DirectX12 Chapter3 초기화부터 화면 클리어까지

3.1 처음 프로젝트

그러면 지금부터, 프로젝트를 만들어 봅시다.

비쥬얼 스튜디오 2019(2022)의 기동화면에서, [새로운 프로젝트 작성]을 선택합니다. 열려진 윈도우에서 언어로서 C++를, 프로젝트템플릿으로서 [빈 프로젝트]를 선택하고, [다음]을 클릭합니다.

그림3.1

표시된 다이알로그에서, 프로젝트명을 입력하고 나서 [작성]버튼을 눌러봅시다. [빈 프로젝트] 같은 것으로 당연 아무것도 포함하지 않는 프로젝트입니다만, [솔루션 탐색기]에서 프로젝트를 우클릭하고, [추가]->새로운 항목]에서 main.cpp를 추가해주세요. 지금은 main.cpp라는 1파일만 프로그램을 기술해봅시다. main.cpp의 내용은, 우선 다음의 내용을 그대로 써주세요.

[스크립트]

여기까지 쓰셨다면 일단 실행해주세요. 아직 DirectX는 관계없기 때문에 실행은 됩니다만, 일단 동작을 확인해봅시다. 또한, 이 코드에서는 디버그모드와 비 디버그모드로 main()함수와 WinMain()함수를 나누어 사용하고 있습니다. 왜냐하면, 초반에는 커맨드라인 출력에서 정보를 표시가능하도록 하기 때문입니다.

\*\* 디버그 표시

<debugapi.h>헤더에 포함된 OutputDebugString()함수를 사용하는 경우는, main()함수를 사용할 필요는 없습니다.

3.2 윈도우 표시와 Direct3D의 초기화

Windows프로그래밍에 있어서, 윈도우표시의 코드는 정형문이라 할 수 있습니다.

기억해도 할 수 없는 것이듯이, 알아보면 바로 나옵니다. 본서에서도 윈도우 표시에 대해서의 상세한 설명은 생략합니다.

3.2.1 윈도우의 생성

우선 [윈도우프로시져](함수)를 만듭니다.

[스크립트 62 ~ 63]

다음으로 main() 함수안에

1. 윈도우 클래스의 생성 & 등록
2. 윈도우 오브젝트의 작성
3. 윈도우 표시

를 처리합니다

//스크립트 const unsigned int window\_height, width 정의할것

여기까지 쓰고 실행하면, 윈도우가 한순간 표시되고, 바로 종료되어버립니다. 거기서 다음에, 메시지 루프(게임 루프)를 쓰고 바로 종료하지 않도록 합니다

//스크립트

여기까지 썼다면, 실행해봅시다. 그림 3.2처럼 되는 윈도우가 표시되었다면 성공입니다.

* + 1. Direct3D의 초기화

그러면 Direct3D의 초기화를 해봅시다. 조금 귀찮은 것이지만, 힘내봅시다. 여기서 초기화(오브젝트생성) 할 필요가 있는 것은

\*D3D12Device

\*DXGIFactory

\*DXGISwapchain

의 3가지 입니다. 실제로는 다른 곳도 커맨드를 위한 오브젝트(ID3DCommandList같은)나 파이프라인스테이트같은 것도 필요하게 됩니다만, 어디까지나 기본적인 초기화 같은 것으로, 이 3가지의 오브젝트를 생성해봅시다.

\*최소한 필요한 헤더include와 라이브러리의 링크

우선 기본이 되는 헤더가 <d3d12.h> 입니다. 여기에서 DXGI를 제어하기 위해 필요한 헤더<dxgi1\_6.h>를 인클루드합니다. 덧붙여 DXGI는 버전마다 헤더의 이름이 다르기 때문에 (예를 들면 <dxgi1\_4.h>같은)용도에 맞춰서 선택합시다. 본서에서는 우선

//include 헤더

를 추가합니다

여기에서 DXGI에 대해 모르는 사람들을 위해 조금 보충해봅시다. DXGI(DirectX Graphics Infrastructure) 라는 것은, Direct3D API 보다도 더욱 드라이버쪽에 가까운, 디스플레이로의 출력에 직접 관계하는 기능을 제어하기 위한 API세트 입니다. 그림 3.3을 봐주세요.

어플리케이션 ========🡺

DXGI=>하드웨어/드라이버

=> Direct3D => DXGI => 하드웨어, 드라이버

그림3.3 DXGI 와 Direct3D, 어플리케이션의 관계도

DXGI는 하드웨어나 드라이버에 가깝기 때문에, 본래는 Direct3D을 통해서 조작해야하는 것입니다만, 디스플레이의 열거나 화면 어플리나 유저쪽에서 DXGI에 직접 명령하지 않으면 안되기 때문에, 이처럼 구조가 되어있습니다.

\*\*DXGI1.6의 주의점

또한, DXGI1.6 이라면 Visual Studio Graphics Debugger가 오동작하는 경우가 있기 때문에, 그 경우에는 DXGI1.4를 선택하면 해소할 수 있습니다.

다음의 링크의 설정을 진행해봅시다. 헤더에 맞는 라이브러리를 링크합니다만, 우선 이 2가지로 충분합니다. 인클루드의 직후정도에 기술해둔다면 좋습니다.

