

# 3D Game Programming Basic with Direct3D

## 3. 3D Program 준비

이전 장에서 우리는 3D 그래픽 파이프라인에서 사용되는 벡터, 행렬 등을 살펴보았고, 회전에서 많이 이용되는 사원수도 알아보았습니다. 쉬운 부분도 있고, 어려운 부분도 있는데 모든 것을 다 이해하면 좋겠지만 그렇지 않은 일들이 대부분입니다. 게임프로그램에서는 알고리즘에 대한 완전한 이해 보다 더 중요한 것은 순서입니다.

머리말에서 언급했듯이 게임은 디지털의 종합 예술이라 컴퓨터와 알고리즘에 대해서 게임 프로그래머가 알아야 하는 지식이 상당히 많아서 "좁고 깊게" 보다는 "넓고 얇게" 아는 것이 더 나을 수 있습니다. 이것은 "코드들의 짜 깊이"가 될 수 있지만 잘 합치는 것도 능력이며 예술이라 이론 보다는 실전에, 순수 알고리즘 개발보다는 기존 알고리즘의 응용에 대한 공학적인 측면에 프로그래머는 더 집중해야 합니다.

이 장에서는 앞으로 배우게 될 다양한 3D 내용을 좀 더 편리하게 연습할 수 있도록 기반을 다지는데 목적이 있습니다.

3D 게임 프로그램은 순서만 잘 지키면 sine, cosine 정도만 알고 있어도 얼마든지 잘 만들 수 있습니다. 앞서 이론으로만 그래픽 파이프라인을 구현한 D3D 렌더링 가상 머신에 기초적인 변수의 전달과 처리 방법을 알고 있으면 3D 프로그램이 2D 프로그램보다 약간 높은 난이가 있을 뿐 어렵지 않다는 것을 여러분은 알게 될 것입니다.

이렇게 간단히 가상 머신 사용을 이론과 함께 배운 후에 다음으로 렌더링을 연습할 수 있도록 기초적인 코드들에 대한 구조(Frame Work)을 만들 것입니다. 또한 이 구조(프레임웍: Frame Work)에 3D를 실시간으로 테스트 할 수 있도록 키보드와 마우스 시스템을 구현할 것입니다.

마지막으로 DXSDK에서 보조로 제공하는 다양한 유틸리티(Utility)를 프레임웍에서 연습해 보도록 하겠습니다.

### 3.1. Graphic Pipeline의 변환

#### 3.1.1 정점 데이터(Vertex Data)와 프리미티브 데이터(Primitive)

3D 그래픽 파이프라인에 최초로 정점 데이터와 프리미티브 데이터를 주어야만 화면에 출력이 된다고 했습니다.

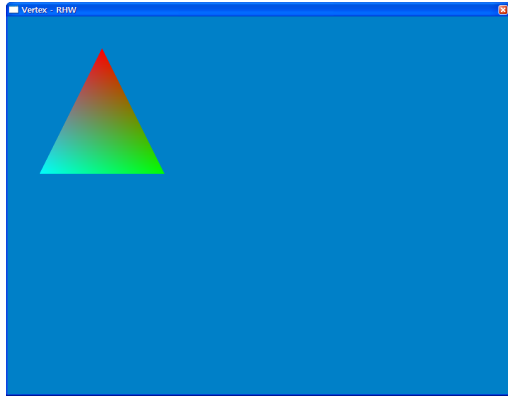
이를 위해서 첫 번째 프로그래머는 최초에 D3D에서 지정하는 방식대로 정점 구조체를 만들어야 합니다.

두 번째는 이 정점 구조체를 가지고 정점 데이터를 D3DDevice(이하 디바이스)를 통해서 만듭니다.

세 번째는 정점 데이터를 디바이스에 연결합니다.

네 번째는 디바이스에게 프리미티브 정보를 전달하면서 렌더링 명령을 내립니다.

