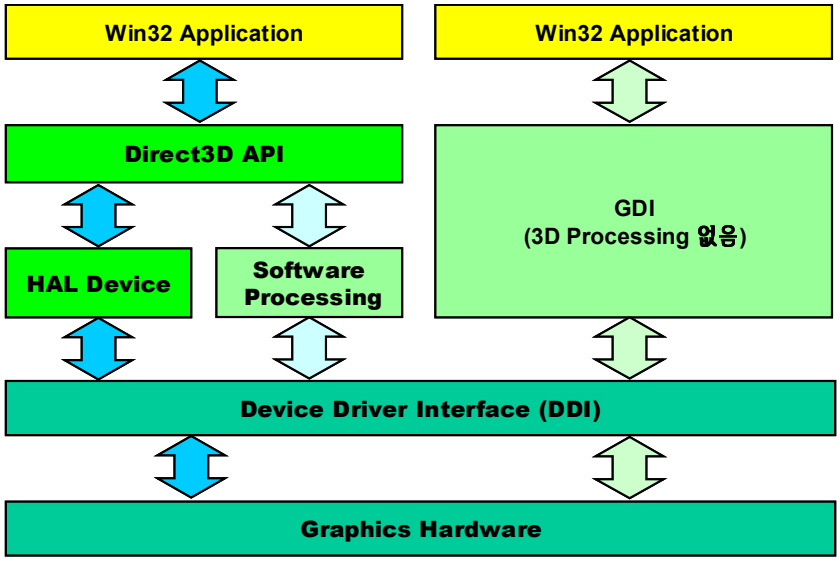
3D Game Programming Basic with Direct3D

1. Direct3D Overview

Direct3D(D3D)를 이용해서 2D 게임을 만들고자 할 때에는 D3D에 대한 많은 지식이 필요 없습니다. 단순히 D3D을 이용하기 위한 환경 설정과 ID3DXSprite 객체를 생성해서 이 객체의 멤버 함수 Draw() 함수를 이용해서 텍스처를 화면에 출력하면 됐습니다.
하지만 3D게임을 만들기 위해서, 중급 이상의 3D 게임 개발자가 되기 위해서는 Diect3D 구조에 대한 지식이 필요합니다. 이 강좌에서는 D3D를 기반으로 3D 게임에 필요한 그래픽 표현에 대한 기본적인 방법을 살펴 보겠습니다.

1.1 Direct3D 개요

지금까지 우리는 화면에 이미지를 표현하기 위해서 GDI(Graphic Device Interface)를 이용하는 방법과 D3D를 이용한 방법 두 가지를 배웠습니다. 이 두 가지 방법을 그림을 통해 비교하면 다음과 같습니다.



<Direct3D와 GDI 비교>

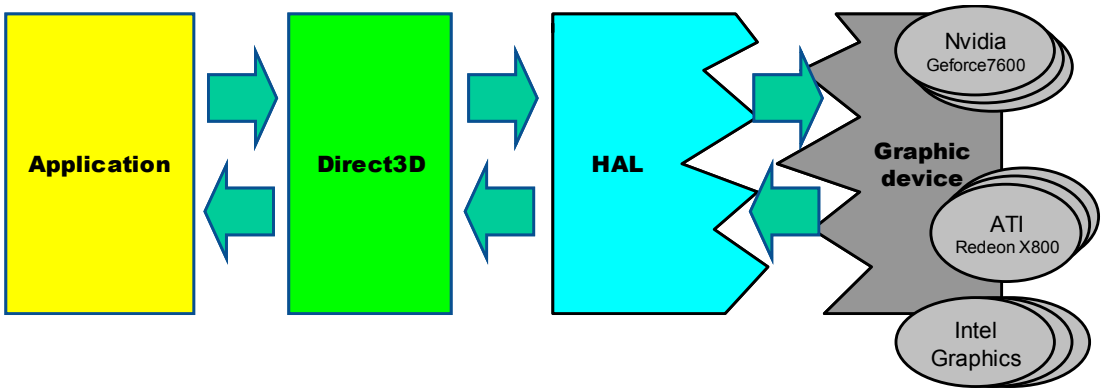
GDI는 그래픽 하드웨어에서 장면을 연출하는 렌더링을 지원이 되더라도 그와는 상관없이 운영체제가 CPU에서 화면 출력에 관한 모든 것을 제어하는 방식입니다. GDI를 이용하면 프로그래머는 유저의 하드웨어 종류에 독립적인 가장 일반적인 프로그램을 작성 할 수 있습니다.
3D 장면 연출은 많은 수학적인 계산이 필요합니다. 이러한 계산들은 주로 CPU에서 처리하는데 CPU

에서 처리하지 않고 그래픽 카드에 있는 그래픽 프로세서(GPU: Graphic Processing Unit)가 이를 담당할 때 "(그래픽 카드의) 하드웨어 가속을 받는다"라고 합니다. 그래픽 카드의 하드웨어 가속을 받게 되면 CPU는 그 만큼 자신이 하는 일이 줄어들게 되고, 컴퓨터는 좀 더 안정적인 상태가 됩니다.

D3D는 만약 하드웨어 가속이 지원되는 그래픽 카드를 만나면 HAL(Hardware Abstraction Layer: 하드웨어 추상 계층) 인터페이스를 통해서 하드웨어와 통신을 합니다. HAL이 지원이 안 되는, 즉, 그래픽 카드의 가속이 지원이 안 되는 하드웨어를 만나더라도 REF(Reference Rasterizing)를 이용해 소프트웨어적으로 CPU에서 3D 관련 수학처리를 할 수 있어서 프로그램을 크게 수정할 필요 없이 일관된 코드를 만들 수 있습니다.

만약 GDI를 이용한다면 여러분은 하드웨어 가속을 받을 수 없을 뿐만 아니라 3D 장면을 연출하는 모든 함수들을 스스로 작성해야 합니다. 반면에 D3D를 사용하면 그래픽 처리 과정을 D3D가 담당을 해서 좀 더 게임 프로그램에 집중할 수 있게 됩니다.

D3D와 HAL의 역할은 다음 그림처럼 제조사마다 다양한 그래픽 카드와 일관된 제어와 데이터 전송에 대한통신의 인터페이스라 할 수 있겠습니다.



<Application, Direct3D, 그래픽 장치와의 관계>

이 D3D와 HAL이 없다면 여러분은 게임 유저의 그래픽 카드 종류 마다 각각 코드를 만들어야 하고 이것은 아주 끔찍한 일이 아닐 수 없습니다.

게임 개발자는 Direc3D안에 있는 그래픽처리 객체 또는 함수를 이용해서 장면을 구하는 프로그램을 작성하고, HAL를 이용해서 하드웨어 가속을 받을 수 있습니다. 만약 유저의 그래픽 카드가 하드웨어 가속이 안되더라도 코드의 변경 없이 REF를 이용해서 CPU에서 장면을 에뮬레이트(Emulate)할 수 있습니다.

보통 REF를 사용하면 수 프레임 정도로 렌더링이 되는데 이것은 게임에서 사용할 수 없을 정도로 느린 속도이지만 GPU를 직접 디버깅 하기 어려울 때 그래픽 파이프라인의 디버깅을 위해서 사용되기도 합니다.

