면 좋습니다.

한번 설명해두면, 종래의 AlphaTest플래그와 차이는, 멀티샘플링에 의한 안티엘리어싱이 적용된 때 발군합니다. 종래의 AlphaTest 플래그에서는 [대상 픽셀의 a가 0인가 아닌가]밖에 보지 않았기 때문에, 멀티샘플링때에 발생하는 [중간치의 a]가 빠지지 않고 더러운 결과로 되어버렸습니다. 그 때문에, 멀티샘플링때의 [망라율] 까지 고려해서 묘화를 하는 것이 AlphaToCoverageEnable 입니다.

Coverage(망라율) 이라는 단어에서 알 수 있듯이, 안티엘리어싱의 결과, 대상픽셀의 점유정도로 a블렌딩을 처리합니다. 이번에는 a테스트도 안티엘리어싱도 하지 않기 때문에 false로 합니다만, 독자의 용도에 따라 true로 해도 좋습니다.

다음으로 IndependentBlendEnable 을 true 로 지정한다면, 후술하는 합계 8개의 렌더타겟 (RenderTarget[])에 대해 각각 다른 블렌드 스테이트를 할당되도록 합니다. False를 지정하면 하나의 블렌드 스테이트 (RenderTarget[0])가 전부의 렌더타겟이 적용됩니다. 이번에는 false로 지정합시다.

렌더타겟의 설정인 D3D12\_RENDER\_BLEND\_DESC 구조체도 봅시다.

typedef struct D3D12\_RENDER\_TARGET\_BLEND\_DESC

{

BOOL BlendEnable; //가산, 증산, a같은 블렌드를 할지 말지

BOOL LogicOpEnable; //논리연산할지 말지

D3D12\_BLEND SrcBlend; //묘화 원 계수

D3D12\_BLEND DestBlend; //묘화 선 계수

D3D12\_BLEND\_OP BlendOp; //어떻게 블렌드 할 것인가

D3D12\_BLEND SrcBlendAlpha; //묘화원a계수

D3D12\_BLEND DestBlendAlpha; //묘화선a계수

D3D12\_BLEND\_OP BlendOpAlpha; a를 어떻게 블렌드할 것인가

D3D12\_LOGIC\_OP LogicOp; //어떤 논리연산할것인가

UINT8 RenderTargetWriteMask; //입력할 때 마스크 치

} D3D12\_RENDER\_TARGET\_BLEND\_DESC;

멤버 수가 많습니다만, 하나하나 어려운 것은 아닙니다. 기본적으로는, 렌더링때의 [묘화] 와 [묘화원]을 어떻게 합성할지에 대한 지정을 하고 있습니다.

우선, BlendOp(D3D12\_BLEND\_OP열거체)이나 LogicOp(D3D12\_LOGIC\_OP열거체)인 Op와는, Operator(연산)의 약어로서, 이 멤버는 [어떻게 연산할지] 를 나타냅니다.

또한 BlendEnable과 LogicOpEnable은, [각각 연산을 하는가 안하는가]를 나타냅니다. 또한, 이 2가지의 플래그는 동시에 true로 되지 않고, 어느것의 연산을 고를지가 됩니다. 그렇다고해도, 현 단계에서는 특별히 블렌드를 하지 않기 때문에, 이번의 코드에서는 Blend도 LogicOp도 false로 해 둡시다.

게다가, RenderTargetWriteMask는 RGBA 각각의 수치를 입력하는지 아닌지 지정하기 위한 수치입니다. 예를 들면 빨간 성분만 블렌드 하고 싶다면 D3D12\_COLOR\_WRITE\_ENABLE\_RED 를 지정합니다. 이 정수에는 RED/GREEN/BLUE/ALPHA 가 있고, 각각 1비트씩으로 표현되고, 전체가 4비트로 관리되고 있습니다. 또한, 전부의 요소를 블론드했다면 D3D12\_COLOR\_WRITE\_ENABLE\_ALL을 지정합니다.

이상, 이번의 코드는 이하와 같습니다.

//gpipeline.blendstate.alphatocoverageenable

\*블렌드처리의 설정

그러면, 이번에 특별히 블렌드를 하진 않습니다만, 블렌드를 하는 경우에 대해서도 해설해보겠습니다. 하지만, 이하는 약간 귀찮은 내용이기 때문에, 초심자는 4.8.6까지 건너뛰어도 상관없습니다,

호깃 가감산 같은 블렌드를 하는, 즉 BlendOpEnable 을 true로 하는 경우, BlendOp 는 이하의 열거치에서 선택하게 됩니다.

D3D12\_BLEND\_OP\_ADD = 1, //2가지의 수치를 가산

D3D12\_BLEND\_OP\_SUBTRACT = 2, //2가지의 수치를 감산

D3D12\_BLEND\_OP\_REV\_SUBTRACT = 3, //2가지의 수치를 감산(역순)

D3D12\_BLEND\_OP\_MIN = 4, //2가지의 수치 중 작은 쪽

D3D12\_BLEND\_OP\_MAX = 5 ////2가지의 수치 중 큰 쪽

\*\*증산이 없는 이유

가산, 감산만이고 증산이 없는 것은, 후술한대로 계수의 설정과 조합으로 실현 가능하기 때문입니다.