이 방식들은 [m3d\\_01\\_overview02\\_vertex.zip](#)의 Main.cpp파일에 구현되어 있습니다. 압축을 풀어 dsp파일을 클릭하고 컴파일 하면 다음과 같이 삼각형 하나가 렌더링 될 것입니다.



소스 코드 Main.h 파일을 열면 다음과 같은 정점 구조체가 선언되어 있음을 볼 수 있습니다.

```
// 정점 구조체
struct VtxRHWD
{
    FLOAT x, y, z, rhw; // 위치
    DWORD color;         // 색상
};
```

이 정점 구조체를 가지고 만든 정점 데이터 주소는 CMain 클래스의 멤버 변수인

```
LPDIRECT3DVERTEXBUFFER9 m_pVB;
```

에 저장되는데 CMain::Init() 함수 구현 코드를 보면 디바이스의 CreateVertexBuffer()

함수 호출로 두 번째 단계인 정점 데이터에 대한 정점 버퍼의 인스턴스가 생성되는 것을 볼 수 있습니다.

```
m_pd3dDevice->CreateVertexBuffer( 3*sizeof(VtxRHWD),
                                   0, D3DFVF_VTXST,
                                   D3DPOOL_MANAGED, &m_pVB, NULL );
```

이렇게 D3DDevice를 통해서 생성한 자원은 직접 접근을 못하고 함수 호출을 통해서 접근해야 합니다

다. 정점 버퍼의 데이터는 정점 버퍼의 함수를 이용해서 다음과 같이 Lock() 함수로 독점 권을 얻은 다음, 버퍼의 내용을 가져오거나 아니면 갱신한 후에 Unlock() 함수를 통해서 독점 권을 해제합니다.

```
VtxRHWD vertices[] =
{
    { 150.0f,  50.0f,  0.5f,  1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f,  0.5f,  1.0f, 0xff00ff00, },
    {  50.0f, 250.0f,  0.5f,  1.0f, 0xff00ffff, },
};

VtxRHWD* pVertices;
if( FAILED( m_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 ) ) )
    return E_FAIL;

memcpy( pVertices, vertices, sizeof(vertices) );
```

정점 버퍼의 내용을 갱신하는 예는 CMain::FrameMove() 함수에서도 볼 수 있습니다. 여러분은 지금 보고 있는 예제를 실행해서 Left, Right, Up, Down키를 눌러 보면 삼각형이 이동하고 있음을 볼 수 있는데 CMain::FrameMove() 함수에 이 부분이 구현되어 있으니 살펴보기 바랍니다.

이렇게 정점 데이터(버퍼)를 만들고 나서 디바이스에 연결하는 세 번째 단계 입니다.

디바이스는 여러 개의 정점 데이터를 렌더링 해야 하므로 매번 루프를 돌면서 정점들을 연결하고 렌더링 해야 합니다. CMain::Render() 함수가 구현된 코드에서 BeginScene()/EndScene() 사이에 SetStreamSource()와 SetFVF() 함수 호출을 볼 수 있습니다.

디바이스의 SetStreamSouce() 함수는 정점 버퍼를 연결할 때 사용하는 함수 입니다. 그런데 디바이스는 다양한 정점 구조를 지원합니다. 디바이스가 정점을 올바르게 처리할 수 있도록 지금 연결하는 정점의 구조 형식을 프로그래머는 SetFVF() 함수를 호출해서 정점의 형식을 알립니다. 여기서 FVF는 Flexible Vertex Format 의 약자로 번역하면 "유연한 정점 포맷" 입니다.

```
// 정점을 디바이스에 연결
m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VtxRHWD) );
// 정점 포맷을 설정
m_pd3dDevice->SetFVF( D3DFVF_VTXST );
```

정점 데이터와 정점의 형식을 디바이스에 연결하고 나서 마지막 네 번째 단계인 렌더링 입니다. 렌더링은 디바이스의 DrawPrimitive() 함수 호출로 명령을 내립니다. 이 함수의 인수에 프리미티

브 종류, 시작 정점의 인덱스, 프리미티브 개수를 지정해서 파이프라인에서 렌더링을 처리하도록 합니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

지금까지의 내용을 줄여서 버퍼 생성과 렌더링까지의 함수를 정리하면 다음과 같습니다.