프로그램에서 HAL을 사용하기 위해서 다음과 같이 작성합니다.

```
// 윈도우 생성
m_hWnd = CreateWindow( …);

// D3D생성
m_pD3D = Direct3DCreate9( D3D_SDK_VERSION );

// Present Parmeter 구조체 설정
D3DPRESENT_PARAMETERS d3dpp={0};
…

// HAL을 지원받는 디바이스 생성
if( FAILED( m_pD3D->CreateDevice( D3DADAPTER_DEFAULT
                        , D3DDEVTYPE_HAL
                        , m_hWnd
                        , D3DCREATE_MIXED_VERTEXPROCESSING
                        , &d3dpp
                        , &m_pd3dDevice ) ) )
{
        if( FAILED( m_pD3D->CreateDevice( D3DADAPTER_DEFAULT
                                , D3DDEVTYPE_HAL
                                , m_hWnd
                                , D3DCREATE_SOFTWARE_VERTEXPROCESSING
                                , &d3dpp
                                , &m_pd3dDevice ) ) )
        {
                m_pD3D->Release();
                return -1;
        }
}
```

이 코드는 전에 2D 게임프로그램에서 작성해 본 것이니 어려움은 없을 것입니다. 위의 코드에서 D3D의 멤버 함수인 CreateDevice()에서 두 번째 인수를 D3DDEVTYPE_HAL로 설정해야만 HAL 지원을 받으며 만약 이것이 실패하면 REF로 바꾸어야 하나 REF는 렌더링 속도가 느려서 게임프로그램에 적합하지 않아 REF로 전환하지 않고 디바이스를 생성하지 않고 빠져 나가는 것이 중요합니다. 물론 “메시지는 HAL이 지원되지 않는 그래픽 카드입니다” 라는 메시지를 출력해주고 빠져나가면 더 좋습니다.
보충으로 CreateDevice() 함수에서 중요한 부분은 네 번째 인수인 Behavior Flags입니다. 위의 코

드에서 MIXED_VERTEXPROCESSING으로 실패하면 SOFTWAFE_VERTEXPROCESSING으로 디바이스를 생성하는데 이것은 만약 여러분이 D3DCREATE_PUREDEVICE라는 플래그를 사용하면 디바이스에 값을 설정할 수 있지만 가져올 수 없습니다. 하지만 MIXED나 SOFTWARE를 사용하면 디바이스에 대한 Set 뿐만 아니라 Get도 가능하고 또한 데스크탑과 노트북에서 잘 동작하지만 PURE를 사용하면 일부 노트북과 같이 사양이 조금 낮은 컴퓨터에서는 디바이스를 만들 수 없는 경우도 생깁니다. 물론 PURE로 처리하는 것이 가장 이상적이지만 하드웨어의 폭이 좁기 때문에 마지막으로 SOFTWARE를 선택합니다.
때로는 GPU의 실수 처리 능력을 정확히 하기 위해서 배정도(Double-Precision)를 요구할 수 있습니다. 이때는 D3DCREATE_FPU_PRESERVE 플래그와 같이 or 비트 연산자('|')를 이용해서 값을 전달합니다.

Ex) D3DCREATE_FPU_PRESERVE | D3DCREATE_SOFTWARE_VERTEXPROCESSING

멀티 쓰레드에서 안전하게 동작하기 위해서는 D3DCREATE_MULTITHREADED 플래그도 마찬가지로 or 비트 연산자로 연결하는데 D3DCREATE_FPU_PRESERVE, D3DCREATE_MULTITHREADED를 사용하게 되면 프레임이 급격히 떨어질 수 있음을 주의해야 합니다.

2D 과정에서 DirectX에 관련된 모든 객체들은 COM 객체를 상속받기 때문에 메모리는 Release()함수를 통해서 해제 한다고 했습니다. DirectX 객체들은 일정한 이름 패턴이 있는데 생성은 거의 다음과 같이 구성되어 있습니다.

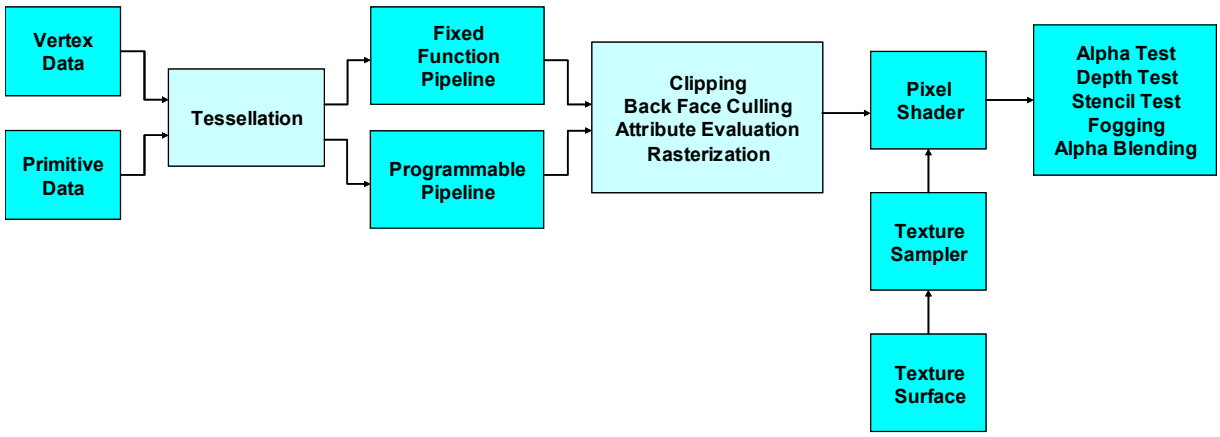
(객체를 생성하는 주체)->Create+ "대상 객체"(…, &"대상객체 주소")

보통은 객체를 통해서 객체를 생성하는 것이 일반적인 패턴이지만 때로는 함수를 통해서도 객체를 생성합니다. 이런 경우 통상적으로 "D3DX" 라는 접두어가 붙는 경우가 많이 있는데 D3DX로 시작하는 함수는 DirectX의 Utility Library들 입니다. 필자의 개인적으론 X가 아마 Extended가 아닌가 하는 생각인데 이 부분에 대해서 아는 분 있으면 연락 부탁 드리겠습니다.

D3DX+"대상 객체"(…, 객체를 생성하는 주체, &"대상객체 주소")

1.2 Direct3D Architecture 개요

간단히 컴퓨터 시스템에서 D3D와 HAL 살펴 보았습니다. 이제 본격적으로 D3D가 여러분이 입력한 data를 어떻게 화면에 만들어 내는지 설명해 드리겠습니다. 먼저 다음 그림을 눈 여겨 보기 바랍니다.



<Direct3D 그래픽 파이프라인 구조>

위의 그림은 DirectX SDK Help에서 얻은 D3D의 내부 구조입니다. 이 구조는 일반적인 그래픽 파이프라인의 구조를 D3D로 구현한 것입니다. D3D는 화면에 장면을 연출하기 위해서 크게 Rasterization(래스터 변환 과정)까지의 Vertex Processing(정점 처리 과정)과 이후의 Pixel Processing(픽셀 처리 과정)을 수행합니다.

D3D를 이용하는 프로그래머는 반드시 Vertex Data(정점 데이터)와 Primitive Data(프리미티브 데이터)를 입력해야만 화면에 출력할 수 있습니다. 정점 데이터와 프리미티브 데이터는 이후에 자세히 설명해 드리겠지만 일단 여러분은 정점 데이터를 생명을 구성하는 분자, 프리미티브 데이터는 생명의 기본 단위인 세포로 생각하기 바랍니다. 이렇게 제안하는 이유는 생명체의 기본단위는 세포 이듯이 렌더링의 기본단위는 프리미티브 데이터 이고, Primitive는 Vertex Data를 조직화 해서 만들기 때문입니다.

버텍스 데이터와 프리미티브 데이터는 최초로 Tessellation 과정을 거치는데 Tessellation라는 용어는 크고 작은 같은 모양의 도형들을 이용하여 어떤 면이나 공간을 빈틈 없이 메우는 미술의 장르를 지칭합니다. 예를 들면 조각보를 이용한 미술 작품이라든가 성당의 타일 작품 등이 있고, 이것을 회화에 적극적으로 이용한 사람은 에셔(M.C. Escher)인데 이분의 그림은 “미학 오디세이-진중권” 1편에 자주 이용되고 있으니 도서관에서 한 번 대출해 읽어보기 바랍니다.

