프레임 버퍼는 한 프레임에 출력 되어야하는 비디오 메모리의 영역에 있는 사각형 이미지

Direct3D에서 비디오 메모리로 내용을 쓰고 모니터가 읽어간다.

프레임 버퍼가 바뀌는데 렉이 없게 하기 위하여 이중 버퍼링이라는 2개 프레임 버퍼를 사용한다.

전면 버퍼 – 현재 화면에 출력되는 이미지

후면 버퍼 – Direct3D가 쓰는 다음에 출력할 이미지

Swap Chain – 순차적으로 연결된 프레임 버퍼들의 집합

기본 적으로 2개가 존재하며 보여지는 것이 전면 버퍼, 그 외가 후면 버퍼이다.

Presentation – 후면 버퍼의 내용을 전면 버퍼로 옮기는 것

COM (Component Object Model)객체

* DLL의 형태로 제공이 된다.
* 모든 Direct3D 객체는 COM객체이다.
* 외부로 노출된 인터페이스를 사용할 수만 있고 내부는 노출되어 있지 않다.
* 객체의 생성과 소멸이 일반적인 C++객체들과 다르다.

1. AddRef

인터페이스의 참조 카운터를 1 증가시킨다.

이 COM객체를 사용하고 있음을 나타낸다.

1. Release

인터페이스의 참조 카운터를 1 감소시킨다.

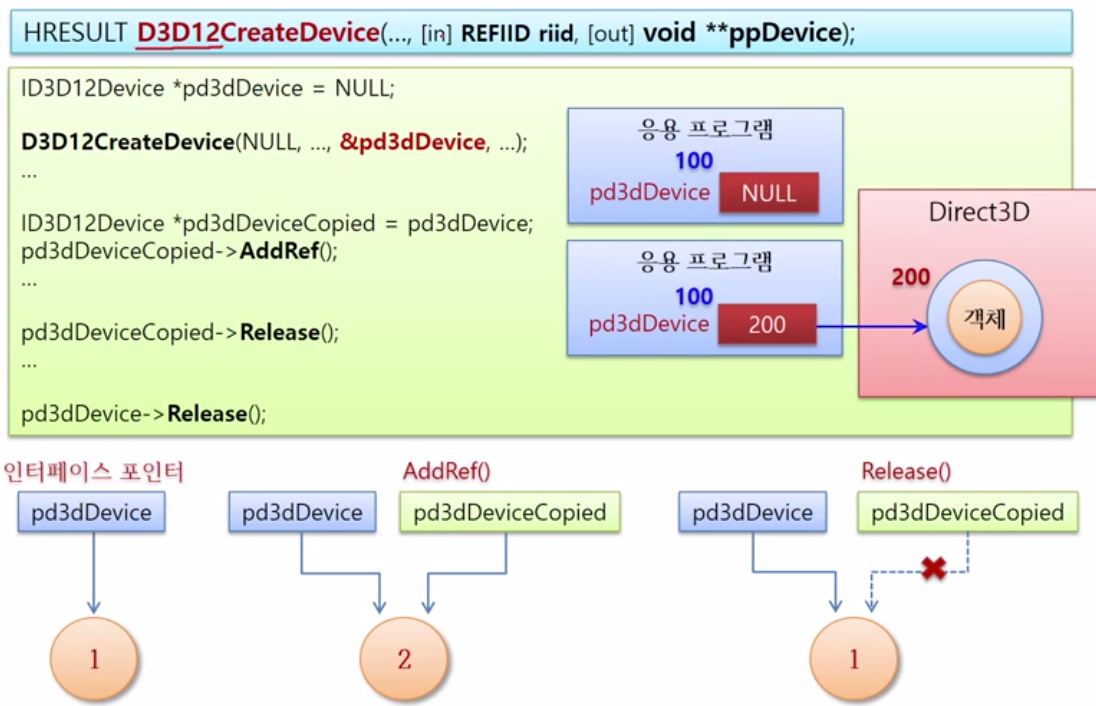
참조 카운트가 0이 되면 아무도 사용하지 않고 있는 것이다.

1. QueryInterface

이 객체가 어떤 인터페이스를 제공하는 가를 확인시켜준다.