1. 정점 버퍼 생성: pDevice->CreateVertexBuffer()
2. 정점 버퍼 디바이스에 연결: pDevice->SetStreamSource()
2. 정점 구조체 형식 알림: pDevice->SetFVF()
4. 프리미티브 정보 전달과 렌더링: pDevice->DrawPrimitive()

실습 과제) [m3d\\_01\\_overview02\\_vertex.zip](#) 예제를 이용해서 빨강색, 파란색, 노랑색, 분홍색 삼각형을 임의의 위치에 렌더링 하시오.

### 3.1.2 그래픽 파이프라인에서의 변환

[m3d\\_01\\_overview02\\_vertex.zip](#) 예제의 구조체를 다음과 같이 위치(x, y, z)를 벡터에 대한 연산이 정의되어 있는 Vector3 구조체의 변수로 만들면 정점 데이터를 다루기가 편리합니다.

```
struct VtxRHWD
{
    Vector3 p;           // 위치
    float   rhw;         // reciprocal homogeneous w
    DWORD   color;       // 색상
    ...
};
```

[m3d\\_01\\_overview03\\_vector.zip](#) 예제는 위의 구조체를 적용해서 이전과 동일하게 상하, 좌우를 키보드로 움직이는 예제입니다.

눈 여겨 볼 것은 삼각형을 움직이기 위해서 이동에 대한 방향 벡터를 만들어서 이 값을 다음 수식처럼 이동에 적용하고 있는 부분입니다.

벡터의 이동 = 현재 위치 + 이동 변위 벡터 =  $\vec{P}' = \vec{P} + \vec{T}$

이것을 의사 코드(Pseudo-code)로 구현하면

```
Vector3      현재위치;
Vector3      변위벡터;
```

현재위치 = 현재위치 + 변위벡터;

가 됩니다.

[m3d\\_01\\_overview03\\_vector.zip](#)의 INT CMain::FrameMove()의 구현 부분을 보면 다음과 같이 방향을 미리 설정한 다음 이동이 필요하면 위의 식을 그대로 적용하고 있음을 알 수 있습니다.

```
Vector3 DirectionLeft   (-2, 0,0);
Vector3 DirectionRight  ( 2, 0,0);
Vector3 DirectionUp     ( 0,-2,0);
Vector3 DirectionDown   ( 0, 2,0);
...
if( GetAsyncKeyState( VK_LEFT) & 0x8000)
{
    VtxRHWD* pvtx;
    if( FAILED( m_pVB->Lock( 0, 0, (void**)&pvtx, 0 ) ) )
        return E_FAIL;

    pvtx[0].p += DirectionLeft;
    pvtx[1].p += DirectionLeft;
    pvtx[2].p += DirectionLeft;
    pvtx[3].p += DirectionLeft;

    m_pVB->Unlock();
}
```

벡터와 행렬에 대한 클래스가 없는 분들은 D3DXVECTOR{2|3|4} 구조체와 D3DXMATRIX 구조체를 이용하십시오. 이 구조체들은 벡터, 행렬 연산에 대해서 아주 잘 만들어져 있습니다. 저도 게임을 만들 때 DXSDK에서 제공하는 구조체들을 주로 사용하며, DXSDK를 사용할 수 없는 플랫폼에서는 이 구조체들을 참고해서 비슷하게 만들어 사용하고 있습니다.

실습 과제) 여러 개의 삼각형을 작은 삼각형을 출력하는 화면 보호기 프로그램을 작성하십시오.