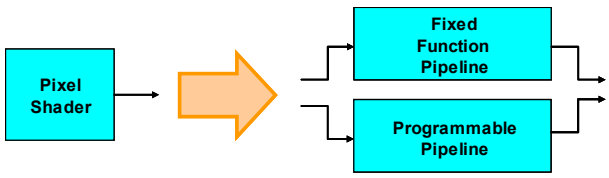
Tessellation 과정 이후 Fixed Function Pipeline(고정 기능 파이프라인: 고정 파이프라인) 또는 Programmable Pipeline(프로그램 가능한 파이프라인) 과정을 거치게 됩니다. Pipeline이라는 것은 컴퓨터 공학에서 말하는 명령어의 처리 흐름입니다. 3D 장면처리는 마치 공장에서 우유를 만들 듯 규격화된 일정한 과정을 순서대로 처리하는데 이것을 그래픽 파이프라인이라 부르며 그냥 줄여서 파이프라인이라고도 합니다. Fixed Function Pipeline은 우유에 초코, 딸기 등 첨가 요소 결정을 말하는데 그래픽 파이프라인에 첨가할 것은 빛에 대한 광원의 종류, 빛의 세기, 안개 적용 값, 이미지 적용 값 등이 정해진(고정된) 방법에 의해서 설정하고 그 처리 또한 정해진 공식을 통해서

처리되는 파이프라인입니다.
이와 대비되는 것이 Programmable Pipeline입니다. 이것은 GPU를 Assembly나 고급 언어를 이용해서 프로그래머가 처리의 과정을 조작하는 것인데 이것을 구현한 것이 바로 Shader 입니다. 현재의 Shader는 GPU 전체를 프로그램 할 수 있지 않고, 파이프라인의 일부만 가능합니다.
프로그램 가능한 파이프라인(Shader)을 이용할 때 가장 주의할 점은 주어진 값을 적용하는 방법이 유연한 것이지 장면을 연출하는 그래픽 파이프라인의 순서 자체는 바꿀 수 없고 고정 파이프라인과 동일 합니다. 그리고 현재의 Shader는 파이프라인 전체를 프로그램 할 수 있는 것이 아니라 일부만 프로그램 할 수 있어서 프로그램 가능한 파이프라인을 Flexible Pipeline 으로 부르기도 합니다. (저는 유연한 파이프라인이라는 것이 더 좋습니다.)

Fixed Function Pipeline 또는 Programmable Pipeline에서는 입력된 정점 데이터와 프리미티브 데이터를 이용해서 3차원 데이터를 2차원 화면 데이터로 전환하고, 안개나 조명 효과를 처리합니다. 이 과정을 거치고 나서 비로소 Pixel 단위로 바꾸는 작업을 하게 되는데 이것을 래스터 처리 이라 합니다. 이 과정이 GPU에서 가장 많은 일을 하는 과정인데 속도를 높이기 위해서 Pixel을 만들 수 있는 삼각형만 고르는 은면 제거(Back Face Culling)와 렌더링 화면 영역을 벗어나는 픽셀들은 제거하는 화면 Clipping을 수행합니다.
이 과정까지 정점 처리 과정이라 하고 이후에는 래스터 과정을 통해서 만들어진 픽셀 데이터, 프로그래머가 지정한 텍스처 데이터(이미지) 등을 혼합해서 화면과 동일한 크기와 해상도를 갖는 버퍼에 출력하는 픽셀 처리 과정인데 픽셀 처리 과정에서 처음으로 하는 것은 이미지에서 픽셀의 추출해서 텍스처를 만듭니다. 이렇게 추출하는 과정을 Sampling 이라 하고, 샘플링 데이터를 지정한 방식(Addressing Mode)에 따라 보간(Interpolation)을 수행하는 필터링(Filtering)합니다.

텍스처와 이전에 래스터 과정으로 만든 픽셀을 혼합하는 Multi-Texturing 과정을 거친 후, 출력에 쓰여져 있는 픽셀과의 비교인 Alpha Test, Depth Test, Stencil Test를 거쳐 최종 버퍼의 픽셀 데이터를 갱신합니다. 여기서 Test라는 것은 출력 버퍼의 내용을 갱신 여부와 갱신의 방법에 대한 Test이며 Alpha, Depth, Stencil Test 중 하나라도 실패하면 출력버퍼의 픽셀 값을 갱신하지 않습니다.

위의 그림은 DirectX9_SDK 2003 summer의 도움말에서 얻은 것인데 픽셀과정에서의 Pixel Shader 부분도 정점 처리 과정과 비슷하게 고정 기능 파이프라인과 프로그램 가능한 파이프라인으로 수정되어야 현재의 D3D 그래픽 파이프라인과 맞습니다.



<D3D 그래픽 파이프라인 수정 구조>

앞서 프로그램 가능한 파이프라인이 GPU 일부만 프로그램 하는 것이라 했었는데 픽셀 처리 과정의 샘플링, 필터링, 텍스처링 정도만 현재로서는 Pixel Shader를 통해서 프로그램 할 수 있습니다.
물론 정점 처리 과정과 마찬가지로 그래픽 파이프라인의 순서를 바꿀 수는 없습니다.

D3D 그래픽 파이프라인의 구조를 정리하면 처리 과정은 정점 처리 과정과 픽셀 처리 과정으로 나누어 진행되는데 입력된 정점 데이터는 제인 먼저 Transform(변환)과정을 거칩니다. 변환이라는 것은 장면을 구성하는 3차원 데이터를 수학적 처리 과정을 통해서 2차원 화면 장치로의 값으로 바꾸는 것입니다. 쉽게 말한다면 3차원 공간의 정점 위치를 2차원 화면 좌표계로 바꾸는 것입니다.
이 때 정점의 위치는 월드 변환 → 뷰 변환 → 정규 변환 → 장치 의존 변환 순으로 바뀝니다.
변환이 끝나고 나면 프로그래머가 지정한 정점 데이터를 이용해서 픽셀을 만드는데 이때 프리미티브를 이용합니다. 프리미티브는 렌더링의 기본 단위라고 앞서 이야기 했는데 D3D의 프리미티브는 크게 점, 선 삼각형 세 종류가 있으며 프리미티브는 같은 정점을 가지고도 점, 선, 삼각형 등 서로 다른 프리미티브를 구성할 수 있습니다.
변환을 거친 정점의 위치, 프리미티브 정보, 광원 효과 변수들, 안개 효과 변수들을 이용해서 목적 화면에 대한 픽셀을 만드는 것이 래스터 과정이며 앞서 이야기 했듯이 처리 속도를 높이기 위해 은면 제거나 화면 클리핑(Clipping)등을 수행합니다.
픽셀 처리 과정은 주어진 이미지에서 샘플링과 어드레싱(Addressing), 픽셀을 보간하는 필터링을 통해서 텍스처를 만들고 이 텍스처와 래스터 이후에 만들어진 픽셀들을 혼합하는 멀티텍스처링(Multi-Texturing)을 거치고 나서 마지막으로 출력 버퍼의 색상과의 교체 여부인 알파 테스트, 깊이 테스트, 스텐실 테스트를 수행한 후 통과한 픽셀을 출력버퍼에 쓰게 됩니다.
이러한 D3D의 그래픽 파이프라인은 하드웨어인 GPU에서 처리가 될 수 있도록 HAL제공 하고, 하드웨어가 지원이 안되더라도 CPU에서 처리할 수 있도록 REF 또한 지원합니다.
Software Rendering은 모든 그래픽 장치에 독립적인 코드를 만들 수 있지만 3D 렌더링 전 과정을 CPU에서 처리하기 때문에 속도가 느리고 특히 래스터 과정과 픽셀 처리부분에서 상당히 과부하가 생깁니다.
이와 반대로 Hardware Rendering을 수행하면 렌더링이 장치에 민감하게 의존하지만 하드웨어에서 3D 처리가 이루어 지므로 렌더링 속도의 이득이 있으며 D3D의 HAL을 사용하면 장치에 독립적인 코드를 만들 수 있습니다.
고정 기능 파이프라인은 렌더링 과정과 이에 필요한 변수들을 함수를 통해서 설정하는 것이고 프로그램 가능한 파이프라인은 고정 기능 파이프 라인의 일부를 저급언어 또는 고급 언어를 통해서 제어하는 것입니다. 이를 D3D에서는 Shader로 구현 했으며 D3D는 정점 처리과정의 Vertex Shader(정점 쉐이더) 픽셀 처리 과정의 Pixel Shader(픽셀 쉐이더) 두 종류가 있는데 정점 쉐이더, 픽셀 쉐이더를 사용하더라도 정점의 처리와 픽셀의 처리 과정은 고정 기능 파이프라인과 동일합니다.
D3D에서는 가상 머신(Virtual Machine) 줄여서 머신(기계)이라는 개념이 있는데 이것은 그래픽 장치, 또는 디바이스의 개념을 좀 더 추상화 시켜 하드웨어에 좀 더 독립적인 프로그램을 만들 수 있도록 하기 위함입니다. 프로그래머는 기기 내부의 동작 원리를 자세히 알고 있지 않아도 머신

또는 기계 내부에서 필요한 객체 생성, 머신이 동작하기 위해 필요한 변수 설정, 머신의 상태 설정을 함수 등을 통해서 쉽게 설정할 수 있도록 하는 것이 가상 머신입니다.