//스크립트

여기까지 됐다면, 드디어 Direct3D의 기본부분의 초기화에 들어갑니다

\*기본오브젝트의 생성

//스크립트

여기서, 우선은 디바이스 (ID3D12Device) 오브젝트를 만들어봅시다. 여기서는 D3D12CreateDevice()함수를 사용합니다. 이 함수는 이하처럼 인수를 얻습니다.

제1 인수는 어댑터를 나타내는 포인터입니다. Nullptr 로 해두면, 자동으로 어댑터(그래픽스드라이버) 가 선택됩니다, 사용하시는 PC에서 사용되고있는 그래픽스보드가 1종류만이라면 문제 없습니다만, 복수의 그래픽스보드가 들어가있는경우 (예를 들면 노트PC로 Intel HD Graphics 와 NVIDIA 또는 AMD의 그래픽스보드가 내장되어있는 경우) 에는, 자동으로 선택되는 경우가 용도에 대해 최적이라고는 할 수 없습니다. 이 경우에는 대처가 필요하게 됩니다만, 그 방법에 대해서는 후술합니다.

제2인수는, 열거형으로 D3D\_FEATURE\_LEVEL 이라는 것이 있고, DirectX12의 경우, 집필시점에서는 D3D\_FEATURE\_LEVEL\_12\_1이 최고 레벨입니다. 그런데, 선택한 드라이버가 대응하지 않으면, 이 함수는 실패합니다. 혹시 디바이스 불러오기가 실패한다면 피쳐레벨을 내리던가, 다른 드라이버를 지정하는 것이 됩니다.

마지막의 2가지 인수는, 마지막의 포인터의 주소(디바이스의 실체)는 물론, REFIID라는 것이 조금 알기 어렵습니다. 이것은 얻고싶은 오브젝트의 형을 식별하기 위한 ID입니다만, 본래라면 포인터의 형에서 계산하는 것입니다. 그런데, 자신이 그 계산을 처리하는 것은 귀찮고, 또 별 의미도 없기 때문에 IID\_PPV\_ARGS라는 매크로를 사용합니다. 이 IID\_PPV\_ARGS라는 매크로는 매우 사용빈도가 높기 때문에, 제대로 사용하기 쉽도록 해봅시다.

\*\*매크로가 없는 경우

혹시 매크로를 사용하지 않는다면, REFIID는 \_\_uuidof(\*\*변수명)으로 취득가능합니다.

IID\_PPV\_ARGS매크로는, 인수에 포인터의 주소를 넘기면 REFIID와 포인터의 주소에 해석됩니다. 예를들면

D3D12CreateDevice(nullptr, D3D\_FEATURE\_LEVEL\_12\_1, IID\_PPV\_ARGS(&\_dev));

처럼 마지막의 2인수 (REFIID와 포인터의 주소)부분에 IID\_PPV\_ARGS매크로를 사용합니다. 여기까지 됐다면 D3D12CreateDevice() 함수의 불러오기는 완료입니다

\*\*공식 사이트의 정보를 반드시 참조합시다

D3D12CreateDevice()함수에 한하지 않고, DirectX를 사용한 프로그래밍 때에, 함수나 클래스, 구조체의 이용 때에, 반드시 공식사이트를 알아보고 최신 사양을 확인하도록 합시다.

<https://docs.microsoft.com/ja-jp/windows/win32/direct3d12/direct3d-12-graphics>

다만, 일부의 그래픽스보드(드라이버)에 따라서는, 지정한 피쳐레벨(D3D\_FEATURE\_LEVEL\_12\_1)에서는 불러오기에 실패할 가능성이 있습니다. 그 경우에는 피쳐레벨을 하나씩 내리면서 합치하는 설정을 찾을 필요가 있습니다. 그 때문에, 조금 귀찮습니다만

D3D\_FEATURE\_LEVEL levels[] = {

D3D\_FEATURE\_LEVEL\_12\_1,

D3D\_FEATURE\_LEVEL\_12\_0,

D3D\_FEATURE\_LEVEL\_11\_1,

D3D\_FEATURE\_LEVEL\_11\_0,

};

와 같은 배열을 만들어두고, for루프로 디바이스생성이 가능한 피쳐레벨을 찾으면서 Direct3D디바이스의 초기화를 처리하도록 해봅시다.

//direct3D 디바이스 초기화

여기서는 [어떤것도 안돼]였던 경우의 대처를 쓰지 않았습니다만, 그 때는 \_dev가 nullptr이 되기 때문에, 거기서 어플리케이션을 종료시키면 좋습니다.

\*그래픽스보드가 여러 개 끼워져 있는 경우

혹시 복수의 그래픽스보드가 끼워져있고, 그 때 특정의 그래픽스보드를 어댑터에 하고싶을 경우는 어떻게 하면 좋을까요? 아까처럼 D3D12CreateDevice() 함수의 제1인수를 nullptr로 해버리면, 예기한 그래픽스보드가 선택된다고는 할 수 없습니다. 거기서, 어댑터를 명시적으로 지정할 필요가 나옵니다.

\*\*이것은 어디까지나 하나의 예입니다

실제로는, 여기서 설명하는 방법은 적절한 것이 아닙니다만, 알기 쉬움과 실장의 간단함에서 생각한 결과입니다. 실제로 릴리스하는 경우는 nullptr의 쪽이 적절하기 때문에, 어디까지나 참고과정으로 생각해주세요.