폴리곤(Polygon)은 원래는 다각형이나 3D의 모델은 삼각형으로 만들기 때문에 하나의 모델을 여러 개의 삼각형으로 구성할 때 이 삼각형들을 폴리곤이라 합니다.

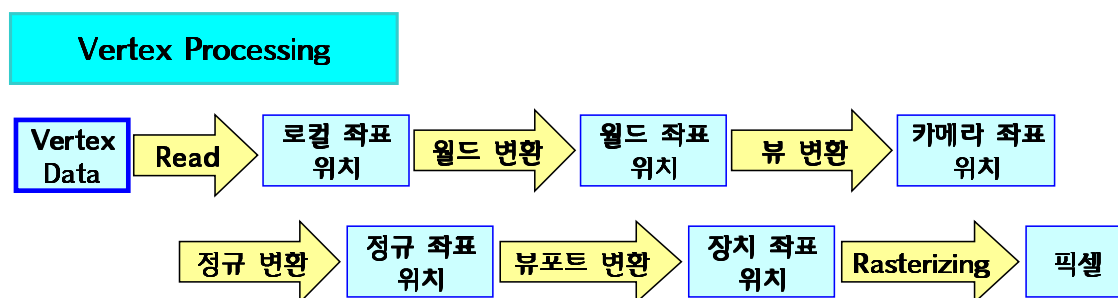
여러분이 인체를 흉내내기 위해서 삼각형이 1000개 되는 폴리곤으로 모델을 만든다고 가정합니다.

만약 이 모델을 움직이게 하려면 어떻게 하겠습니까?

지금까지 배운 지식으로는 모든 정점을 최소한 1000번 이상 루프를 돌면서 이동해야 할 것입니다. 그런데 좀 더 자세하게 묘사하기 위해서 폴리곤을 10000개를 사용하게 되면 이 폴리곤 수만큼 10000번 정도 루프를 돌아서 움직임을 그렇다면 10000개를 가진 삼각형은? 루프를 10000번 진행합니다. 폴리곤이 많아질수록 사태는 심각합니다.

이 문제는 그래픽 파이프라인에서 정점이 처리되는 방법을 알고 있으면 아무 문제도 되지 않습니다.

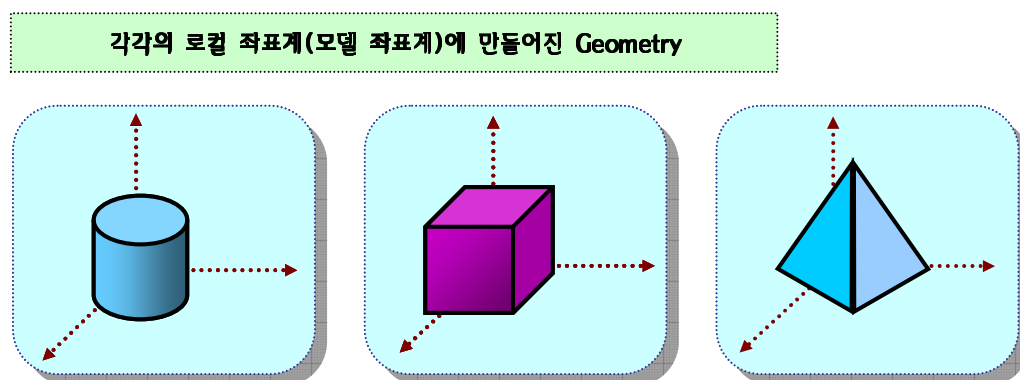
정점의 처리과정은 다음 그림과 같이 진행이 됩니다.