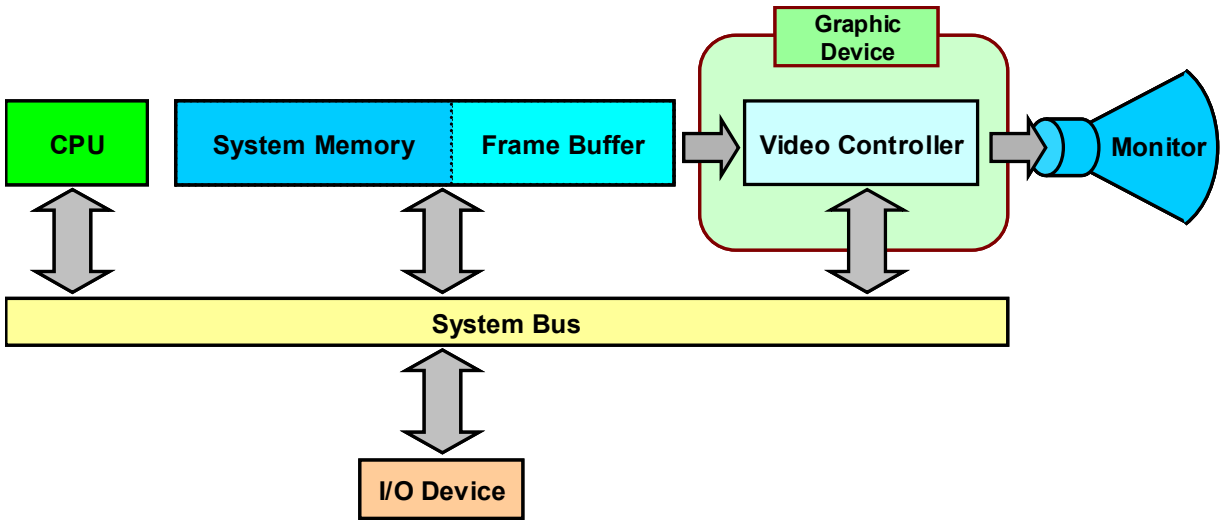
다음의 두 코드는 이미 보았던 2D에서 D3D의 디바이스 생성과 렌더링에 대한 부분과 내용을 동일합니다. 이것을 제시한 이유는 앞으로 3D 프로그램으로 나아가는 길이 길고 그 길의 첫 코드 라는 것입니다. “천리길도 한 걸음부터” 라는 속담이 있듯이 3D 프로그램도 기초부터 하나씩 밟아 나가면서 코드를 구조화 한다면 시간은 좀 들지만 마스터하기가 어렵지 않다는 믿음을 가지고 시작합시다.

m3d_01_overview01_device_c.zip

2. 그래픽 장치

1. 그래픽장치

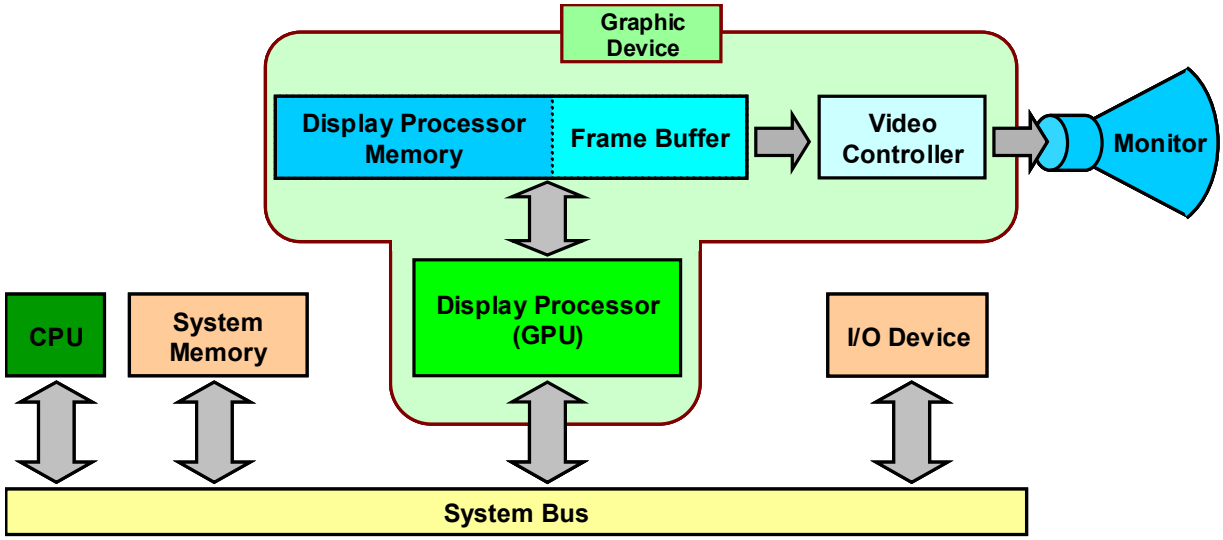
그래픽 카드의 가장 큰 임무는 모니터 제어 입니다. 따라서 그래픽 카드는 모니터의 화면에 완전히 종속입니다. 컴퓨터의 내용을 디스플레이(Display) 하는 방법은 래스터 그래픽스와 벡터 그래픽스가 있습니다. 이중에서 래스터 그래픽스는 현재 개인용 모니터에서 가장 많이 사용되는 방식이며 이장은 래스터 방식 그래픽 장치를 배경으로 진행하겠습니다.
최초의 래스터 방식에서는 간단하게 컴퓨터 구조를 다음과 같이 표현 할 수 있습니다.



<고전적인 그래픽 장치>

그림을 보면 그래픽 장치는 모니터 제어 정도만 사용하고 있습니다. 그리고 모니터에 출력할 픽셀 정보를 시스템 메모리인 RAM의 일부 영역을 모니터 설정 해상도와 동일하게 일치하는 Frame

Buffer로 설정해서 사용하는 것이 가장 큰 특징입니다.
여기서 Frame Buffer는 최종 모니터에 출력하는 픽셀과 이에 관련된 모든 정보라고 말할 수 있는데 래스터 방식의 그래픽 장치 모니터 화면을 동일한 크기의 화소(픽셀)로 나누어서 출력을 합니다. 이 때 보통 가장 많이 사용되는 자료 구조가 비트맵입니다. 비트맵 파일인 BMP파일은 바로 출력하는 화면의 화소와 비트 단위로 대응 되며 비트맵 파일의 색상 위치는 화면 색상과 일치한다고 볼 수 있습니다.
위 그림의 컴퓨터 구조에서 CPU는 모니터에 출력할 데이터를 프레임 버퍼에 저장을 하고 비디오 제어기는 프레임 버퍼의 내용을 모니터로 출력하는 정도의 간단한 역할 정도였는데 이러한 구조는 3D를 표현하기가 무척 어렵습니다.
세월이 흘러 회로를 구성하는 소자의 가격이 낮아짐에 따라 그래픽 카드는 CPU에서 처리하던 그래픽 관련 일들을 직접 처리하는 구조로 바뀌어 갑니다.



<현재의 컴퓨터 시스템과 그래픽 장치>

이전의 단순한 구조에서 그래픽 장치는 Display Processor 라는 GPU와 렌더링 속도의 향상을 위해서 시스템 메모리가 아닌 그래픽 장치 내의 메모리를 가지고 있는데 여기서 퀴즈 하나 내겠습니다. 시스템 메모리를 할당하는 방법은 무엇일까요?

답) 정적 할당, 동적 할당 두 가지 방법이 있습니다. 물론 기억 공간에 따라 자동 변수는 Stack, 전역 변수와 Static 변수는 Global에 new연산자나 동적 할당 함수를 통해서 Heap에 메모리를 할 수 있지만요.