우선, 어댑터를 열거하기 위한 DXGIFactory오브젝트를 생성합니다. 이번에는 오브젝트 변수를 IDXGIFactory6\* 형으로 하기 때문에, CreateDXGIFactory1() 함수를 사용해 생성합니다.

//auto result

여기서 DXGIFactory 오브젝트가 \_dxgiFactory 에 들어갑니다.

\*\* IDGIFactory 의 버전

또한, 여기서 실패하는 경우는 변수의 형을 IDXGIFactory4\*로 해서, CreateDXGIFactory()함수를 불러온다면 성공하는 경우가 있습니다.

다음으로, 이용가능한 어댑터를 열거하고, 열거된 어댑터를 std::vector 에 넣어봅시다.

//어댑터의 열거용 스크립트

여기서, 어댑터를 식별하기 위한 정보 (DXGI\_ADAPTER\_DESC 구조체)를 루프로 습득합니다. DXGI\_ADPATER\_DESC 구조체에는 Description 멤버 변수가 있고, 거기에는 어댑터의 이름이 격납되어 있습니다. 이번에는 그것을 이용해서, 특정의 어댑터를 찾아내기 입니다.

//스크립트

이 프로그램은, 이름에 “NVIDIA”가 포함되어있는 어댑터오브젝트를 찾아서 tmpAdapter에 격납하고 있습니다. 그 후, D3D12CreateDevice() 함수를 불러올 때 제 1인수에 없던 경우에는 nullptr 가 격납되었기 때문에 디폴트 어댑터가 사용되고 있습니다.

여기서 어댑터 명이 “NVIDIA”로 하고 있는 것은, 필자가 지금 사용하고 있는 PC의 그래픽스보드가 NVIDIA이니까, 라는 이유에 지나지 않습니다. 그 때문에, 이 프로그램은 실제로 넓게 이용하는 것을 생각하면 그다지 좋지 않은 예가 되겠습니다. 그 때문에, 혹시 제품판 같은 넓게 배포하는 어플리케이션으로서 사용할 때에는, 예를 들면 [기동전의 옵션화면을 만들어 두고, 어댑터에 무엇을 사용하는가 선택시킨다]같은 대처가 필요할지도 모릅니다.

\*HRESULT형

조금 이야기가 바뀝니다만, DirectX의 API의 대부분은, 결과를 HRESULT라는 형으로 반환합니다. HRESULT형은, 성공시에 S\_OK라는 수치가 되고, 실패시에는 S\_OK이외의 특정의 수치가 됩니다. 특히 초심자의 경우에는 반환수치의 체크를 빠뜨리지 않도록 해둡시다.

실제로, 여기까지 설명한 것을 기반으로 어댑터와 디바이스를 작성해봅시다. 2가지의 처리가 끝났을 때에 브레이크 포인터를 두고, \_dxgiFactory 오브젝트(IDXGIFactory6\*형) 와 \_dev오드브젝트(ID3D12Device\* 형)이 nullptr이 아닌 것을 확인해주세요.

무사히, 확인 됐다면, 우선 [기본적인 초기화는] 종료입니다.

* 1. 화면색의 클리어

현 단계에서는, DirectX가 실제로 화면표시를 제어하고있는가 아닌가 알수없습니다. 그래서 여기서부터는, DirectX가 화면표시를 하고 있는 것을 알도록 화면의 색을 변경해봅시다.

하지만, 이 작업은 의외로 수고가 드는 것이기에, 이해해주세요,

이번의 처리를 실행하기 위해 필요한 것은, 아래와 같습니다.

* 커맨드리스트: ID3D12GraphicsCommandList
* 커맨드큐: ID3D12CommandQueue
* 스왑체인 : IDXGISwapChain4
* 렌더타겟
* 디스크립터히프 : RTV 용 히프

그래도, 이것들의 실장을 하기 전에 DirectX12에 걸친 [커맨드리스트]에 대해 해설하지 않으면 안되겠네요.

3.3.1 커맨드리스트란

커맨드리스트란, 간단히 말하면 [GPU에 대한 명령을 종합하기 위한 오브젝트]입니다. DirectX11에서는 DeviceContext 오브젝트에 그때그때 명령을 보냈으면 좋았지만, DirectX12에서는, 그것들의 명령을 커맨드리스트 (ID3D12GraphicsCommandList\*형)으로서 하나로 모아서, 한번에 GPU에 보내는 사양으로 되어 있습니다.

[GPU에 대해 명령을 리스트로 쌓아서, 한번에 실행해버린다] 라면 알기 어렵습니다만, 함수오브젝트를 std::list을 담아두고, 그 후에 하나로 실행하는 것이라 생각하면 알기 쉬울지도 모릅니다. 이 구조의 자금 귀찮은 점은, 명령이 지연실행되는 점입니다. DirectX11까지는, GPU에 대해 명령하는 함수를 부른다면 기본적으로 그 자리에서 실행되고 있습니다만, DirectX12에서는 모아두고 한번에 실행하기 때문에, 예를 들면 제 8장에서 후술하는 [머티리얼루프]에 때 같은 귀찮은 경우가 될 지도 모릅니다.