<래스터 처리까지의 정점 처리과정>

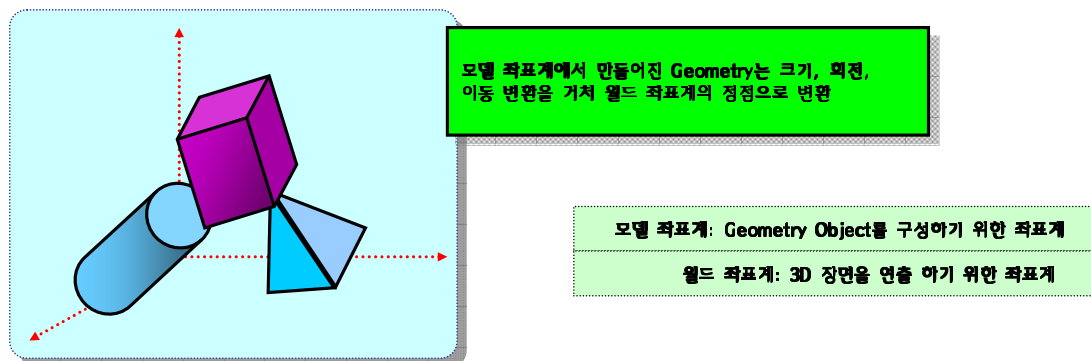
### 3.1.3 월드 변환

DrawPrimitive() 함수가 호출되는 순간 GPU는 입력 받은 정점 데이터(정점 버퍼)를 읽고, 이 정점 데이터에서 위치 값을 읽습니다. 이 위치 값을 로컬 좌표계(Local Coordinate System: 지역 좌표계)의 위치라 합니다. 여기서 다음 그림처럼 각각의 렌더링 오브젝트(Rendering Object) 또는 Geometry Object가 각각의 기준 좌표를 가지고 있는 시스템입니다.



<로컬 좌표계의 모델들- 로컬 좌표계에서는 각자의 좌표계가 존재>

이 모델들을 적절히 배치해서 장면을 연출해야 합니다. 이 때 장면을 연출하기 위한 기준 좌표계가 필요한데 이 좌표계를 월드 좌표계(World Coordinate System)이라 합니다.



<월드 좌표계(World Coordinate System - 장면을 구성하기 위한 기준 좌표계)>

그래픽 파이프라인은 로컬 좌표계의 오브젝트의 위치를 월드 좌표계의 위치로 바꿉니다. 이것을 월드 변환(World Transform)이라 합니다. 또한 월드 변환은 오브젝트의 위치에 이 오브젝트에 해당하는 행렬을 단순히 곱하는데 이 때 이 행렬을 월드 행렬이라 합니다.

월드 행렬을 곱하는 과정은 GPU에서 처리가 되므로 화면 출력에만 영향을 주고 정점 데이터에는 전혀 아무런 변화를 주지 않습니다.

그림 <래스터 처리까지의 정점 처리과정>처럼 GPU는 입력한 정점 데이터에 무조건 월드 변환 과정을 거치게 합니다. 따라서 이 월드 변환 행렬을 설정하면 앞서 제기된 폴리곤의 증가로 발생하는 루프의 반복을 피하게 됩니다.

디바이스에 월드 변환 행렬을 설정하는 방법은 다음과 같습니다.

```
// 월드 행렬
D3DXMATRIX matWorld;
...
// 디바이스의 월드 행렬을 설정한다.
m_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
```

위의 코드에서 D3DXMATRIX 구조체는 동차 좌표를 이용해서 3차원의 크기, 회전, 이동 변환을 한번에 처리하기 위해 원소가 4x4 구성된 행렬입니다.

DXSDK는 월드 행렬을 만드는 다양한 함수들이 존재합니다. 몇 가지 예를 들면 다음과 같습니다.

이동 행렬: D3DXMatrixTranslation ( )

크기 변환 행렬: D3DXMatrixScaling ( )

회전 행렬:

X축 회전: `D3DXMatrixRotationX()`

Y축 회전: `D3DXMatrixRotationY()`

Z축 회전: `D3DXMatrixRotationZ()`

사원수 회전: `D3DXMatrixRotationQuaternion()`

임의의 축 회전: `D3DXMatrixRotationAxis()`

오일러 앵글 회전: `D3DXMatrixRotationYawPitchRoll()`

이 함수들은 이전의 기초 수학에서 한번쯤 언급한 함수들입니다. 이 함수들을 이용해서 다음과 같은 경우 월드 행렬을 만드는 방법을 생각해 봅시다.