그럼 다음 문제 위의 그림에서 Display Processor Memory에 할당하는 방법은 무엇일까요?

답은 여러분이 D3D를 이용하지 않고는 얻을 수 없습니다. 즉, D3D Device 객체의 함수를 통해서만 그래픽 장치의 메모리를 사용할 수 있습니다. 예를 들면 2D 게임에서 텍스처를 만들 때 D3DCreateTextureFromFileEx() 함수를 이용했는데 이 함수는 Device의 CreateTexture() 라는 멤버 함수로 만들 수 있으며 이 때 함수의 전달 인수에 따라 D3D는 비디오 메모리, 또는 시스템 메모리에 텍스처를 할당합니다.
이렇게 D3D통해서만 디바이스의 메모리를 이용할 수 있고 이 메모리들은 Lock()/Unlock()함수를 통해서만 접근할 수 있다는 것을 꼭 기억하기 바랍니다.

2. Frame Buffer

2.1 Surface(서피스)

그래픽 카드는 모니터에 색상을 출력하기 위한 프레임 버퍼라는 메모리를 이용합니다. D3D는 이 프레임 버퍼를 크게 Front Buffer(전면 버퍼)와 Back Buffer(후면 버퍼) 두 개를 이용하는데 전면 버퍼는 현재 모니터에 출력하고 있는 색상을 저장하는 버퍼고, 후면 버퍼는 다음에 렌더링 할 색상을 저장하고 있는 기억 공간입니다.
이 버퍼들은 서피스(Surface)로 구성되어 있습니다. 서피스의 정의는 기하 정보(Geometry), 이미지 처리에 의해 생성된 픽셀들의 정보를 보관하고 있는 장소라 할 수 있습니다.
픽셀들의 정보를 보관하고 있다면 모두 서피스를 포함 한다고 말할 수 있습니다. 즉, 프레임 버퍼 이외에 텍스처도 서피스를 가지고 있다고 볼 수 있습니다. 보통 픽셀은 2D 만 있다고 생각하기 쉬운데 3D에서는 픽셀이 2D 뿐만 아니라 1D, 3D도 있음을 기억하기 바랍니다.
서피스는 색상 이외에 D3D는 장면의 가상 카메라에 대한 거리인 깊이 값도 가질 수 있습니다.이 서피스를 깊이 버퍼라 부릅니다. 서피스는 또한 스텐실이라는 효과에서 사용됩니다. 이 때 이 서피스를 스텐실 버퍼라 부릅니다.

다음 예는 텍스처와 디바이스에서 서피스를 가져오는 코드 입니다.

```
// 텍스처 서피스
IDirect3DSurface9* pSfc;
pTex->GetSurfaceLevel(0, &pSfc);
...
pSfc->Release();


// 디바이스 색상 버퍼
IDirect3DSurface9* pSfc;
m_pd3dDevice->GetBackBuffer( 0, 0, D3DBACKBUFFER_TYPE_MONO, &pSfc);
```

```
//m_pd3dDevice->GetColorBuffer(0, &pSfc);

...
pSfc->Release();


// 디바이스 깊이-스텐실 버퍼
m_pd3dDevice->GetDepthStencilSurface( &pSfc);
...
pSfc->Release();
```

모든 서피스의 데이터는 픽셀과 일치하므로 서피스에서 사용하는 단위는 3D 서피스의 경우 너비(Width), 높이(Height), 깊이(Depth) 단위를 사용합니다.

서피스 인터페이스에서 자주 사용되는 함수는 GetDesc(), LocRect()/UnlockRect() 함수 입니다.
GetDesc - 서피스에 대한 정보인 서피스 정보 구조체(D3DSURFACE_DESC) 정보를 반환 해줍니다.
LockRect - 다를 프로세스가 침범하지 못하도록 서피스 메모리 사용에 대한 독점과 픽셀이 저장되어 있는 메모리 포인터 가져옵니다.
UnlockRect - Lock에 의한 서피스 사용에 대한 독점 모드를 해제합니다.

```
EX)
D3DSURFACE_DESC surfaceDesc;
pSfc->GetDesc(&surfaceDesc);
…
D3DLOCKED_RECT lockedRect;
pSfc->LockRect(&lockedRect, 0, 0);
…
DWORD* pImageData = (DWORD*)lockedRect.pBits;           // 이미지 포맷에 맞게 casting 필요

INT nPixelByte = lockedRect.Pitch/surfaceDesc.Width;    // 픽셀당 바이트 구하기
if(1 == nPixelByte)
        pImageData = (BYTE*)lockedRect.pBits;
else if(1 == nPixelByte)
        pImageData = (WORD*)lockedRect.pBits;
else if(4 == nPixelByte)
```

```
        pImageData = (DWORD*)lockedRect.pBits;
...
pSfc->UnlockRect();
```

위의 코드에서 pImageData을 DWORD형 포인터로 캐스팅을 했습니다. 정확한 캐스팅은 LockRect() 함수에서 얻은 D3DLOCKED_RECT 구조체의 Pitch 값을 D3DSURFACE_DESC 구조체의 Width로 나누어 한 픽셀의 바이트를 구한 다음 이에 적절한 캐스팅을 수행해야 합니다.

2.2 멀티 샘플링(Multisampling)

필요한 데이터를 특정한 간격으로 데이터를 추출하는 것을 샘플링(Sampling: 표본화) 이라 합니다. 샘플링 간격이 좁으면 원본 데이터와 근접하지만 데이터의 크기가 증가합니다. 반대로 샘플링 간격이 넓으면 표본화한 데이터는 줄어들지만 원본에 비해서 출력의 질이 낮아집니다. 따라서 적절한 간격을 설정해야 하는데 이 때 Nyquist Sampling Frequency (나이키스트 표본화 율)을 이용합니다. 이것을 이용하면 표본화 된 데이터에서 원본을 그대로 재현할 수 있습니다. 참고 삼아 나이키스트 표본화 율 계산은 다음과 같습니다.

f_sampling = 2 f_max (f_max :Sampling 대상에서 나타나는 최대 주파수)

이것을 샘플링 간격에 적용하면 원본을 그대로 재생할 수 있습니다.

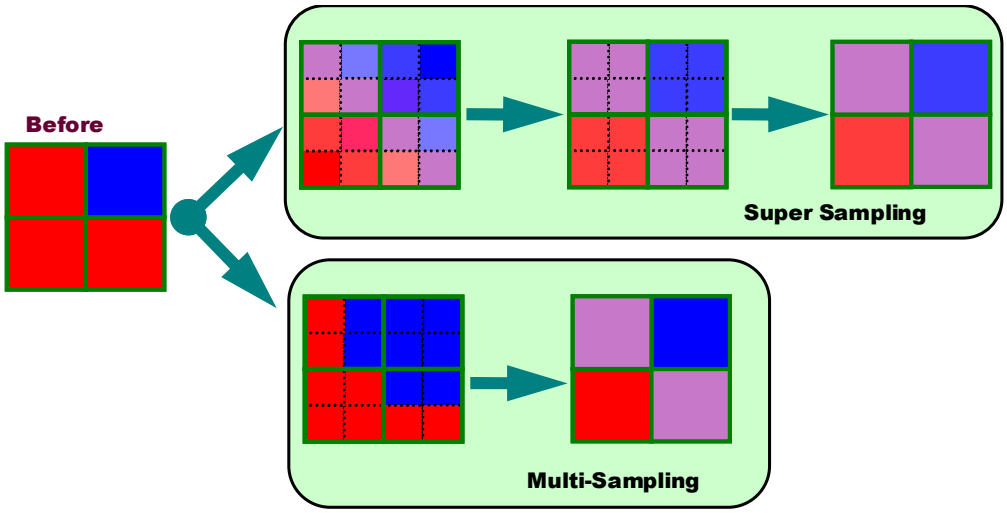
Δsampling = 1/fmax * 0.5

음성 같은 경우 이 것을 많이 이용하는데 안타깝게도 모니터의 크기는 그리 가변적이지 못합니다. 따라서 낮은 단계로의 표본화인 Under Sampling를 수행해야만 이 과정에서 원하지 않는 픽셀들의 계단 현상과 같은 발생하는 경우가 있습니다. 이것을 Aliasing이라 하며 Aliasing 을 줄이는 방법을 Antialiasing이라 합니다. 래스터 방식의 모니터에서는 거의 Aliasing 문제를 갖고 있고, 이것을 조금이라도 줄이기 위해서 최근에 여러 그래픽 카드사들이 Super Sampling (Post Filtering)을 이용하고 있습니다.