\*지연실행

커맨드리스트와 지연실행을 유사체험하기 위해, std::vector과 std::function 을 사용해 실험해봅시다. 어디까지나 이해를 위한 실험이니까 DirectX12 프로그래밍과는 직접관계는 없습니다.

DirectX어플리케이션과는 다른 테스트 프로젝트를 만들고, 다음의 프로그램을 실행하면, 어떤 순서로 출력될까요?

#include <vector>

#include <functional>

#include <iostream>

using namespace std;

int main()

{

std::vector<std::function<void(void)>> commandlist; //커맨드리스트를 본뜬 것

commandlist.push\_back([]() {cout << "GPU Set RTV-1" << endl; });

cout << "CPU Set 명령-2" << endl;

commandlist.push\_back([]() {cout << "GPU Set RTV-3" << endl; });

cout << "CPU Set 명령-4" << endl;

commandlist.push\_back([]() {cout << "GPU Set RTV-5" << endl; });

cout << "CPU Set 명령-6" << endl;

cout << endl;

//커맨드큐의 ExecuteCommand를 본뜬 처리

for (auto& cmd : commandlist)

{

cmd();

}

getchar();

return 0;

}

지연실행의 생각법에 익숙해지지 않으면, 1-2-3-4-5-6의 순으로 출력된다고 생각할지도 모릅니다. 하지만, 실제로는 다음의 결과가 됩니다

2

4

6

1

3

5

이 결과를 보면 알수 있듯이, 지연실행되는 처리는 후에 한꺼번에 실행되고, [쓴 순서대로는 ]실행되지 않습니다. 커맨드리스트를 사용할 때에도 같은 것이 일어나기 때문에, 잘 외워둡시다.

* + 1. 커맨드리스트의 작성과 커맨드어로케이터

커맨드리스트 (ID3D12GraphicsCommandList\*형) 을 사용할 때는 커맨드어로케이터(ID3D12CommandAllocator\*형) 이라는 것이 필요합니다. 서로의 관계성은 조금 알기 어렵스빈다만, 실제의 코드를 보고 이해해봅시다.

먼저 각각 선언합니다. 커맨드리스트를 사용하기 위해 커맨드 어로케이터가 필요하기에, 각각 선언합니다.

//선언

다음은 그것들을 생성합니다. 먼저 기술한 [디바이스나 IDXGIFactory 생성]의 후에 불러와주세요.

//result

여기까지 됐다면, 실행하고 각각 result 의 내용이 S\_OK인 것을 확인해보시다

이것으로 우선 커ㅁ내드리스트와 커맨드어로케이터가 생성되었기 때문에, 여기부터는 ID3D12GraphicsCommandList와 ID3D12CommandAllocator과의 관계에 대해 조금 해설해보겠습니다.

우선, ID3D12GraphicsCommandList는, GPU에 명령하기 위한 메소드를 가진 인터페이스입니다. 명령이 되는 메소드의 예로서는

* DrawInstanced() : 점, 선, 폴리곤을 묘화
* DrawIndexedInstanced() : 인덱스가 븥은 선이나 폴리곤 묘화
* ClearRenderTargetView(): 렌더 타겟의 클리어
* Close() : 명령을 닫습니다(필수)

같은 것들을 들 수 있습니다. 실제로는 더욱 많이 있습니다만, 그것들은 적절히 해설해나가면서, 여기서는 [ID3D12GraphicsCommandList는 GPU에 대한 명령의 인터페이스로 되고있다]라는 것을 기억해둔다면 좋겠습니다.

다음은 ID3D12CommandAllocator입니다만, 사실은 이거야 말로 커맨드리스트의 본체입니다.

ID3D12GraphicsCommandList는 어디까지나 명령을 위한 인터페이스이므로, 실제의 [명령 오브젝트]는 ID3D12CommandAllocator에 push\_back()되어있다는 이미지를 가지면 좋을겁니다.

그림3.4

그림 3.4가 가리키는 것처럼, ID3D12CommandAllocator는 명령의 내용을 쌓아가는 메모리 영역입니다. 그리고 ID3D12GraphicsCommandList는 ID3D12CommandAllocator에 대응한 형식으로 명령을 쌓기 위한 인터페이스라 할 수 있습니다. 명령의 본체가 실질적으로는 ID3D12CommandAllocator쪽에 있는 것도 알 수 있습니다.

그러나, 커맨드리스트는 명령을 쌓고 있는 것으로, 이대로는 GPU쪽은 아무것도 해주지 않습니다. 쌓은 명령을 실행하는 커맨드큐(ID3D12CommandQueue\*형) 이라는 오브젝트를 사용할 필요가 있습니다.

* + 1. 커맨드큐

커맨드큐는, 커맨드리스트를 이용해 쌓아올린 명령세트를 GPU가 순차실행하기 위한 것입니다. 큐(Queue) 라는 이름부터 상상이 되듯이, 명령은 넣은 순으로 실행되어갑니다.

\*\*마지막에 Close가 필요

[CPU가 차례대로 명령을 넣고, 그것을 GPU가 흘러가듯이 차례차례 실행한다]라는 이미지를 가질지도 모릅니다만, 실제로는 조금 다릅니다. GPU가 명령 세트를 실행하기 위해서는, 마지막에 [Close명령] 을 넣지 않으면 GPU는 실행가능한 상태가 되지 않습니다. [입력가능상태]와 [실행상태]가 공존하지않는 사양이라 생각할 수 있습니다.