COM객체의 생성과 소멸

* 참조 카운터가 0이되면 자동으로 소멸이 된다.
* 생성을 위한 API함수가 별도로 존재한다.
* New나 delete로 생성과 소멸을 관리하지 않는다.
* Create시 참조 카운터가 1이 되고 AddRef시 +1, Release시 -1이 된다.
* 참조 카운터가 0이되면 가비지 콜렉터가 COM객체의 메모리는 반납시킨다.



디바이스라는 COM객체를 생성하는 함수이다.

\*\*ppDevice는 디바이스의 포인터를 받기위한 포인터의 포인터 변수이다.

Pd3dDevice라는 인터페이스 디바이스 포인터 변수를 만들 후 NULL을 넣는다.

이 때 이 변수의 주소는 100이고 NULL이라는 주소를 가리키고 있는데 CreateDevice에 이 변수를 넣으면 새로 생성된 디바이스의 주소 값을 가리키게 할 수 있다.

그리고 pd3dDeviceCopied라는 새로운 인터페이스 디바이스 포인터 변수를 만들고 복사하면 AddRef를 사용하여 참조 카운터를 늘려줘야 한다.

복사본이 아닌 원본인 pd3dDevice는 가장 나중에 Release해주어야 한다.

* 즉 COM객체를 사용하기 위해선 인터페이스 포인터 변수를 선언 한 뒤 Create를 할 때 그 포인터 변수를 넘겨주어 사용해야 한다.
* GetDisplayModeList(디스플레이의 기능 [해상도, 색상 수 등..]을 가져오는 함수)는 포인터가 아니라 int변수를 선언하는 등.. COM객체가 아닌 경우 메모리 할당은 우리의 몫이다.

GUID

* COM객체들은 각각의 객체를 식별할 수 있는 식별자를 가진다. (128비트 정수 문자열)
* IID = 인터페이스를 구별하기 위한 GUID
* \_\_uuidof() = 인터페이스 자료형, 클래스 이름, 인터페이스 포인터에 대한 GUID 반환
* ComPtr = 인터페이스 포인터의 소멸자에서 release를 자동 호출해주는 스마트 포인터

그래픽스 프로그래밍 인터페이스

* DXGI (DirectX Graphics Infrastructure) = 디바이스의 열거, 모니터의 열거, present등.

DirectX 그래픽을 위한 기본적이고 공통적인 프레임워크 제공

DXGI\_SWAP\_CHAIN\_DESC 구조체 (Swap Chain)

테이블이(가) 표시된 사진

자동 생성된 설명

* GetBuffer = Swap Chain의 후면 버퍼를 반환
* GetBuffer(버퍼 인덱스, 후면 버퍼의 표면 종류, 담을 곳)
* Present = 렌더링 된 영상을 Display 모니터로 출력
* Present(0, 보여 줄 프레임) (0이 가장 빠르다)
* ResizeBuffers = Swap Chain의 후면 버퍼의 크기, 형식, 개수 변경
* 후면 버퍼의 모든 참조를 해제해야 가능하다
* ResizeTarget함수를 호출한 후에 호출
* 출력 윈도우 크기가 변경되면 윈도우 크기에 맞게 후면 버퍼 변경
* ResizeTarget = 바탕화면의 크기를 변경
* 윈도우 모드 일 때 = 출력 윈도우 크기 변경
* 전체 화면 모드 일 때 = 디스플레이 모드 변경

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명

Descriptor Handle

* Offset만큼 더해주면서 Descriptor의 시작위치를 알려준다.

Descriptor Heap (서술자 힙)

* 모든 리소스들을 서술자로 표현하기 위해 필요한 메모리들의 블록
* 이를 위해 Descriptor Handle을 사용한다.

CreateRenderTargetView

* 항상 리소스가 먼저 있어야 한다.
* 리소스가 있으면 리소스의 뷰를 만들기 위한 메모리가 필요하다 (Descriptor Heap)
* 리소스가 여러 개일 수도 있으므로 리소스 마다 Descriptor Heap을 만들어야 한다.
* Descriptor가 몇 개인지 지정해주고 Descriptor Heap을 만들어 준다.
* View란 리소스를 설명하는 내용이 담긴 구조체
* 즉, 똑 같은 리소스라도 View를 다르게 하면 다르게 읽힌다.
* VIEW\_DESC를 NULL로 하면 리소스와 같은 차원을 사용한다.

CreateDepthStencilView

* Depth/Stencil뷰를 만들기 위한 함수이다.
* VIEW\_DESC를 NULL로 하면 리소스와 같은 차원을 사용한다.
* Painter Algorithm을 사용하기 위하여 필요하다.