수퍼 샘플링은 Aliasing문제를 해결하기 위해 먼저 화면에 표시할 픽셀의 적절한 값을 설정하기 위해 고 해상도에서 렌더링 물체를 표본화(Sampling)한 후 최종 픽셀 값을 결정합니다. 다음으로 하위 픽셀들은 Sampling 이론을 근사한 값들을 이용해 픽셀을 결정하고 마지막으로 하위 픽셀들을 모아서 최종 픽셀 계산 합니다.

멀티 샘플링은 수퍼 샘플링의 한 방법으로 수퍼 샘플링을 간소화 시켰다 할 수 있습니다. 모든 그래픽 카드가 수퍼 샘플링 방식으로 만들어져 있다면 개인이 이를 구입하는 것은 부담이 커서 그래픽 카드 회사는 몇 개의 단계를 단순화 시킨 멀티 샘플링 방식 카드로 만들어 판매하고 있습니다.

수퍼 샘플링과 멀티 샘플링은 이 책의 내용을 벗어나므로 다음 그림을 통해서 "아 이런 것이구나" 하고 넘어가고 나중에 3D 실력이 쌓이면 그 때 다른 3D 관련 책이나 논문 등을 참고하기 바랍니다.



<수퍼 샘플링과 멀티 샘플링>

하드웨어의 멀티샘플링 능력을 알아보기 위해서 IDirect3D9::CheckDeviceMultiSampleType() 함수를 사용해서 다음과 같이 작성합니다.

```
// Present Parameter 구조체의 Back Buffer 포맷 설정
D3DPRESENT_PARAMETERS d3dpp={0};
...
// for 루프로 멀티샘플링의 최대 레벨 검사
DWORD dQualityLevels;

for(int nType=D3DMULTISAMPLE_16_SAMPLES; nType>=0; --nType)
{
	if(SUCCEEDED(m_pD3D->CheckDeviceMultiSampleType(D3DADAPTER_DEFAULT
					, D3DDEVTYPE_HAL
					, d3dpp.BackBufferFormat
					, TRUE
					, (D3DMULTISAMPLE_TYPE)nType
					, &dQualityLevels)))
	{
		d3dpp.MultiSampleType		= (D3DMULTISAMPLE_TYPE)nType;
		d3dpp.MultiSampleQuality	= dQualityLevels-1;
```

```
                break;
        }
}

// 디바이스를 생성한다
m_pD3D->CreateDevice(…, &d3dpp, &m_pd3dDevice );
```

Quality Level은 하나의 최종 픽셀에 대한 멀티 샘플링의 하위 레벨 단위입니다. 이 값이 클수록 Antialiasing이 적용되어 좀 더 부드러운 화면이 만들어 지지만 멀티샘플링으로 인한 렌더링 속도의 저하가 옵니다. 적절한 레벨은 테스트 해본 결과 4 ~ 8 레벨 사이이고, 이 레벨에서도 렌더링 속도가 좋다면 점점 더 레벨을 올리도록 합니다.

2.3 픽셀 포맷

앞서 서피스는 색상, 깊이, 스텐실 등에서 사용된다고 했습니다. 서피스의 역할에 따라 픽셀 포맷을 설정합니다.

색상으로 사용되는 서피스에서 가장 많이 사용되는 포맷은 다음과 같습니다.

D3DFORMAT

D3DFMT_R8G8B8 - 24bit. R: 8bit, G: 8Bit, B: 8Bit

D3DFMT_X8R8G8B8 - 32bit. X는 사용하지 않음

D3DFMT_A8R8G8B8 - 32bit. A: alpha 8Bit

D3DFMT_X1R5G5B5 - 16bit.

D3DFMT_R5G6B5 - 16bit.

D3DFMT_A1R5G5B5 - 16bit. Alpha 1Bit

D3DFMT_A16B16G16R16F - 64bit 부동소수점 포맷.

D3DFMT_A32B32G32R32F - 128bit 부동소수점 포맷

이 중에서 **D3DFMT_X8R8G8B8**(화면 해상도 32비트), D3DFMT_R5G6B5(화면 해상도 16비트) 타입이 프레임 버퍼의 색상 포맷으로 가장 많이 사용되며 이외의 타입들은 텍스처에서 주로 사용되는 포맷입니다. D3DFMT_A16B16G16R16F, D3DFMT_A32B32G32R32F 포맷은 픽셀이 64, 128 비트 형식으로 구성되어 있어서 높은 정밀도를 요구하는 하이 다이내믹 레인지(HDR: High Dynamic Range) 렌더링에서 주로 이용 되는 포맷입니다.

만약 서피스가 깊이와 스텐실 용으로 사용될 경우는 다음과 같습니다.

깊이 버퍼의 포맷은 깊이 테스트에 대한 정확도와 일치합니다. 대부분 게임에서 24bit 깊이이면

충분합니다.

D3DFMT_D32 - 32bit 깊이 버퍼
D3DFMT_D24S8 - 32bit 중 24bit 깊이, 8bit 스텐실 버퍼
D3DFMT_D24X8 - 32bit 중 24bit 깊이 버퍼, 8bit 사용 안함
D3DFMT_D24X4S4 - 24bit 깊이, 4bit 스텐실, 4bit 사용 안함
D3DFMT_D16 - 16bit 깊이 버퍼

가장 많이 이용되는 포맷은 **D3DFMT_D24S8** 형식입니다. 현재의 게임에서 D3DFMT_D16 형식은 거의 사용되지 않습니다.

2.4 디바이스 능력(Capability)과 D3DPRESENT_PARAMETERS 구조체
테스트 코드에서 출발한 프로그램이 점차 완성도 있는 게임 프로그램으로 전환됨에 따라 예상하지 못했거나 아니면 전혀 도움을 받을 수 없는 문제가 발생하는 경우가 있습니다. 예를 들면 PC에서는 되는데 노트북에서는 안 되는 경우 이럴 때 무엇부터 디버깅 해야 할지 막막합니다.
아무리 찾아보아도 못 찾았을 때 그래픽 카드의 성능을 의심해 볼 필요가 있습니다. 개인용 컴퓨터의 그래픽 카드는 고 성능이 이지만 노트북은 절전을 위해서 PC보다 저 사양 그래픽 카드가 많이 장착됩니다.
만약 하드웨어의 성능 차이로 문제가 생겼다고 생각이 들면 D3D의 IDirect3D9::GetDeviceCaps() 함수나 디바이스의 IDirect3DDevice9:: GetDeviceCaps() 함수를 통해서 해당 장치의 성능을 찾고 비교해 봅니다.

```
// 디바이스 능력
D3DCAPS9 pCaps;
// D3D
m_pD3D->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &pCaps);

// Device
m_pd3dDevice->GetDeviceCaps(&pCaps);
```

특히 쉐이더와 최대 이미지 크기는 이 코드를 이용해서 꼭 확인해야 합니다. 종종 2D에서도 화면 스크롤을 편리하게 작성하기 위해 큰 이미지를 사용할 때가 있고, 이 이미지의 크기가 장치에서 이용될 수 있는지 반드시 확인이 필요합니다.

D3DPRESENT_PARAMETERS는 장면의 프레임 버퍼 스펙(Specification) 을 결정하는 구조체 입니다.

이 구조체의 변수와 설명은 다음과 같습니다.