그러면, ID3D12CommandQueue 오브젝트를 만들어봅시다. 우선 변수를 선언합니다

//선언

다음으로 ID3D123Device::CreateCommandQueue()메소드를 이용한 커맨드큐의 실체를 작성합니다. 그때, D3D12\_COMMAND\_QUEUE\_DESC 구조체에 의해 여러 설정을 진행합니다.

또한, 각 설정은 커맨드대로입니다.

//커맨드큐스크립트

\*\*ㅇㅇ\_DESC구조체

DirectX에서 오브젝트로 만들 때는, 이처럼 ㅇㅇ\_DESC라는 구조체에서 설정하는 것이 많기때문에 알아둡시다. 귀찮습니다만, 구조체나 함수를 이용할 때는 반드시 공삭사이트의 정보를 보도록 합시다.

여기까지 됐다면, result에 S\_OK가 반환하도록 확인해둡시다.

* + 1. 스왑체인

색을 변경하기 위해서는, 아직 설정하지 않으면 안되는 것이 있습니다. 그것은 스왑체인(Swapchain) 입니다. 스왑체인은 [더블버퍼링]을 하기 위한 구조입니다

\*\*더블버퍼링

경우에 따라서는 [더블], 즉 [버퍼가 2개] 라 할 수 있습니다만, 본서에서는 어디까지나 2개로서 해설합니다

\*더블버퍼링

여러분이 보고 있는 윈도우에는, 그림이 그려져있거나 문자가 쓰여져있거나 합니다. 그처럼 윈도우 내부에 여러가지 것을 표시하거나, 갱신하거나 할 수 있다는 것은, 화면과 끈을 붙일 수 있는 버퍼(일시적으로 데이터를 쌓아두는 영역)이 반드시 존재합니다.

화면을 갱신하기 위해서는 [프로그램이 화면과 끈을 붙인 버퍼를 직접 바꿔쓴다]라는 방법이 생각날지도 모릅니다. 확실히 그것만으로 열면 어떤 문제도 없다고 생각할수도 있습니다만, 화면을 보고 있는 사람은 연계해서 화면을 관찰하고 있기 때문에, 버퍼를 바꿔쓰고 있는 도중 경과가 보이게 되버립니다. (그림 3.5). 보이고 있는 시간은 아주 일 순간입니다만, 인간의 감각은 매우 날카롭고, [깜빡깜빡] 으로 지각하는 것입니다.

그림3.5 묘화도중의 화면이 표시되는 모습

거기에서 생각해낸 것이 더블버퍼링이라는 방식입니다. 이 방식은, 간단히 말하면 [화면]을 2 가지, 확실히는 [화면으로서 할당하는 메모리 공간(버퍼)]를 2개 준비합니다. [그림3.6]. 또한, 이 메모리 공간을 렌더타겟이라 부릅니다.

그림 3.6 2가지의 렌더타겟 (앞화면, 뒷화면)

2가지의 렌더타겟 중, 그 순간에 디스플레이에 표시되고 있는 쪽을 [앞화면], 표시되고 있지 않은 쪽을 [뒷화면]으로 하면, 더블버퍼링으로 실제로 묘화처리를 행하는 쪽은 뒷화면입니다. [뒷화면이라면, 모처럼 묘화해도 디스플레이에 보이지 않는지]라 생각하신다면 안심하세요. 바꿔쓰는게 끝났다면, 앞화면과 뒷화면을 바꿔서 [스왑], 뒷화면을 [디스플레이에 표시하는쪽]으로서 할당하는 것으로 묘화한 내용이 디스플레이에 표시됩니다.

그림3.7 앞화면이 표시되고, 뒷화면은 묘화된다.

이어서, (아까 앞화면이었던) 뒷화면에 묘화처리를 하고, 묘화가 완료했다면 다시 앞과 뒤를 바꿉니다, 이것으로 사람의 눈에는 [칠하기완료]의 화면만 보이고, 깜빡깜빡이는 것을 방지하는 것이 가능합니다.

\*스왑체인의 개요

더블버퍼링을 실현하는 것은 스왑체인의 역할입니다. 스왑체인은, 스왑(바꾸기)을 행하는 화면용의 비디오메모리를 2개 이상 확보하고, 화면으로의 묘화를 플래쉬(한번에 입력한다)하는 타이밍으로 비디오메모리의 참조처를 전환해줍니다.

\*\*윈도우 모드의 경우

여기까지의 설명은, 사실은 풀스크린모드입니다.

윈도우모드의 움직임이 신경쓰이는 사람을 위해 조금이나마 보충한다면, 윈도우모드의 경우는 플립과 동시에 윈도우 영역으로 버퍼의 전송이 이뤄집니다. 그 때문에 윈도우모드와 풀스크린모드에서는 움직임이 다를수도 있습니다만, 본서에서는 디버그의 쉬움을 우선해서 윈도우모드로 하고 있습니다.

\*스왑체인의 생성

그러면 실제로 스왑체인오브젝트 (IDXGISwapChain1\* 형)을 생성해봅시다. 이번에는, IDXGIFactory::CreateSwapchainForHwnd() 메소드를 사용합니다.

