**DirectX를 이용한**

**3D Game 프로그래밍 입문**

**< 개요 >**

Direct3D는 3D가속 하드웨어를 이용해 3D 세계를 표현할 수 있게 돕는

저수준 그래픽 API를 말함.

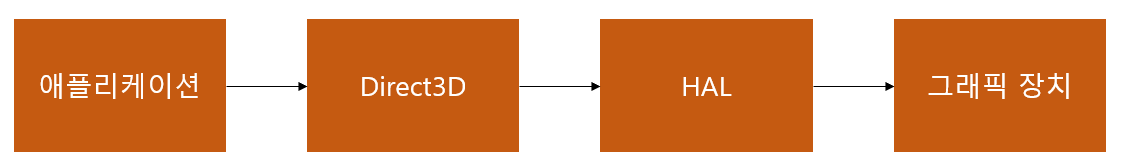


그림 1 애플리케이션, Direct3D, 하드웨어 간의 관계

**< HAL(Hardware Abstraction Layer) >**

HAL은 장치로 하여금 특별한 작업을 수행하도록 하는 장치 고유의 코드로, 이를 통해 Direct3D가 각 장치의 세부적인 부분을 제어할 필요가 없도록 하고, 하드웨어 장치에 독립적인 규약을 확립하는 것이 가능함. 장치 제조사에서는 자신들의 장치가 지원하는 모든 기능을 사용하여 HAL을 구현한다.

다만, Direct3D가 정의하지만 장치에서 지원하지 않는 기능들은 포함되지 않으며, HAL에서 구현되지 않은 기능을 사용하는 Direct3D함수를 호출하면 오류가 발생한다.

**< REF 장치 >**

개발자의 장치에서는 지원하지 않지만 Direct3D에서 제공하는 기능을 이용하는 프로그램을 작성하고자 할 때, 모든 Direct3D API를 소프트웨어로 애뮬레이트하는 레퍼런스 래스터라이저(REF)를 제공하고 있음. 이를 통해 지원하지 않는 Direct3D기능을 이용하는 코드를 작성하고 테스트할 수 있음.

REF 장치는 순수한 개발 목적으로 제공된다는 것에 주의.

REF장치는 SDK에만 포함되며, 최종 사용자에게는 배포할 수 없음.

또한 테스트 이외의 목적으로 이용하기에는 상당히 느림.

**< D3DDEVTYPE >**

HAL 장치는 D3DDEVTYPE열거형의 멤버인 D3DDEVTYPE\_HAL로 지정된다.

비슷하게 REF장치 역시 D3DDEVTYPE열거형의 멤버인 D3DDEVTYPE\_REF로 지정된다.

**< COM(컴포넌트 객체 모델, Component Object Model) >**

COM은 DirectX를 프로그래밍 언어로부터 독립적으로 만들어주며, 하위 호환성을 갖출 수 있게 하는 기술. 보통 인터페이스로 부르며, C++ 클래스와 비슷하게 이용됨.

다른 COM 인터페이스의 메서드나 특수한 함수를 통해 COM 인터페이스의 포인터를 얻는 것이며(빌려쓴다.), 직접 COM인터페이스를 만드는 것이 아님.

모든 COM인터페이스는 IUnknown COM 인터페이스에서 기능을 상속받음.

또한, 사용이 모두 끝나면 직접 삭제하지 않고 인터페이스에서 제공하는 Release 메서드를 호출해야 함.

COM 객체는 자신의 메모리 관리는 스스로 수행.

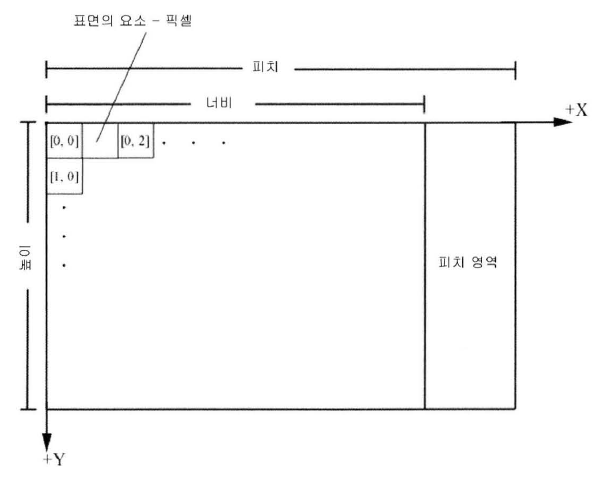
|  |
| --- |
| COM 인터페이스는 접두어로 대문자 I를 가짐. |

**< 기본적인 그래픽 개념과 Direct3D 형에 대한 이해 >**

**표면(Surface)**

표면은 Direct3D가 주로 2D 이미지 데이터를 보관하는 데 이용하는 픽셀의 행렬.

그림에서는 표면 데이터를 행렬로 그렸지만, 실제로 픽셀데이터는 선형 배열에 보관됨.

****

표면의 너비와 높이는 픽셀 단위로 계산됨.

피치는 바이트로 계산되며, 하드웨어 구현에 따라서는 너비보다 길 수도 있음.

따라서 피치=너비 \* sizeof(pixelFormat)라고 가정할 수는 없음.

코드에서 표면을 이용하는 데는 IDirect3DSurface9 인터페이스가 사용됨.

**IDirect3DSurface9**

중요한 메서드들

* LockRect : 표면 메모리로의 포인터를 제공. 여기서 약간의 포인터 연산을

거치면 표면내의 각 픽셀을 읽고 쓸 수 있게 됨.

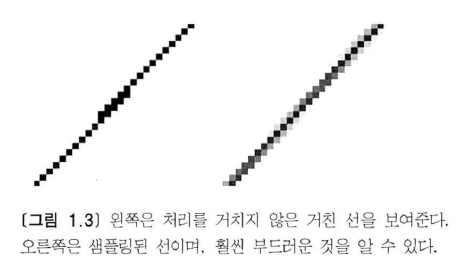
* UnlockRect: LockRect를 호출하고 표면 메모리에 대한 작업이 끝난 뒤에는

이 메서드를 호출하여 표면의 잠금을 해제해야 함.