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT            BackBufferWidth;   // 후면 버퍼 너비. 0이면 DC의 너비
    UINT            BackBufferHeight;  // 후면 버퍼 높이. 0이면 DC의 높이
    D3DFORMAT       BackBufferFormat;  // 후면 버퍼 픽셀 포맷. 모르면 D3DFMT_X8R8G8B8
    UINT            BackBufferCount;   // 더블 버퍼링을 위한 후면 버퍼 수. 보통=1, 0이면 1
    D3DMULTISAMPLE_TYPE MultiSampleType; // 후면 버퍼에 이용할 멀티 샘플링의 타입
    DWORD           MultiSampleQuality;        // 멀티 샘플링 레벨
    D3DSWAPEFFECT   SwapEffect;    // Swap chain 교환 방법 보통 = D3DSWAPEFFECT_DISCARD
    HWND            hDeviceWindow;             // 서비스의 윈도우 핸들. 반드시 지정
    BOOL            Windowed;                  // 윈도우 모드: TRUE, Full Window: FALSE
    BOOL            EnableAutoDepthStencil;    // 자동으로 깊이-스텐실 버퍼 생성 보통 = TRUE
    D3DFORMAT       AutoDepthStencilFormat;    // 깊이/스텐실 버퍼의 포맷  보통 = TRUE
    DWORD           Flags;                     // Flag 값 보통=0
    UINT            FullScreen_RefreshRateInHz;
        // 재생률. 윈도우 모드=0, Full Mode: 지원되는 Hz 수동 지정
    UINT     PresentationInterval;
        // Present 간격. D3DPRESENT_INTERVAL_IMMEDIATE : 즉시 처리
        // D3DPRESENT_INTERVAL_DEFAULT : 주사선이 다 그리고 나서 초기 위치로 이동할 때 복사
}D3DPRESENT_PARAMETERS;
```

2.5 메모리 풀(Memory Pool)

비디오 메모리를 사용하려면 D3D를 통해야만 한다고 했습니다. 이것은 D3D 인터페이스에 의해 생성된 자원은 D3D에 의해 관리되기 때문입니다. D3D는 사용자가 요청한 자원을 메모리 풀로 관리를 합니다.

DirectX 9.0의 D3D 메모리 풀은 Default, Managed, System, Scratch 4종류가 있습니다.

D3DPOOL_DEFAULT: 자원의 타입과 이용에 가장 적합한 메모리에 보관하도록 D3D에 요청합니다. 대부분 비디오 메모리나 AGP를 이용이 되며 가장 빠르게 동작하는 메모리 풀입니다. 사용 예를 들면 Post Effect에서 화면 전체를 저장하는 텍스처가 필요할 때 Default를 이용하면 다른 메모리 풀을 이용하는 것보다 렌더링 속도가 좋습니다. 이 메모리 풀은 빠르게 동작하는 장점이 있지만 자원은 디바이스의 IDirect3DDevice9:: Reset() 함수 호출 이전에 반드시 해제, Reset() 함수 호출 이후에 다시 생성해야 합니다.

D3DPOOL_MANAGED: 디바이스가 필요하다면 자원을 복사해서 백업을 만듭니다. 디바이스가 Lost 상

태에서도 자원을 다시 재 생성 필요가 없습니다. 가장 많이 사용되는 메모리 풀로 노트북과 같이 시스템 메모리와 비디오 메모리를 공유해서 사용하는 장치에 적합하며 메모리 풀의 지정이 마땅하지 않을 때 기본적으로 이 메모리 풀로 먼저 이 지정하는 것이 좋습니다.

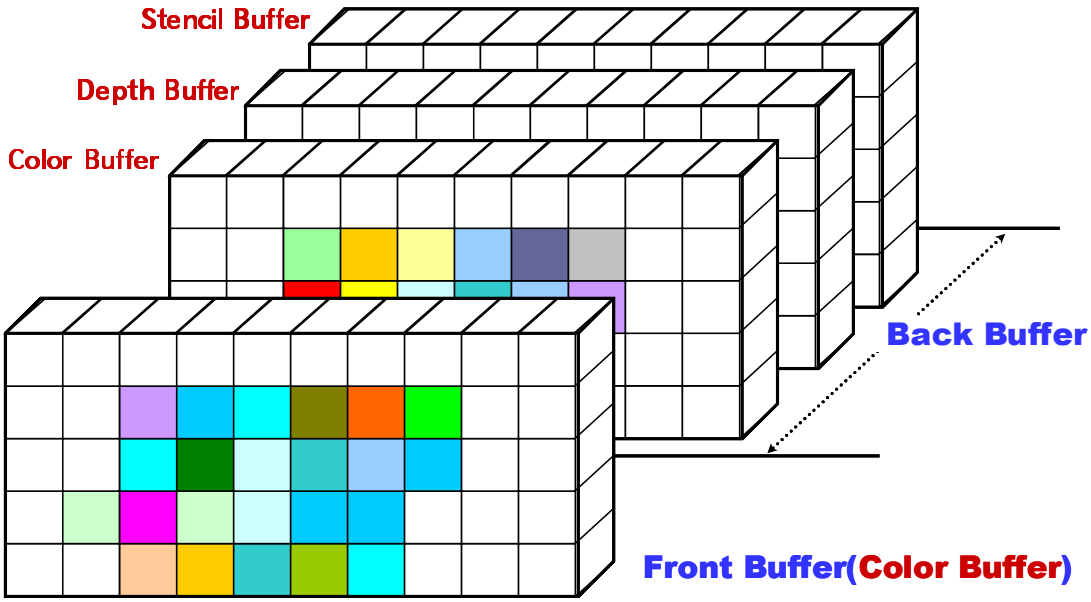
D3DPOOL_SYSTEMMEM: 일반적으로 3D 장치가 접근할 수 없는 메모리 영역으로 시스템 메모리에 자원을 생성합니다. 때때로 PDA에서 Direct3DMobile을 이용할 때 이 메모리 풀을 요구하는 경우가 많습니다.

D3DPOOL_SCRATCH: 자원을 시스템 메모리에 생성합니다. 장치의 제한을 따르지 않습니다. 장치를 통해 자원 접근이 어려우나 생성과 Lock, 복사는 가능합니다.

개인용 컴퓨터에서 속도를 요구하는 텍스처를 실시간으로 만들 때 D3DXCreateTexture() 함수를 이용하는데 이 때 먼저 D3DPOOL_DEFAULT로 시도해 보고 실패하는 경우에는 D3DPOOL_MANAGED를 메모리 풀로 지정해 봅니다. D3DPOOL_MANAGED 방식으로도 실패한다면 프로그램을 종료 시키는 것이 좋습니다.
파일에서 읽어서 텍스처를 만드는 경우 D3DPOOL_MANAGED가 가장 무난합니다. D3DPOOL_DEFAULT이 좋지만 의도하지 않는 화면의 전환에서 디바이스를 다시 리셋(Reset)해야 할 때도 MANAGED 방식은 이러한 고민 없이 사용이 가능해 게임 시스템이 복잡해 지지 않습니다.

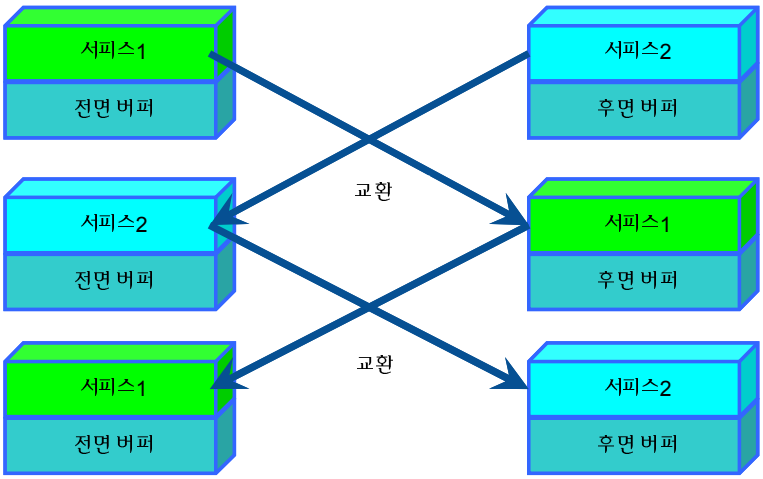
2.7 Frame Buffer
프레임 버퍼는 모니터와 같은 비디오 장치에 색상을 출력하기 위한 여러 버퍼들의 집합, 재생버퍼의 집합입니다.
D3D는 개념 상으로 5개의 버퍼, Front Buffer(전면 버퍼), Back Buffer(후면 버퍼), Color Buffer(색상 버퍼), Depth Buffer(깊이 버퍼), Stencil Buffer(스텐실 버퍼)로 구성되어 있습니다. D3D가 실제로 구현한 버퍼는 깊이 버퍼와 스텐실 버퍼를 하나의 버퍼에서 나누어 사용하고 있고, 전면 버퍼는 이중 버퍼링(Double Buffering)을 적용하면 색상 버퍼를 이용하게 되어 3개의 버퍼라 말할 수 있습니다.
전면 버퍼는 현재 비디오 장치에 출력되고 있는 픽셀 메모리 영역입니다. 이 버퍼는 색상 버퍼의 한 종류 입니다. 과거에는 프로그래머가 전면 버퍼의 메모리를 접근할 수 있으나 현재는 막아놓았습니다.
후면 버퍼는 속도를 향상 시키기 위해 전면 버퍼와 동일한 색상 정보를 저장하고 있는 메모리 영역으로 프로그래머는 이 버퍼에 접근해서 값들을 변경할 수 있습니다. 후면 버퍼는 다시 3종류의 버퍼인 색상, 깊이, 스텐실 버퍼로 구성 되어 있습니다.
색상 버퍼는 픽셀을 저장하고 있는 공간 입니다.
깊이 버퍼는 새로운 색상 값(픽셀)을 색상 버퍼에 덮어 쓸 것인가에 대한 판별 값인 깊이 값을 가지고 있는 버퍼로 파이프라인의 깊이 테스트에 사용됩니다. 또한 가상 카메라의 Z축 값에 의존하

기에 깊이 버퍼 대신 Z Buffer라 부르기도 합니다.
스텐실 버퍼도 깊이 버퍼와 비슷하게 색상 갱신에 대한 판별 값을 저장하고 있는데 이것을 마스킹(Masking) 값이라 합니다. 이 마스킹 값을 이용해서 스텐실 테스트를 파이프라인에서 진행합니다. 주로 이용되는 것은 그림자를 표현하거나 속이 비워있는 3D물체의 절단면 등에서 사용되는 스텐실 효과입니다.



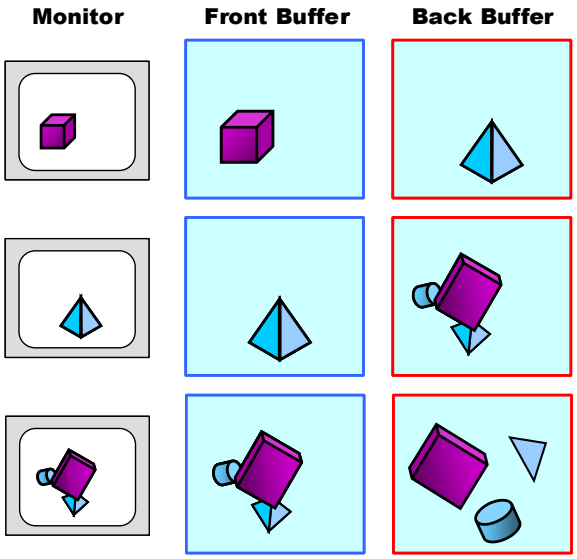
<D3D의 프레임 버퍼 구조>

D3D는 보통 두 개나 세 개의 서피스를 하나의 컬렉션으로 Chain처럼 연결해서 관리를 하고 각각의 버퍼들을 교환해 가면서 렌더링 합니다. 이것을 스왑 체인(Swap Chain)이라 합니다.
이중 버퍼링으로 구성된 프레임 버퍼를 D3D는 다음과 같이 스왑 체인을 구현합니다.



<이중 버퍼링에서의 스왑 체인>

다음 그림은 이러한 스왑 체인의 동작에 대한 설명입니다.



전면 버퍼의 내용이 모니터에 출력 되는 동안 후면버퍼의 내용을 갱신합니다.

주사선이 좌상단 처음으로 이동하는 시간에 후면 버퍼의 내용을 전면 버퍼에 복사하거나 메모리 포인터만 변경합니다. 이것을 플리핑(Flipping) 이라 합니다.

전면버퍼의 내용을 모니터에 출력되는 동안 후면 버퍼의 내용을 다시 갱신합니다.

후면 버퍼의 내용을 특정한 색상 값으로 채우는 함수는 IDirect3DDevice9:: Clear() 함수 입니다. 이 함수의 전달 인수에 대상 버퍼를 각각 설정해서 버퍼를 하나의 값들로 변경하거나, or 비트 연산자 ('|') 로 후면 버퍼들을 묶어서 변경합니다.

색상 버퍼는 D3DCLEAR_TARGET을, 깊이 버퍼는 D3DCLEAR_ZBUFFER를, 스텐실 버퍼는 D3DCLEAR_STENCIL 플래그를 다음 코드처럼 Clear 함수의 전달 인수로 지정합니다.