IDXGIFactory::CreateSwapchainForHwnd()메소드의 인수는 이하가 되겠습니다.

//HRESULT

제1인수는 pDevice라는 이름으로, 이전에 만든 ID3D12Device 오브젝트를 넣고싶게 됩니다만, 시험삼아 ID3D12Device오브젝트를 넘기면 파라미터에러가 나옵니다. 그러면 무엇을 넘기면 좋을까라면, 아까 만든 ID3D12CommandQueue 오브젝트입니다.

\*\*Device인데 왜 CommandQueue?

제1인수가 Device로 되어있는데 실제로는 ID3D12CommandQueue를 넘기는 것은, DXGI인터페이스가 DirectX11과 DirectX12로 공통이기 때문이라 추측됩니다.

제4인수, 제5인수는 nullptr로 좋기 때문에, 제3인수의 DXGI\_SWAP\_CHAIN\_DESC1에 대해 설명해봅시다. DXGI\_SWAP\_CHAIN\_DESC1 구조체는 다음처럼 정의되어 있습니다.

//typedef

실제로 스왑체인생성 프로그램을 쓴다면, 이하처럼 됩니다.

//DXGI\_SWAP\_CHAIN\_DESC1

여기까지 됐다면 실행하고, result에 S\_OK가 반환되는 것을 확인해주세요. 성공했다면, 스왑체인이 사용할수있게된, 즉 화면을 위해 버퍼가 BufferCount 만큼 확보됐다는 것이 됩니다.

그렇다고 해서, 아직 백버퍼를 확보해 스왑체인오브젝트를 만든 것뿐입니다. 화면의 표시를 위해서는, 아직아직 해야할 것이 남아있습니다.

* + 1. 렌더타겟뷰(RTV)

전술한듯이 버퍼는 그대로라면 그 역할을 다할 수 없습니다. 이대로라면, 만들어진 버퍼를 바꿔넣을 수 있을 지라도 내용을 바꿔쓰는 것은 안되기 때문입니다. 클리어는 물론 안됩니다.

버퍼의 내용을 바꿔쓰기 위해서는 렌더타겟뷰(RTV)라는 뷰가 필요합니다.

\*버퍼와 뷰

여기서, 버퍼와 뷰라는 용어에 대해 설명합니다.

버퍼(GPU리소스)란 [메모리의 덩어리]로, 이하처럼, CPU와 GPU와의 여러 주고받기에 사용되는 것입니다.

* 초점버퍼 (4.3절 참조)
* 인덱스버퍼(4.11.2항 참조)
* 정수(constant) 버퍼(6.2절 참조)
* 텍스쳐버퍼(5.5절참조)
* 심도(depth)버퍼(7.9절참조)

그렇게, 작성한 버퍼의 용도를 정의한 것이 뷰입니다. 정수버퍼나 텍스쳐버퍼 같은, 몇 개의 버퍼에서는 뷰의 정의가 필수로 되어있습니다.(각각 해설할 때 설명하겠습니다)

\*\*어째서 버퍼와 뷰를 나누고 있는가

경우에 따라서는, 같은 데이터라도 사용상황에 따라서 보는 법을 바꾸지 않으면 안되는 경우도 있기 때문에, [데이터 그 자체(버퍼)]와 [그 해석(뷰)]를 나눠서 관리하고 있는 것이라 생각할 수 있습니다.

\*디스크립터

여기까지 몇번인가 나온 디스크립터라는 개념에 대해 알아둡시다.디스크립터(Descriptor)란, 일본어로 [기술자] 가 됩니다만, 명확한 의미는 알기 어렵기 때문에, 본서에서는 디스크립터라 부르기로 했습니다.

디스크립터란, [GPU리소스 (메모리덩어리)의 용도나 사용법에 대해 설명하고 있는 데이터] 입니다. 현명한 독자 여러분은, [아까 나온 뷰가 아닌가]라 생각할 수도 있습니다. 어떤 의미에서는 정말로 그렇다고 할 수 있습니다만, 디스크립터는 아래에 가리키는 것처럼, 각종 뷰에 더한 샘플러를 포괄하는 듯한 보다 넓은 개념입니다.

* ShaderResourceView(SRV) : 텍스쳐버퍼에 대해 설명
* ConstantBufferView(CBV) : 정수버퍼에 대해 설명
* UnorderedAccessView(UAV) : 컴포트세이더에 사용하는 버퍼에 대해 설명
* Sampler : 샘플러에 대해 설명
* RenderTargetView(RTV) : 렌더타겟뷰에 대해 설명
* DepthStencilView(DSV) : 심도스텐실뷰에 대해 설명

\*\*뷰에 관한 DirectX11 과의 차이

DirectX12에서는 뷰를 디스크립터로스 다루기 때문에, DirectX11 까지 처럼 텍스쳐나 정수버퍼같은 [뷰오브젝트]영역을 [직접]만들거나 사용하거나 하는 경우가 없습니다.

\*디스크립터힙

여기서 드디더, 디스크립터힙 이라는, 디스크립터를 하나로 모아 다루는 구조에 대해 해설합니다. 매우 귀찮은 것입니다만, 이미지만으로 알아두세요.