* GetDesc: 표면에 대한 정보를 D3DSURFACE\_DESC구조체를 통해 얻는다.

**멀티 샘플링**

멀티 샘플링은 픽셀 매트릭스로 이미지를 표현할 때 나타나는 거친 이미지를 부드럽게 만드는 데 이용되는 기술. 가장 일반적으로 안티 앨리어싱을 위한 표면 멀티 샘플링이 있음.



멀티 샘플링은 응용프로그램의 속도를 지나치게 떨어뜨림.

멀티 샘플링을 이용하고자 한다면 IDirect3D9::CheckDeviceMultiSampleType 메서드를 통해 그래픽 카드에서 지원하는 멀티 샘플링 타입과 품질 레벨에 대해 확인해야 함.

**D3DMULTISAMPLE\_TYPE**

D3DMULTISAMPLE\_TYPE 열거형은 표면에 적용할 멀티 샘플링 레벨을 지정할 수 있도록 하는 값들로 구성되어 있음.

* D3DMULTISAMPLE\_NONE : 멀티 샘플링을 지정하지 않음.
* D3DMULTISAMPLE\_1\_SAMPLE, … , D3DMULTISAMPLE\_16\_SAMPLE :

1에서 16까지의 멀티 샘플링 레벨을 지정함.

**픽셀포맷**

표면이나 텍스처를 만들기 위해서는 Direct3D 자원의 픽셀 포맷을 지정해야 함.

픽셀 포맷은 D3DFORMAT 열거형 멤버로 지정되며, 자주 이용되는 포맷은 아래와 같음.

* **D3DFMT\_R8G8B8** : 24비트 픽셀 포팻을 지정. 가장 왼쪽의 8비트는 적색, 다음 8 비트는 녹색, 마지막 8비트는 청색에 할당.
* **D3DFMT\_X8R8G8B8** : 32비트 픽셀 포맷을 지정. 가장 왼쪽의 8비트는 이용되지 않으며, 다음 8비트는 적색, 다음 8 비트는 녹색, 마지막 8비트는 청색에 할당.
* **D3DFMT\_A8R8G8B8** : 32비트 픽셀 포맷을 지정하고, 가장 왼쪽 8비트는 투명도(알파)에 해당됨.
* **D3DFMT\_A16R16G16B16** : 64비트 부동소수점 픽셀 포맷.(투명도 포함)
* **D3DFMT\_A32R32G32B32** : 128비트 부동소수점 픽셀 포맷.(투명도 포함)

첫 세 개의 포맷은 대부분 하드웨어에서 지원하지만, 부동소수점 픽셀 포맷이나 나머지 포맷은 그리 널리 지원되지 않음.

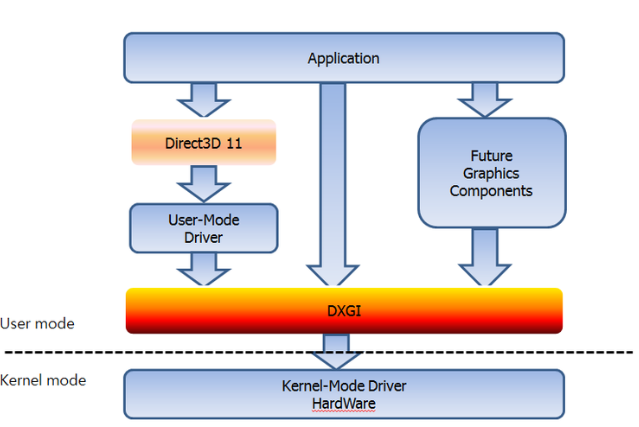
**메모리 풀**

표면이나 그 밖의 다양한 Direct3D자원들은 여러가지 종류의 메모리 풀에 보관할 수 있음. 메모리 풀은 D3DPOOL 열거형의 멤버로 지정되며, 이용할 수 있는 메모리 풀에는 아래와 같은 것들이 있음.

* **D3DPOOL\_DEFAULT** : 디폴트 메모리 풀은 자원의 타입과 이용방식에 가장 적합한 자원들을 메모리에 보관하도록 Direct3D에 요청함. 여기서 말하는 자원이란, 비디오 메모리나 AGP 메모리, 혹은 시스템 메모리 등을 말하는 것으로, 디폴트 풀 내의 자원은 반드시 IDirect3DDevice9::Reset호출 이전에 파괴(해제)되어야 하며, reset 호출 이후에 다시 초기화 되어야 한다.
* **D3DPOOL\_MANAGED** : 관리 풀에 보관된 자원은 Direct3D에 의해 관리된다.(즉 필요에 따라 자동으로 비디오 메모리나 AGP메모리로 옮겨진다.) 부가적으로 자원의 백업 복사본이 시스템 메모리 내에 보관되는데, 응용프로그램이 자원에 접근하고 수정할 때는 시스템 복사본을 이용하며, Direct3D는 필요에 따라 자동으로 이를 비디오 메모리에 갱신함.
* **D3DPOOL\_SYSTEMMEM** : 시스템 메모리 내에 보관될 자원을 지정함.
* **D3DPOOL\_SCRATCH** : 시스템 메모리 내에 보관될 자원을 지정함. 앞서의 D3DPOOL\_SYSTEMMEM과는 달리 이 풀의 자원은 그래픽 장치의 제한을 따라서는 안 된다. 따라서 장치는 이 풀 내의 자원에 직접 접근할 수 없지만, 자원을 두 풀 사이에 서로 복사하는 것은 가능함.

**DXGI**

DXGI는 Direct11등 여러 그래픽스 기능이나 어플리케이션으로부터 오는 표시를 받아 커널 모드 드라이버나 하드웨어와 주고받는 역할을 함.