```
pDevice->Clear( 0, NULL, (D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER|D3DCLEAR_STENCIL), …);
```

주의할 것은 D3DCLEAR_STENCIL은 반드시 스텐실 버퍼가 있을 경우에만 사용해야 합니다.
플리핑은 디바이스의 Present()함수를 사용합니다.

```
pDevice->Present( NULL, NULL, NULL, NULL );
```

2.6 기타
지금까지 내용을 정리하는 과정으로 디바이스의 생성과 화면의 플리핑 코드를 정리해봅시다.

- 1. IDirect3D9 객체 생성

```
pD3D = Direct3DCreate9( D3D_SDK_VERSION );
```

- 2. D3DCAPS9 구조체 이용 장치 특성 확인.

```
// 바탕 화면 해상도 파악(옵션)
D3DDISPLAYMODE d3ddm={0};
pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm );

// 디바이스 능력
D3DCAPS9 pCaps;
pD3D->GetDeviceCaps(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, &pCaps);
```

- 3. D3DPRESENT_PARAMETERS 구조체 설정

```
// 프리젠트 파라메터 구조체 설정
D3DPRESENT_PARAMETERS d3dpp={0};

d3dpp.Windowed                  = TRUE;
d3dpp.SwapEffect                = D3DSWAPEFFECT_DISCARD;
d3dpp.EnableAutoDepthStencil    = TRUE;
d3dpp.BackBufferFormat          = d3ddm.Format;         // 바탕 화면 해상도로 설정할 경우
d3dpp.BackBufferFormat          = D3DFMT_X8R8G8B8;      // 강제로 설정
d3dpp.AutoDepthStencilFormat    = D3DFMT_D24S8;         // Depth-Stencil 포맷 설정
d3dpp.PresentationInterval      = D3DPRESENT_INTERVAL_IMMEDIATE;
```

- 4. IDirect3D9 객체 멤버 함수 CreateDevice를 이용 Device 객체를 생성

```
if( FAILED( m_pD3D->CreateDevice( D3DADAPTER_DEFAULT
        , D3DDEVTYPE_HAL, m_hWnd, D3DCREATE_MIXED_VERTEXPROCESSING
        , &d3dpp, &pDevice ) ) )
        {
                if( FAILED( m_pD3D->CreateDevice(D3DADAPTER_DEFAULT
                        , D3DDEVTYPE_HAL, m_hWnd, D3DCREATE_SOFTWARE_VERTEXPROCESSING
                        , &d3dpp, &pDevice ) ) )
                {
                        pD3D->Release();
                        return -1;
        }
}
```

- 5. 후면 버퍼 클리어(Clear)

```
m_pd3dDevice->Clear(0, NULL, (D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER|D3DCLEAR_STENCIL)
                , D3DCOLOR_XRGB(0,128,200), 1.0f, 0 );
```

-6. 렌더링

```
// BeginScene과 EndScene에 렌더링
if( FAILED( m_pd3dDevice->BeginScene() ) )
        return;

// Rendering of scene objects can happen here

// End the scene
m_pd3dDevice->EndScene();
```

-7. 후면 버퍼 전면 버퍼 교체

```
// 후면버퍼를 전면버퍼로 교체( flipping)
m_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

이것으로 D3D의 개요를 마치겠습니다. 전체 코드는 다음을 참고 하기 바랍니다.

m3d_01_overview01_device_c++.zip