또한, LogicOpEnable을 true로 하는 (논리연산을 행함) 경우, LogicOp를 이하의 열거치에서 고르게 됩니다.

D3D12\_LOGIC\_OP

{

//전부 비트를 0으로 한다

D3D12\_LOGIC\_OP\_CLEAR = 0,

//전부 비트를 1로 한다.

D3D12\_LOGIC\_OP\_SET = ( D3D12\_LOGIC\_OP\_CLEAR + 1 ) ,

//묘화원의 수치를 그대로 사용(Src)

D3D12\_LOGIC\_OP\_COPY = ( D3D12\_LOGIC\_OP\_SET + 1 ) ,

//묘화원의 수치를 반전시킨 것을 사용 (~Src)

D3D12\_LOGIC\_OP\_COPY\_INVERTED = ( D3D12\_LOGIC\_OP\_COPY + 1 ) ,

//연산하지 않는, 즉 묘화처 그대로(Dst)

D3D12\_LOGIC\_OP\_NOOP = ( D3D12\_LOGIC\_OP\_COPY\_INVERTED + 1 ) ,

//타겟(묘화대상)을 반전 (~Dst)

D3D12\_LOGIC\_OP\_INVERT = ( D3D12\_LOGIC\_OP\_NOOP + 1 ) ,

//묘화원과 묘화대상을 논리 곱연산(Src & Dst)

D3D12\_LOGIC\_OP\_AND = ( D3D12\_LOGIC\_OP\_INVERT + 1 ) ,

//묘화원과 며화처를 부정논리 곱(논리적 이외의 부분) 연산(~(Src & Dst))

D3D12\_LOGIC\_OP\_NAND = ( D3D12\_LOGIC\_OP\_AND + 1 ) ,

//묘화원과 묘화대상을 논리합연산(Src | Dst)

D3D12\_LOGIC\_OP\_OR = ( D3D12\_LOGIC\_OP\_NAND + 1 ) ,

//묘화원과 묘화대상을 부정논리합(논리합이외의 부분)연산(~(Src & Dst))

D3D12\_LOGIC\_OP\_NOR = ( D3D12\_LOGIC\_OP\_OR + 1 ) ,

//묘화원과 묘화대상을 배타적논리합연산(Src ^ Dst)

D3D12\_LOGIC\_OP\_XOR = ( D3D12\_LOGIC\_OP\_NOR + 1 ) ,

//묘화원과 묘화대상이 같은 비트면 1, 그 이외는 0(~(Src ^ Dst))

D3D12\_LOGIC\_OP\_EQUIV = ( D3D12\_LOGIC\_OP\_XOR + 1 ) ,

D3D12\_LOGIC\_OP\_AND\_REVERSE = ( D3D12\_LOGIC\_OP\_EQUIV + 1 ) ,

D3D12\_LOGIC\_OP\_AND\_INVERTED = ( D3D12\_LOGIC\_OP\_AND\_REVERSE + 1 ) ,

D3D12\_LOGIC\_OP\_OR\_REVERSE = ( D3D12\_LOGIC\_OP\_AND\_INVERTED + 1 ) ,

D3D12\_LOGIC\_OP\_OR\_INVERTED = ( D3D12\_LOGIC\_OP\_OR\_REVERSE + 1 )

} D3D12\_LOGIC\_OP;

\*\*LogicOp의 해설에 대해

필자도 LogicOp는 보통 사용하지 않는 지정이기 때문에, 이해부족에서 불충분한 설명이 되고 있습니다. 본서에서는 BlendOp 밖에 다루지 않기 때문에, 일단의 참고정도로서 봐주세요.

또한, 이것들의 해설에 관해서는, DirectX12 의 리퍼런스보다도 Vulkan의 쪽이 충실하고 잇습니다. 혹시 알아보고 불명한 점이 있다면, Vulkan을 참조해보면 좋을겁니다.

\*블렌드 계수

다음으로, SrcBlend, DestBlend, SrcBlendAlpha, DestBlendAlpha 에 대한 것입니다. 이것들은 여러 계수를 지정하기 위한 멤버이므로, 여기서는 그것들의 공통의 형태인 D3D12\_BLEND 열거형에 대해 봅시다. 주요 열거치를 나열해보면, 이하와 같습니다.

D3D12\_BLEND\_ZERO = 1,

D3D12\_BLEND\_ONE = 2,

D3D12\_BLEND\_SRC\_COLOR = 3,

D3D12\_BLEND\_INV\_SRC\_COLOR = 4,

D3D12\_BLEND\_SRC\_ALPHA = 5,

D3D12\_BLEND\_INV\_SRC\_ALPHA = 6,

D3D12\_BLEND\_DEST\_ALPHA = 7,

D3D12\_BLEND\_INV\_DEST\_ALPHA = 8,

D3D12\_BLEND\_DEST\_COLOR = 9,

D3D12\_BLEND\_INV\_DEST\_COLOR = 10,