**DXGI 유저모드, 커널모드**

커널 모드에서는 중요한 자우너을 관리하기 때문에 사용자가 자원에 접근하지 못하도록 모드를 나눔.

* 유저모드 : 유저가 접근할 수 있는 영역을 제한적으로 두고, 프로그램의 자원에 침범하지 못하게 함. 보통 코드를 작성하고 프로세스를 실행.
* 커널모드 : 모든 자원(드라이버, 메모리, CPU 등)에 접근하여 명령할 수 있음. DirectX에서도 자원을 할당하고 받아올 때 커널모드를 통해 자원을 받아옴. 보통 직접 조작하지 않으나, 다음과 같은 직접 조작할 필요가 있는 경우엔 직접 조작하기도 함.
* 그래픽스 카드 선택
* 윈도우 사이즈 갱신 시 대응(백버퍼 사이즈 갱신)
* 윈도우 사이즈 갱신 (디스플레이 모드 갱신)
* 필요없는 화면 렌더링 제어
* 화면 모드 전환(풀스크린모드 <-> 윈도우 모드)
* 톤 커브에 의한 계조 보정 등

**DXGI 인터페이스**

|  |  |
| --- | --- |
| IDXGIFactory | DXGI를 사용하기 위해 필요한 각종 인터페이스를 얻어오는 역할. DXGI의 기능을 사용하는 경우는 우선 CreateDXGIFactory() 함수를 사용해 인터페이스를 얻어옴. |
| IDXGIAdapter | 그래픽스 카드 관련 기능을 여기서 다룸.  이것은 반드시 독립된 카드가 아닌 칩셋에 내장된 그래픽스 기능인 경우도 있음. (CPU 내장 그래픽 정도..?) |
| IDXGIOutput | 그래픽스 카드에 연결된 디스플레이 관련 기능을 여기서 다룸.  이 인터페이스를 사용함으로써 대응하고 있는 디스플레이모드(해상도나 Refresh모드) 또는 감마 설정 등을 할 수 있음. |
| IDXGISwapChain | 화면을 표시하는 스왑체인의 기능을 다룸.  이 인터페이스는 D3D11CreateDeviceAndSwapChain 함수로 DirectX11에 ID3D11Device인터페이스를 얻을 때 같이 얻을 수 있음. |

**스왑 체인과 페이지 플리핑**

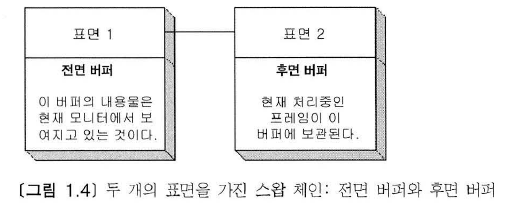
엔진에서 GPU가 프레임 버퍼에 이미지를 그리는 동안 비디오 컨트롤러가 GPU가 이미지를 그리고 있는 프레임 버퍼를 참조하여 화면에 출력할 때 덜 그려져 화면이 깜빡거리는 현상이 발생하게 됨. (**플리커 현상**)

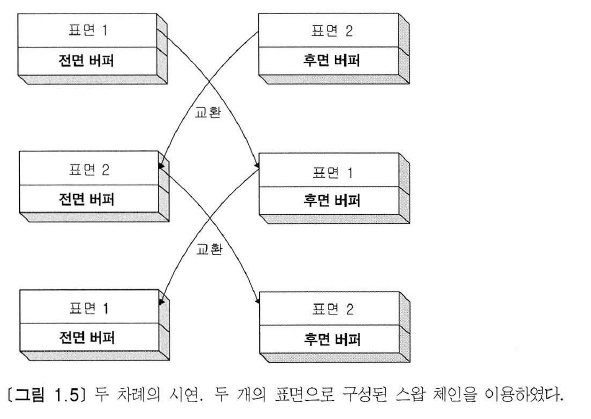
이를 해결하기 위해 GPU가 그릴 프레임 버퍼(후면 버퍼)와 화면에 출력할 프레임 버퍼(전면 버퍼)를 두 개 만들어 두고, 이 둘을 계속 교체하는 방식을 사용함.

Direct3D는 보통 두 개나 세 개의 표면(프레임 버퍼)을 하나의 컬렉션으로 관리하며, 이를 **스왑 체인 혹은 더블 버퍼링**이라고 부름.

스왑 체인은 IDirect3DSwapChain9 인터페이스를 통해 이용할 수 있지만, 대부분의 작업은 Direct3D가 직접 관리하므로 우리가 이 인터페이스를 이용하는 경우는 거의 없음.

스왑 체인과 페이지 플리핑 기술은 프레임 간의 부드러운 애니메이션을 제공하기 위한 것임.





화면 밖의 표면(후면 버퍼)에서 렌더링을 수행하고, 전면 버퍼 표면의 디스플레이가 완료되면 스왑 체인의 끝으로 돌아가 후면 버퍼를 전면 버퍼로 전환하는 방법을 이용. 이와 같은 과정을 시연(Presenting)이라고 함.

**각종 헤더파일**

* **ID3D11Device**.**h** : GPU를 제어할 수 있는 인터페이스. 기능 지원 점검과 자원할당에 사용함. D3D11CreateDeviceAndSwapChain으로 생성할 수 있음.

<https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nn-d3d11-id3d11device>

* **ID3D11DeviceContext** : 마찬가지로 GPU를 제어할 수 잇는 인터페이스. 렌더 대상을 설정하고, ID3D11Device로 할당된 자원을 그래픽 파이프 라인 묶고, GPU가 렌더링 명령을 지시하도록 함.

마찬가지로D3D11CreateDeviceAndSwapChain으로 생성할 수 있음.