디스크립터는, 각종뷰와 샘플러를 포괄하는 듯한 개념입니다만, 간단히 사용되는 것은 전혀 아닙니다. 물론 뷰 같은 디스크립터 오브젝트 자체도 데이터의 덩어리이며, 메모리상에 존재하는 것도 됩니다만, DirectX12에서는 하나하나의 디스크립터를 정의하기 전에, 복수의 디스크립터를 입력하는 것이 가능한 디스크립터힙 이라는 메모리영역을 미리 확보하는 것이 되겠습니다.

즉, 그림3.8처럼 디스크립터힙에 복수의 디스크립터 정보를 입력해가면 생각해둔다면 좋습니다. 그래서 그 입력한 정보의 용도를 (예를 들면[뷰] 인 것), GPU에 전달하고 있습니다.

\*디스크립터힙을 작성하기

그러면, 실제의 코드로 돌아가봅시다. DirectX12에서는, 뷰는 디스크립터힙에서 만들어졌기 때문에, 우선은 디스크립터힙(ID3D212DescriptorHeap\*형)을 작성합니다.

디스크립터힙은 ID3D12Device::CreateDescriptorHeap() 메소드를 이용해서 생성합니다. 힙을 작성하기 위해 사용하는 여러가지의 설정은, D3D12\_DESCRIPTOR\_HEAP\_DESC라는 구조체를 이용해 다음처럼 기술합니다

//D3D12\_DESCRIPTOR\_HEAP\_DESC

D3D12\_DESCRIPTOR\_HEAP\_DESC 의 멤버 변수를 설명해봅시다.

우선 Type은, [어떤 뷰를 만드는가]를 지정하기 위한 열거자 (D3D12\_DESCRIPTOR\_HEAP\_TYPE)입니다. 이번은 렌더타겟뷰(RTV) 를 위한 D3D12\_DESCRIPTOR\_HEAP\_TYPE을 지정합니다만, 이외에도 용도에 따라 표3.1처럼 수치를 지정 가능합니다.

표3.1

|  |  |
| --- | --- |
| 열거치 | 용도 |
| D3D12\_DESCRIPTOR\_HEAP\_TYPE\_RTV | 렌더타겟뷰(RTV) |
| D3D12\_DESCRIPTOR\_HEAP\_TYPE\_CBV\_SRV\_UAV | 정수버퍼뷰(CBV), 텍스쳐버퍼(SRV), 컴퓨트세이더용버퍼(UAV) |
| D3D12\_DESCRIPTOR\_HEAP\_TYPE\_DSV | 심도스텐실뷰 |

NodeMask는, 본래 복수의 GPU가 있는 경우에 식별을 하기 위한 비트플래그입니다. 하지만, 이번에는 GPU를 하나만 사용하는 상정이기에 0으로 해둡시다.

NumDescriptors는, 디스크립터의 수를 나타내는 멤버입니다. 더블버퍼링의 경우, 앞화면 뒷화면 각각의 버퍼에 대응하는 뷰이기에, 2를 지정합니다.

마지막의 Flags는, 이 [뷰에 해당하는 정보]를 세이더쪽에서 참조할 필요가 있는가 아닌가를 지정하는 열거자입니다. 텍스쳐버퍼(SRV)나 정수버퍼(CBV)라면 D3D12\_DESCRIPTOR\_HEAP\_TYPE\_SHADER\_VISIBLE 입니다만, 이번에는 세이더쪽에서 보일 필요가 없기 때문에, D3D12\_DESCRIPTOR\_HEAP\_FLAG\_NONE 을 지정합니다.

여기까지 됐다면 거의 끝났을 지도 모릅니다. 다음은 평소처럼 IID\_PPV\_ARGS 매크로와 함께 ID3D12Device::CreateDescriptorHeap() 메소드를 불러옵니다.

//ID3D12DescriptorHeap\* rtvHeaps

디버그실행해서, result가 S\_OK가 되는가 확인해봅시다. 혹시 실패했다면 어딘가 틀린것입니다. 다시 한번 고쳐봐주세요.

\*스왑체인의 메모리와 끈을 이은다

이것으로 뷰가 만들어졌고, 라고 말하고 싶습니다만, 사실은 아직 [뷰 용의 메모리 영역을 확보했다]에 지나지 않습니다.

거기서 다음은, 디스크립터와 스왑체인상의 버퍼(백버퍼) 와의 관련잇기를 진행해봅시다. 그렇게 위해서는 ID3D12Device::CreateRenderTargetView() 메소드를 사용하는것입니다만, 백버퍼도 디스크립터도 각각 2가지씩이기에, 2가지 각각을 관련잇도록 프로그램을 써갑니다.

우선, 백버퍼의 수만 설정하지 않으면 안되기 때문에, 이번에는 for문의 루프보다 ID3D12Device::CreateRenderTargetView() 메소드를 불러옵니다. 백버퍼의 수는 2라 바로 수치를 사용해도 좋고, 힙을 만든 때의 파라미터(NumDescriptors멤버)를 사용해도 좋습니다. 혹시 힙을 만들었을때의 파라미터가 스코프 내에 없으면, 스왑체인오브젝트의 메소드인 IDXGISwapchain::GetDesc() 메소드라도 습득가능합니다.