모델(렌더링 오브젝트)의 위치를 상대적으로 (tx, ty, tz)만큼 이동한다.

모델을 Y축으로  $\theta$  만큼 회전한다.

모델의 전체 크기를 S만큼 키운다.

위의 조건에 맞는 월드 행렬을 구할 때는 변환의 순서에 맞게 크기, 회전, 이동 순으로 행렬을 만들어서 이들을 곱합니다.

다음과 같이 주어진다고 합시다.

// 주어진 입력 값

`FLOAT` fScale; // 크기 변환 값

`FLOAT` fAngle; // 회전 값

`D3DXVECTOR3` vcT; // 위치 이동 값

`D3DXMATRIX` mtWld; // 최종 월드 행렬

먼저 크기 변환 행렬을 만듭니다.

// 같은 크기로 커지므로 x, y, z세 방향에 같은 크기 값(S)으로 벡터를 만든다.

`D3DXVECTOR3` vcS(fScale, fScale, fScale);

// 크기 변환 행렬을 구한다.

`D3DXMATRIX` mtSc1; // 크기 변환 행렬

`D3DXMatrixScaling`(&mtSc1, vcS.x, vcS.y, vcS.z);

먼저 다음으로 회전에 대한 행렬을 만듭니다. 주의해야 할 것은 DXSDK는 각도에 대해서 Degree가 아닌 Radian을 사용합니다. 따라서 사용자가 Degree 값을 주었을 때 Radian으로 바꾸어야 하는데 DXSDK의 `D3DXTToRadian` 매크로를 사용하면 편합니다.



// Degree 값으로 주어지면 Radian으로 바꾼다.

```
float fAngle;
```

```
float fRadian = D3DXToRadian( fAngle);
```

// Y축에 대한 회전 행렬을 구한다.

```
D3DXMATRIX mtRot;          // 회전 행렬
```

```
D3DXMatrixRotationY(&mtRot, fRadian);
```

세 번째 단계로 다음과 같이 D3DXMatrixTranslation() 함수를 이용해서 이동 행렬을 만듭니다.

// 이동 행렬을 구한다.

```
D3DXMATRIX mtTrn;          // 이동 행렬
```

```
D3DXMatrixTranslation(&mtTrn, vcT.x, vcT.y, vcT.z);
```

마지막 단계에서 이들 행렬을 크기 \* 회전 \* 이동 순으로 곱해서 월드 행렬을 만듭니다.

// 최종 월드 행렬을 구한다.

```
mtWld = mtScl * mtRot * mtTrn;
```

때로는 세 번째 단계인 이동행렬을 만들지 않고 다음과 같이 직접 월드 행렬의 \_41, \_42, \_43 값을 설정하기도 합니다. 제가 즐겨 사용하는 방법인데 이 방법은 행렬을 만들지 않고, 곱셈 연산을 줄여줍니다. 요즘은 CPU가 워낙 빨라서 이 방법을 사용해도 성능은 차이가 없습니다.

//크기와 회전만 곱함

```
mtWld = mtScl * mtRot;
```

// \_41, \_42, \_43 에 이동 값 직접 설정

```
mtWld._41 = vcT.x; mtWld._42 = vcT.y; mtWld._43 = vcT.z;
```

이렇게 월드 행렬을 만드는 방법을 살펴 보았습니다. 때로는 입자 효과(Particle Effect)처럼 그래픽 파이프라인의 변환을 이용하지 못하고 앞에서 제시한 문제처럼 직접 정점의 위치를 전부 이동해야 할 경우도 있습니다.