<https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nn-d3d11-id3d11devicecontext>

* **ID3D11RenderTargetView** : IDXGISwapChain에서 SwapChain을 통해 렌더버퍼를 교체하는데, 이때 특정 버퍼를 렌더하도록 타겟팅하는 역할.

<https://vsts2010.tistory.com/517>

* **XMVECTOR, XMMATRIX** : DirectX에서는 XMVECTOR라는 벡터 형식을 사용. SIMD하드웨어 레지스터에 대응됨.

<https://stonzeteam.github.io/SIMD-%EB%B3%91%EB%A0%AC-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D/>

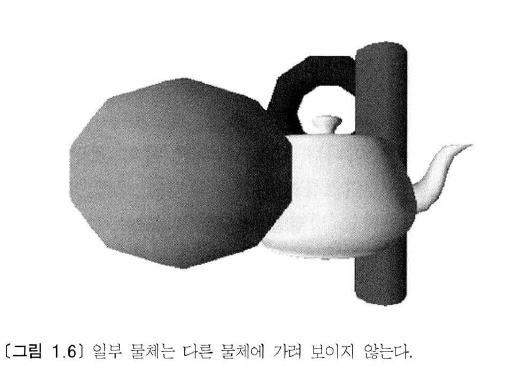
XMMATRIX는 XMVECTOR로 이루어져 있는 행렬임.

**깊이버퍼**

이미지 데이터가 아닌 특정 픽셀의 깊이 정보를 포함하는 표면을 말함.

깊이 버퍼 내에는 최종 렌더링된 이미지의 각 픽셀에 해당하는 항목들을 포함.

예를 들어 렌더링된 이미지가 640 \* 480 해상도를 가진다면, 640 \* 480 개의 깊이 항목이 존재함.



위 그림은 물체의 일부가 다른 물체에 가려 보이지 않는 간단한 상황을 보여주고 있음. Direct3D는 물체의 픽셀이 다른 픽셀을 가리는지의 여부를 판단하기 위해 깊이 버퍼링 혹은 z-버퍼링이라는 테크닉을 이용함.

깊이 버퍼링은 각 픽셀의 깊이 값을 계산하고 깊이 테스트를 수행함으로써 이루어지며, 깊이 테스트는 특정 픽셀의 위치에서 경쟁하는 픽셀의 깊이를 비교하는 과정임. 카메라와 가장 가까운 깊이 값을 가지는 픽셀이 경쟁에서 승리하며, 승리한 픽셀이 이미지에 그려진다. 카메라와 가까운 픽셀이 뒤쪽의 픽셀을 가리는 간단한 원리임.

깊이 버퍼의 포맷은 깊이 테스트의 정확도를 결정함. 즉, 24 비트 깊이 버퍼는 16비트 깊이 버퍼에 비해 더욱 정확함. Direct3D는 32비트 깊이 버퍼를 제공하고 있지만, 대부분의 응용프로그램에서는 24비트 깊이 버퍼로도 충분함.

* **D3DFMT\_D32** : 32비트 깊이 버퍼를 지정.
* **D3DFMT\_D24S8** : 24비트 깊이 버퍼를 지정하고 스텐실 버퍼로 8비트를 예약.
* **D3DFMT\_D24X8** : 24비트 깊이 버퍼를 지정
* **D3DFMT\_D24X4S4** : 24비트 깊이 버퍼를 지정하고 스텐실 버퍼로 4비트를 예약.
* **D3DFMT\_D16** : 16비트 깊이 버퍼를 지정.

**버텍스 프로세싱**

버텍스는 3D 기하 물체를 구성하는 기본 단위로, 소프트웨어나 하드웨어 의 두 가지 방법으로 처리할 수 있음. 소프트웨어 버텍스 프로세싱은 언제나 지원되며 항상 이용 가능한 반면, 하드웨어 버텍스 프로세싱은 그래픽 카드가 버텍스 프로세싱을 지원하는 경우에만 이용할 수 있음.

소프트웨어에 비해 전용 하드웨어의 성능이 우수하므로 하드웨어 버텍스 프로세싱을 이용하는 것이 유리하다. 게다가 하드웨어로 버텍스를 처리하면 CPU를 다른 계산에 할당할 수 있다.

**텍스트, 테이블이(가) 표시된 사진

자동 생성된 설명**

**장치특성**

Direct3D가 제공하는 모든 기능들은 D3DCAPS9 구조체 내의 비트와 데이터 멤버에 대응됨.

기본적인 방식은 특정 하드웨어의 특성에 따라 D3DCAPS9 인스턴스의 멤버를 초기화하는 것으로, 우리의 응용프로그램에서는 D3DCAPS9 인스턴스의 대응되는 비트나 데이터 멤버를 확인하여 장치가 특정한 기능을 제공하는 지를 확인할 수 있다. (SDK 문서 참고)

텍스트이(가) 표시된 사진

자동 생성된 설명

**<Direct3D 초기화>**

1. 수직 동기화 : 수직 동기화는 그래픽 카드의 프레임 생성과 모니터의 프레임 출력 타이밍을 딱 맞게 설정. 타이밍이 맞지 않으면 테어링이라는 화면 깨짐 현상이 발생하며, 보통 프로그램의 유저가 설정함.  
   <https://namu.wiki/w/%EC%88%98%EC%A7%81%EB%8F%99%EA%B8%B0%ED%99%94>
2. DirectX 그래픽 인터페이스 팩토리 생성:

텍스트이(가) 표시된 사진

자동 생성된 설명

\* \_\_uuidof(class) : GUID반환 함수. COM에서 GUID인터페이스들을 구별하기 위해 사용함. 서로 호환되지 않을 수 있는 두개의 컴포넌트가 동일한 인터페이스 이름을 사용하더라도 각각의 인터페이스는 언제나 고유한 GUID를 갖기 때문에 구별할 수 있음.  
https://ko.wikipedia.org/wiki/%EC%A0%84%EC%97%AD\_%EA%B3%A0%EC%9C%A0\_%EC%8B%9D%EB%B3%84%EC%9E%90

\* (void\*\*) : 반환될 팩토리가 IDXGIFactory가 아닌 다른 형태일 경우가 있어 void 이중 포인터로 받음.