//DXGI\_SWAP\_CHAIN\_DESC

for루프의 큰 축은 이하가 되겠습니다

//std::vector

다음으로 for문의 내용을 실장해봅시다. ID3D12Device::CreateRenderTargetView() 메소드로, 스왑체인 안의 버퍼와 뷰를 관련짓습니다. 또한 스왑체인의 안의 메모리는 IDXGISwapChain::GetBuffer() 메소드로 습득합니다

//result

이렇게 하는 것으로, backbuffer의 안에 스왑체인상의 백버퍼를 취득가능합니다. 더불어 제1인수에는 [몇매째의 백버퍼인가]를 지정합니다. 이번에는 전부의 백버퍼에 대해 처리를 하기 때문에, 인덱스를 그대로 지정하고 있습니다.

이렇게 받은 버퍼를 ID3D12Device::CreateRenderTargetView() 메소드의 인수로서 넘기고, 렌더타겟뷰를 생성합니다. 이 메소드의 인수는 이하처럼 선언되어 있습니다.

//void CreateRenderTargetView

제1인수는 앞서 취득한 버퍼입니다. 제2인수는 공식사이트에 따르면 [nullptr를 대입하면 밉0번의 서브리소스에 엑세스 가능]합니다. 이번, 렌더타겟은 밉맵(오브젝트의 크기의 단계에 따라 다른 사이즈의 텍스쳐를 준비할 것)과는 관계없기 때문에 nullptr로 좋습니다.

제3인수의 디스크립터힙핸들은, 디스크립터힙상에 걸친 뷰의 주소 같은 것입니다. ptr멤버를 갖고 잇고, 여기에 주소가 들어갑니다. 디스크립터힙핸들을 취득하는 것은, ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart() 메소드를 불러냅니다.

그러므로, 여기까지의 내용에서 이하의 처럼 코드를 쓴다면 좋다고 생각합니다.

//\_dev->

그런데, 이 프로그램에서는 크게 문제가 있습니다. 함수명에서 상상가능하듯이, ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart() 메소드에 취득가능한 것은 어디까지나 [선두의 주소] 입니다. 0번째는 아직 괜찮습니다만, 1번째 이후의 디스크립터를 취득하는 것은 하나만큼 뒤로 비켜줘야할 필요가 생깁니다.

일반적인 포인터형의 변수처럼, 포인터+i라 된다면 좋습니다만, 그렇게 안되는 이유가 2가지 있습니다.

* 뷰의 종류에 따라 디스크립터가 필요한 사이즈가 다르다(비키는 바이트수가 다르다)
* 넘기는 것에 사용하는 것은 핸들이므로, 주소 그 자체가 아니다.

//D3D12\_CPU\_DESCRIPTOR\_HANDLE

ID3D12Device::GetDescriptorHandleIncrementSize() 와는, 함수명이 나타나도록, 디스크립터 1개정도 사이즈를 반환하는 메소드입니다. 인수에는 디스크립터의 종류 (이번에는 렌더타겟뷰를 나타내는 D3D12\_DESCRIPTOR\_HEAP\_TYPE\_RTV) 를 지정하는 것으로, 종류에 맞는 사이즈를 얻을 수 있습니다. 얻은 사이즈를 핸들의 ptr멤버로 가산하는 것으로, 렌더타겟뷰가 생성됩니다.

\*\*다른 법으로 쓰기

여기까지 내용은, 다음처럼 쓰는 것도 가능합니다.

D3D12\_CPU\_DESCRIPTOR\_HANDLE handle = rtvHeaps->GetCPUDescriptorHandleForHeapStart();

for (int idx = 0; idx < swcDesc.BufferCount; ++idx)

{

handle.ptr += idx \* \_dev->GetDescriptorHandleIncrementSize(D3D12\_DESCRIPTOR\_HEAP\_TYPE\_RTV);

\_dev->CreateRenderTargetView(\_backBuffers[idx], nullptr, handle);

}

어느쪽도, 알기쉬운방법으로 괜찮습니다.

실제로는, 후에 루프코드에 조금 손을 대는 것이 됩니다만, 지금은 이대로 괜찮습니다. 히프조차 제대로 만들수 있으면 실패하는 것은 아니기 때문에 다음으로 진행합시다.

* + 1. 스왑체인을 동작시킨다

드디어, 다음은 메인루프안에 스왑체인에 동작해주는 것입니다.

여기서, 메인루프 안의 처리의 해설을 진행합니다. 다음 같은 흐름으로 처리가 진행됩니다.

1. 커맨드어로케이터와 커맨드리스트를 클리어(어로케이터를 리셋합니다)
2. [커맨드] 렌더타겟을 백버퍼에 세트
3. [커맨드] 렌더타겟을 지정색으로 클리어
4. [커맨드] 렌더타겟의 클로즈
5. 쌓인 커맨드를 커맨드리스트에 던진다
6. 스왑체인의 플립처리(present 명령)

사실은 이것만으로 모자랍니다만(문제가 발생합니다), 그것에 대해 후술하기로 하고, 우선 이 흐름을 빨리 만들어봅시다.

앞서 설명한 대로, 커맨드어로케이터는 명령오브젝트의 리스트처럼 존재합니다. 우선은, ID3D12CommandAllocator::reset() 메소드로 그 클리어를 해봅시다.

//result reset

다음은 여기에 명령 오브젝트를 쌓아가는것입니다.

\*렌더타겟의 설정