CreateCommittedResource

* 리소스를 생성하기 위한 함수
* Heap Properties는 Heap Type을 설정해주면 된다.
* Heap Flags는 Heap이 Texture를 가지는지, 어디에 공유가 되는지를 정한다.
* Resource State는 리소스의 상태이다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* Resource Description의 Dimension은 리소스가 Buffer인지 Texture인지 정한다.
* Render Target과 Depth Stencil로 사용하려면 Resource Flags을 설정해 줘야 한다.
* Texture는 반드시 Default Heap에 만들어야 하며 Upload Heap에 생성 불가하다.

HEAP TYPE

* 리소스를 만들 때 HEAP 종류를 정해줘야 한다.
* HEAP\_TYPE\_DEFAULT: GPU가 읽고 쓰기가 가능하며 CPU는 접근 불가 (대부분)
* HEAP\_TYPE\_UPLOAD: 업로드를 위한 CPU접근에 최적화 되어있다.
* HEAP\_TYPE\_READBACK: 읽기를 위한 CPU접근에 최적화 되어있다 (최대한 사용 안함)

기본적으로 리소스는 GPU만이 접근 가능하고 CPU는 접근이 불가능하다.

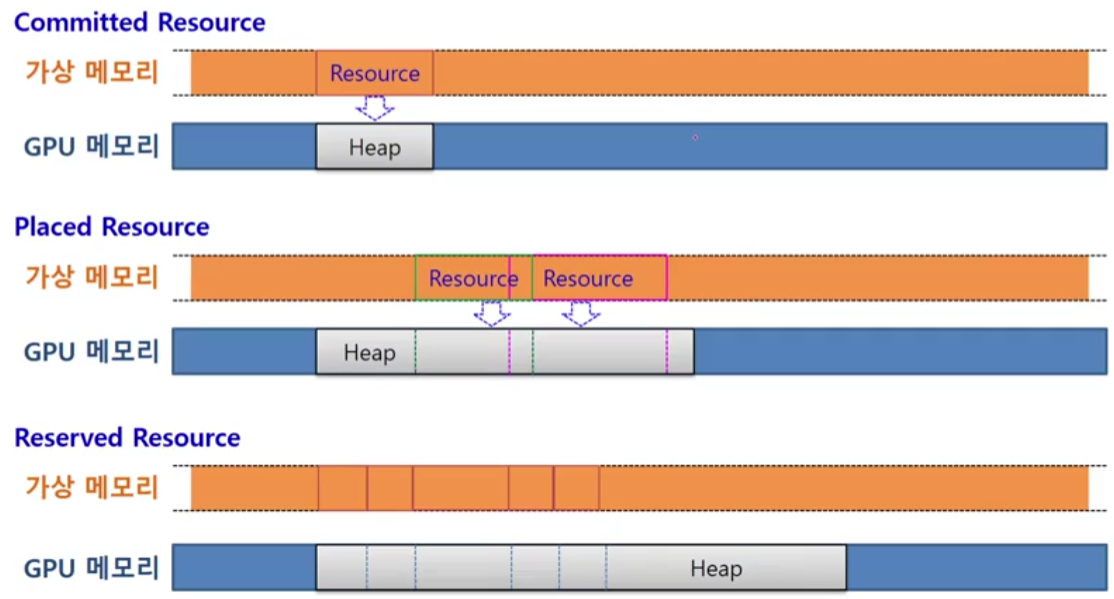
그래서 Default Heap을 만들려면 반드시 Upload Heap을 만들어서 CPU에서 Upload Heap으로 데이터를 입력하고 Upload Heap의 내용을 Default Heap으로 Copy하여 GPU가 읽는다.

CPU가 GPU에 데이터를 보내는 일은 많지만 GPU의 데이터를 CPU가 읽는 일은 거의 없다.

그렇기 때문에 CPU->GPU는 속도가 빠르지만 GPU->CPU는 매우 느리다.

리소스를 생성하면 가상 메모리에 생성이 되며, 가상 메모리 주소가 반환 된다.

Committed Resource는 가상메모리와 GPU메모리가 같이 생성된다.



OMSetRenderTargets

* 여러 개의 Render Target들과 Depth/Stencil을 Output Merger에 설정하는 것이다.
* Render Target이 아닌 Render Target View를 연결한다.