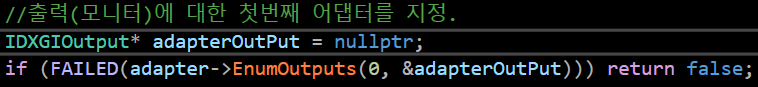
(이중포인터)  
<https://m.blog.naver.com/PostView.nhn?blogId=skout123&logNo=50131421237&proxyReferer=https%3A%2F%2Fwww.google.com%2F>

1. 팩토리 객체를 사용하여 첫번째 그래픽 카드 인터페이스 어댑터 생성.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 출력(모니터)에 대한 첫번째 어댑터 지정.



1. 출력(모니터)에 대한 DXGI\_FORMAT\_R8G8B8A8\_UNORM 표시 형식에 맞는 모드 수를 가져옴.

텍스트이(가) 표시된 사진

자동 생성된 설명

\* GetDisplayModeList(enum, UINT1, UINT2, UINT3, DXGI\_MODE\_DESC):

enum : 열거할 색상 포맷.

UINT(1) : 디스플레이 열거 옵션.

UINT(2) : 디스플레이 포맷과 옵션을 기준으로 디스플레이 모드의

개수를 반환. DXGI\_ENUM\_MODES\_INTERLACED는 [인터레이스] 모드를 포함하라는 뜻.

<https://macinjune.com/all-posts/mac/tip/final-cut/1080i-vs-1080p-%EC%9D%B8%ED%84%B0%EB%A0%88%EC%9D%B4%EC%8A%A4%EB%93%9C%EC%99%80-%ED%94%84%EB%A1%9C%EA%B7%B8%EB%A0%88%EC%8B%9C%EB%B8%8C-%EB%B0%A9%EC%8B%9D%EC%9D%B4%EB%9E%80/>

UINT(3) : 디스플레이 모드의 리스트를 반환.

1. 가능한 모든 모니터와 그래픽카드 조합을 저장할 리스트를 생성.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

1. 디스플레이 모드에 대한 리스트 채우기.

텍스트이(가) 표시된 사진

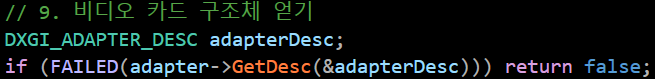
자동 생성된 설명

1. 모든 디스플레이 모드에 대해 화면 너비/높이에 맞는 디스플레이 모드를 찾음. (만약 적합한 것을 찾으면 모니터의 새로고침 비율의 분모와 분자값을 저장.)

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 비디오 카드 구조체 받아오기.



<https://docs.microsoft.com/en-us/windows/win32/api/dxgi/ns-dxgi-dxgi_adapter_desc>

1. Byte로 저장되었던 비디오 카드 메모리를 2^10씩 나누어 KB, MB로 변환.



1. 어댑터 및 팩토리 관련 변수 해제.

텍스트, 점수판이(가) 표시된 사진

자동 생성된 설명

1. 스왑체인 구조체 초기화

텍스트, 오렌지, 설정이(가) 표시된 사진

자동 생성된 설명

백버퍼 개수 및 높이, 너비 지정

텍스트이(가) 표시된 사진

자동 생성된 설명

32bit 서페이스 지정



백버퍼 새로고침 비율 설정(수직 동기화 설정)

텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

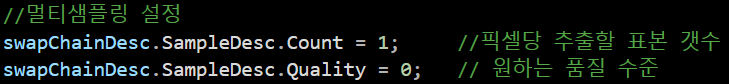
자동 생성된 설명

백버퍼 사용용도 및 윈도우 핸들 지정.

텍스트이(가) 표시된 사진

자동 생성된 설명

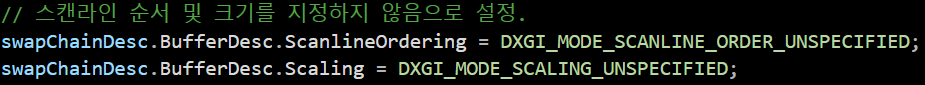
멀티 샘플링 지정



창모드 설정



스캔라인 순서 및 크기를 지정하지 않음으로 설정



출력 후 백버퍼를 비우도록 지정.



추가옵션 플래그는 사용하지 않음

텍스트이(가) 표시된 사진

자동 생성된 설명

피처레벨 DirectX 11로 설정



1. 스왑체인, Direct3D장치 및 Direct3D 장치 컨텍스트 생성

텍스트이(가) 표시된 사진

자동 생성된 설명

1. RenderTargetView 초기화

텍스트, 실내, 스크린샷, 화면이(가) 표시된 사진

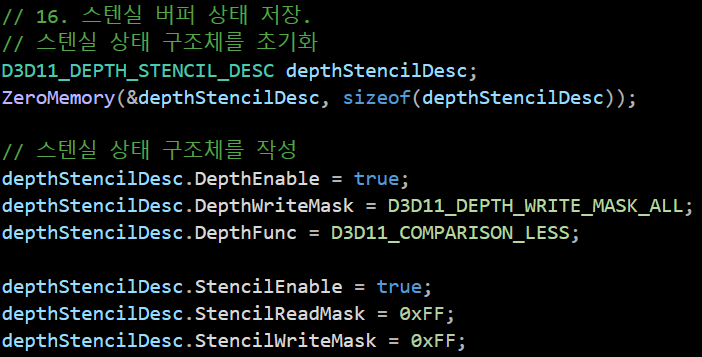
자동 생성된 설명

1. 깊이 버퍼 텍스쳐 생성

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 스텐실 버퍼 상태 저장



픽셀 정면 및 뒷면 스텐실 설정

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 깊이 스텐실 상태를 생성 및 저장.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 깊이/스텐실 뷰 바인딩

텍스트, 스크린샷, 모니터, 화면이(가) 표시된 사진

자동 생성된 설명



1. 레스터 라이저 설정

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 뷰포트 설정

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 투영 행렬 설정

텍스트, 모니터, 화면, 스크린샷이(가) 표시된 사진

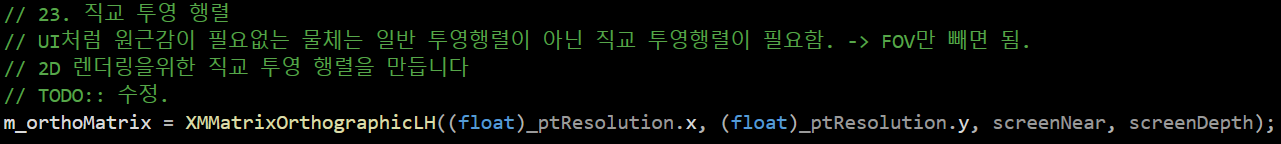
자동 생성된 설명

1. 월드 행렬 설정

텍스트이(가) 표시된 사진

자동 생성된 설명

1. 직교 행렬 설정



Direct3D 초기화 전체 코드

|  |
| --- |
| bool D3DClass::Initialize(HWND \_hWnd, POINT \_ptResolution, bool \_bVSync, bool \_bFullscreen, float screenDepth, float screenNear)  {    m\_bVsync = \_bVSync; // 1. 수직 동기화 상태 저장.  // 2. DirectX 그래픽 인터페이스 팩토리를 생성.  IDXGIFactory\* factory = nullptr;  if (FAILED(CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory))) return false;  // 3. 팩토리 객체를 사용해 첫번째 그래픽 카드 인터페이스 어댑터를 생성.  IDXGIAdapter\* adapter = nullptr;  if (FAILED(factory->EnumAdapters(0, &adapter))) return false;  // 4. 출력(모니터)에 대한 첫번째 어댑터를 지정.  IDXGIOutput\* adapterOutput = nullptr;  if (FAILED(adapter->EnumOutputs(0, &adapterOutput))) return false;  // 5. 출력 (모니터)에 대한 DXGI\_FORMAT\_R8G8B8A8\_UNORM 표시 형식에 맞는 모드 수를 얻음.  unsigned int numModes = 0;  if (FAILED(adapterOutput->GetDisplayModeList(  DXGI\_FORMAT\_R8G8B8A8\_UNORM,  DXGI\_ENUM\_MODES\_INTERLACED,  &numModes,  NULL)))  {  return false;  }  // 6. 가능한 모든 모니터와 그래픽카드 조합을 저장할 리스트를 생성.  DXGI\_MODE\_DESC\* displayModeList = new DXGI\_MODE\_DESC[numModes];  if (!displayModeList) return false;  // 7. 디스플레이 모드에 대한 리스트를 채웁니다  if (FAILED(adapterOutput->GetDisplayModeList(  DXGI\_FORMAT\_R8G8B8A8\_UNORM,  DXGI\_ENUM\_MODES\_INTERLACED,  &numModes, displayModeList)))  {  return false;  }  //8. 모든 디스플레이 모드에 대해 화면의 너비/높이에 맞는 디스플레이 모드를 찾음.  // (만약 적합한 것을 찾으면, 모니터의 새로고침 비율의 분모와 분자값을 저장함.)  UINT numerator = 0;  UINT denominator = 0;  for (UINT i = 0; i < numModes; i++)  {  if (displayModeList[i].Width == \_ptResolution.x &&  displayModeList[i].Height == \_ptResolution.y)  {  numerator = displayModeList[i].RefreshRate.Numerator;  denominator = displayModeList[i].RefreshRate.Denominator;  }  }    // 9. 비디오 카드 구조체 얻기  DXGI\_ADAPTER\_DESC adapterDesc;  if (FAILED(adapter->GetDesc(&adapterDesc))) return false;    // 10. 비디오카드 메모리 용량 단위를 메가바이트 단위로 저장.  m\_videoCardMemory = (int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);  // 비디오카드 이름 저장.  // TODO:: 대체할 함수 혹은 오류 수정.  /\*size\_t stringLength = 0;  if (wcstombs\_s(&stringLength, m\_videoCardDescription, 128, strName, 128) != 0) return false;\*/  m\_videoCardDescription = adapterDesc.Description;  // 11. 어댑터 및 팩토리 관련 변수 해제  // 디스플레이 모드 리스트를 해제  delete[] displayModeList;  displayModeList = 0;  // 출력 어뎁터를 해제  adapterOutput->Release();  adapterOutput = 0;  // 어뎁터를 해제  adapter->Release();  adapter = 0;  // 팩토리 객체를 해제  factory->Release();  factory = 0;  // 12. 스왑체인 구조체 초기화  DXGI\_SWAP\_CHAIN\_DESC swapChainDesc;  ZeroMemory(&swapChainDesc, sizeof(DXGI\_SWAP\_CHAIN\_DESC));  // 백버퍼 갯수 지정.  swapChainDesc.BufferCount = 1;  //백버퍼 넓이, 높이 지정  swapChainDesc.BufferDesc.Width = \_ptResolution.x;  swapChainDesc.BufferDesc.Height = \_ptResolution.y;  //32bit 서페이스 지정.  swapChainDesc.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;  //백버퍼 새로고침 비율 설정(수직동기화 처리)  if (m\_bVsync)  {  swapChainDesc.BufferDesc.RefreshRate.Numerator = numerator;  swapChainDesc.BufferDesc.RefreshRate.Denominator = denominator;  }  else  {  swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;  swapChainDesc.BufferDesc.RefreshRate.Denominator = 1;  }  //백버퍼 사용용도 지정.  swapChainDesc.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;  //렌더링에 사용될 윈도우 핸들 지정.  swapChainDesc.OutputWindow = \_hWnd;  //멀티샘플링 설정  swapChainDesc.SampleDesc.Count = 1; //픽셀당 추출할 표본 갯수  swapChainDesc.SampleDesc.Quality = 0; // 원하는 품질 수준  //창모드 or 풀스크린 설정  swapChainDesc.Windowed = !\_bFullscreen;  // 스캔라인 순서 및 크기를 지정하지 않음으로 설정.  swapChainDesc.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED;  swapChainDesc.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED;  // 출력된 다음 백버퍼를 비우도록 지정.  swapChainDesc.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD;  // 추가 옵션 플래그를 사용하지 않음.  swapChainDesc.Flags = 0;  //피처레벨 DirectX 11로 설정.  D3D\_FEATURE\_LEVEL featureLevel = D3D\_FEATURE\_LEVEL\_11\_0;  // 13. 스왑체인, Direct3D 장치 및 Direct3D 장치 컨텍스트 생성  if (FAILED(D3D11CreateDeviceAndSwapChain(NULL, D3D\_DRIVER\_TYPE\_HARDWARE, NULL, 0, &featureLevel, 1,  D3D11\_SDK\_VERSION, &swapChainDesc, &m\_pSwapChain, &m\_pDevice, NULL, &m\_pDeviceContext)))  {  return false;  }  // 14. RenderTargetView 초기화  // 백버퍼 포인터를 얻어오기  ID3D11Texture2D\* backBufferPtr = nullptr;  if (FAILED(m\_pSwapChain->GetBuffer(0, \_\_uuidof(ID3D11Texture2D), (LPVOID\*)&backBufferPtr))) return false;  // 백 버퍼 포인터로 렌더 타겟 뷰 생성  if (FAILED(m\_pDevice->CreateRenderTargetView(backBufferPtr, NULL, &m\_pRenderTargetView))) return false;  // 백버퍼 포인터 해제  backBufferPtr->Release();  backBufferPtr = 0;  // 15. 깊이 버퍼 텍스쳐 생성  // 깊이 버퍼 구조체를 초기화  D3D11\_TEXTURE2D\_DESC depthBufferDesc;  ZeroMemory(&depthBufferDesc, sizeof(depthBufferDesc));  // 깊이 버퍼 구조체를 작성  depthBufferDesc.Width = \_ptResolution.x;  depthBufferDesc.Height = \_ptResolution.y;  depthBufferDesc.MipLevels = 1;  depthBufferDesc.ArraySize = 1;  depthBufferDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;  depthBufferDesc.SampleDesc.Count = 1;  depthBufferDesc.SampleDesc.Quality = 0;  depthBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;  depthBufferDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;  depthBufferDesc.CPUAccessFlags = 0;  depthBufferDesc.MiscFlags = 0;  // 설정된 깊이버퍼 구조체를 사용하여 깊이 버퍼 텍스쳐를 생성  if (FAILED(m\_pDevice->CreateTexture2D(&depthBufferDesc, NULL, &m\_pDepthStencilBuffer))) return false;  // 16. 스텐실 버퍼 상태 저장.  // 스텐실 상태 구조체를 초기화  D3D11\_DEPTH\_STENCIL\_DESC depthStencilDesc;  ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));  // 스텐실 상태 구조체를 작성  depthStencilDesc.DepthEnable = true;  depthStencilDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;  depthStencilDesc.DepthFunc = D3D11\_COMPARISON\_LESS;  depthStencilDesc.StencilEnable = true;  depthStencilDesc.StencilReadMask = 0xFF;  depthStencilDesc.StencilWriteMask = 0xFF;  // 픽셀 정면의 스텐실 설정  depthStencilDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;  depthStencilDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;  depthStencilDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;  depthStencilDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;  // 픽셀 뒷면의 스텐실 설정  depthStencilDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;  depthStencilDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_DECR;  depthStencilDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;  depthStencilDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;  // 깊이 스텐실 상태를 생성  if (FAILED(m\_pDevice->CreateDepthStencilState(&depthStencilDesc, &m\_pDepthStencilState))) return false;  // 17. 깊이 스텐실 상태 저장.  m\_pDeviceContext->OMSetDepthStencilState(m\_pDepthStencilState, 1);  // 18. 깊이/스텐실 뷰 바인딩  // 깊이 스텐실 뷰의 구조체를 초기화  D3D11\_DEPTH\_STENCIL\_VIEW\_DESC depthStencilViewDesc;  ZeroMemory(&depthStencilViewDesc, sizeof(depthStencilViewDesc));  // 깊이 스텐실 뷰 구조체를 설정  depthStencilViewDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;  depthStencilViewDesc.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;  depthStencilViewDesc.Texture2D.MipSlice = 0;  // 깊이 스텐실 뷰를 생성합니다  if (FAILED(m\_pDevice->CreateDepthStencilView(m\_pDepthStencilBuffer, &depthStencilViewDesc, &m\_pDepthStencilView)))  {  return false;  }  // 렌더링 대상 뷰와 깊이 스텐실 버퍼를 출력 렌더 파이프 라인에 바인딩  m\_pDeviceContext->OMSetRenderTargets(1, &m\_pRenderTargetView, m\_pDepthStencilView);  // 19. 레스터 라이저 설정  // 그려지는 폴리곤과 방법을 결정할 래스터 구조체를 설정  D3D11\_RASTERIZER\_DESC rasterDesc;  rasterDesc.AntialiasedLineEnable = false;  rasterDesc.CullMode = D3D11\_CULL\_BACK;  rasterDesc.DepthBias = 0;  rasterDesc.DepthBiasClamp = 0.0f;  rasterDesc.DepthClipEnable = true;  rasterDesc.FillMode = D3D11\_FILL\_SOLID;  rasterDesc.FrontCounterClockwise = false;  rasterDesc.MultisampleEnable = false;  rasterDesc.ScissorEnable = false;  rasterDesc.SlopeScaledDepthBias = 0.0f;  // 방금 작성한 구조체에서 래스터 라이저 상태만들기  if (FAILED(m\_pDevice->CreateRasterizerState(&rasterDesc, &m\_pRasterizerState)))  {  return false;  }  // 래스터 라이저 상태를 설정  m\_pDeviceContext->RSSetState(m\_pRasterizerState);  //20. 뷰포트 설정  // 렌더링을 위해 뷰포트를 설정  D3D11\_VIEWPORT viewport;  viewport.Width = (float)\_ptResolution.x;  viewport.Height = (float)\_ptResolution.y;  viewport.MinDepth = 0.0f;  viewport.MaxDepth = 1.0f;  viewport.TopLeftX = 0.0f;  viewport.TopLeftY = 0.0f;  // 뷰포트 생성  m\_pDeviceContext->RSSetViewports(1, &viewport);  // 21. 투영 행렬 설정 (3차원의 물체를 2차원으로 표현하기 위해 쓰는 행렬.)  //-> 투영과정을 거쳐야 2D모니터에 출력이 가능하기 때문에 필수 과정임.  // 투영 행렬 설정  // FOV(Field Of View) -> 화면에 표현할 시야의 각도를 표현하는 값. 보여주는 각도 만큼 투영시킬 화면의 범위를 넓혀줌.  float fieldOfView = XM\_PI / 4.0f;  float screenAspect = (float)\_ptResolution.x / (float)\_ptResolution.y;  // 3D 렌더링을위한 투영 행렬 만들기  // TODO:: 수정.  m\_projectionMatrix = XMMatrixPerspectiveFovLH(fieldOfView, screenAspect, screenNear, screenDepth);  // 22. 월드 행렬 설정  // 월드 행렬은 모델링을 월드 좌표로 옮겨줄때 사용하는 행렬임.  // TODO:: 공부  // 월드 행렬을 항등 행렬로 초기화합니다  m\_worldMatrix = XMMatrixIdentity();  // 23. 직교 투영 행렬  // UI처럼 원근감이 필요없는 물체는 일반 투영행렬이 아닌 직교 투영행렬이 필요함. -> FOV만 빼면 됨.  // 2D 렌더링을위한 직교 투영 행렬을 만듭니다  // TODO:: 수정.  m\_orthoMatrix = XMMatrixOrthographicLH((float)\_ptResolution.x, (float)\_ptResolution.y, screenNear, screenDepth);  return true;  } |

**< 버텍스 버퍼(정점 버퍼) >**

구형 모델은 실제로 수백 개의 삼각형으로 구성됨.

각 삼각형에는 3개의 점이 있으며, 각 점을 꼭짓점이라고 부름

따라서 우리가 구형모델을 렌더링하려면 구형을 구성하는 모든 정점을 버텍스 버퍼라고 부르는 특수 데이터 배열에 넣어야 함.

구형 모델의 모든 점이 버텍스 버퍼에 존재하면, 우리는 버텍스 버퍼를 GPU에 전송하여 모델을 렌더링 할 수 있음.

**< 인덱스 버퍼 >**

인덱스 버퍼는 정점 버퍼와 관련이 있음. 그 목적은 버텍스 버퍼에 있는 각 정점의 위치를 기록하는 것.

이후 GPU는 인덱스 버퍼를 사용하여 버텍스 버퍼의 특정 정점을 빠르게 찾음.

색인 버퍼의 개념은 책에서 색인을 사용하는 개념과 유사하며, 찾고 있는 주제를 훨씬 더 빠르게 찾을 수 있음. DirectX문서에 보면 인덱스 버퍼를 사용하면 비디오 메모리에 더 빠른 위치에 버텍스 데이터를 캐싱 할 가능성을 높일 수 있다고 설명함. 따라서 성능상의 이유로도 인덱스 버퍼 사용을 권장.

**< 버텍스 셰이더 >**

버텍스 셰이더는 주로 정점 버퍼의 정점들을 3D공간으로 변환시켜주는 작ㅇ은 프로그램. 이 외에도 각 정점의 법선을 계산한다던가 하는 다른 연산도 가능.

정점 셰이더 프로그램은 GPU에서 계산이 필요하다고 판단될 때 호출 됨.

예를 들어, 5,000개의 폴리곤을 가진 모델을 화면에 표시한다면, 단지 저 하나의 모델을 그리기 위해 매 프레임 마다 15000번의 정점 셰이더 프로그램이 실행됨.

따라서 프로그램이 60fps를 가진 그래픽 프로그램에서는 단지 5000개의 삼각형을 그리기 위해 매 초마다 최대 900,000번의 정점 셰이더를 호출하게 됨. 따라서 효율적인 버텍스 셰이더를 작성하는 것이 중요함.

**< 픽셀 셰이더 >**

픽셀 셰이더는 그리고자 하는 도형에 색상을 입힐 때 필요한 작은 프로그램임.

픽셀 셰이더는 화면에 보여지는 모든 픽셀들에 대해 GPU에서 연산됨.

색상을 입히고(Coloring), 텍스쳐를 입히고(Texturing) 광원 효과를 주고(Lighting) 그 외 다른 많은 도형 채색효과를 주는 것을 픽셀 셰이더 프로그램에서 제어.

픽셀 셰이더는 GPU에 의해 수없이 호출되기 때문에 반드시 효율적으로 작성되어야 함.

**< HLSL(High Level Shader Language) >**

HLSL은 DirectX 11에서 사용하는 작은 정점 및 픽셀 셰이더 프로그램을 작성할 때 사용하는 일종의 언어임. 미리 정의된 타입이 있다는 것을 빼면 C언어와 거의 동일. HLSL 프로그램 파일은 전역변수, 타입정의, 버텍스 셰이더, 픽셀 셰이더, 그리고 기하 셰이더(Geometry Shader)로 구성되어 있음.