이 경우도 월드 행렬이 필요합니다. 위의 월드 행렬을 구해서 입자의 정점에 위치는 D3DXVec3TransformCoord() 함수에 법선은 D3DXVec3TransformNormal() 함수를 적용해서 입자의 정점을 직접 변환합니다. 이렇게 그래픽 파이프라인을 이용하지 않고 직접 정점의 위치를 변환하는 과정은 하드웨어 가속을 받지 못하고 정점의 수만큼 변환에 대한 연산 소요된다는 것을 다시 한

변 강조 합니다.

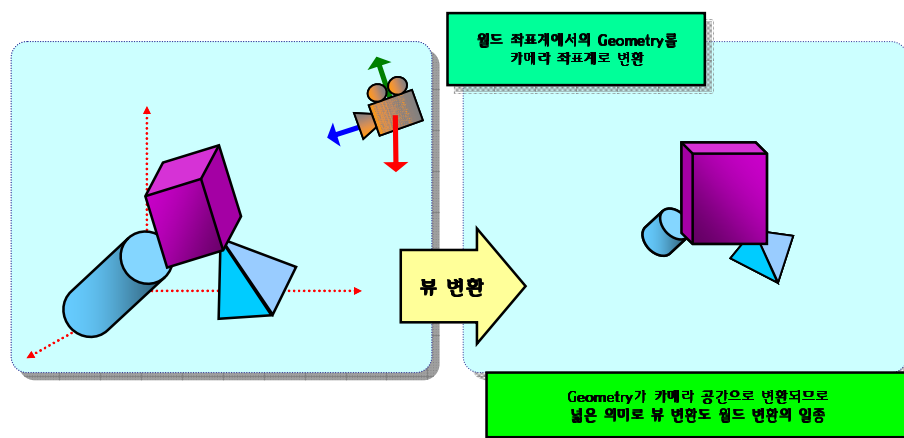
정리하면 렌더링 상태 머신 값(파이프라인의 월드 행렬 값)을 설정해서 장면을 연출하는 것은 하나의 모델을 가지고 월드 행렬만 바꾸어서 장치에 여러 번 출력할 수 있으며 여러 모델을 출력할 때도 그에 해당 하는 월드 행렬을 만드는 정도만 시간 비용이 들기 때문에 전체 렌더링 속도에 이득이 있습니다. 이 때 주의해야 할 것은 모델의 정점들은 절대 변하지 않고 그대로 유지되며 월드 행렬을 만들 때 크기, 회전, 위치 순으로 행렬을 곱해야 하는 것입니다.

### 3.1.4 뷰 변환(View, Viewing Transform)

그래픽 파이프라인에서 정점의 데이터의 위치를 입력 받아 월드 변환을 거친 후에 디바이스는 뷰 변환을 진행합니다. 뷰 변환은 정점을 카메라 공간 좌표계로 변환하는 카메라 뷰에 의한 변환입니다. 이것은 월드 변환과 내용이 같습니다. 단지 월드 공간 대신 카메라 공간이 있을 뿐입니다. 만약 카메라가 움직이지 않고 그래픽 파이프라인의 초기 값(이 때 뷰 행렬은 단위 행렬입니다.)으로 그대로 유지 하고 있다면 뷰 변환은 필요 없게 됩니다.

이러한 이유들로 OpenGL은 월드 변환이 없고, 월드 변환과 뷰 변환을 합친 월드-뷰 변환이 있습니다. 역지로 OpenGL의 월드-뷰 변환 행렬을 D3D로 표현한다면 D3D의 월드 행렬 \* D3D의 뷰 행렬이 될 것입니다.

다음 그림은 파이프라인에서 처리된 월드 공간의 좌표를 카메라 공간으로 변환 하는 뷰 변환 그림입니다.



<뷰 변환 - D3D는 월드 변환한 좌표들을 뷰 행렬을 적용해 뷰 변환을 진행한다>

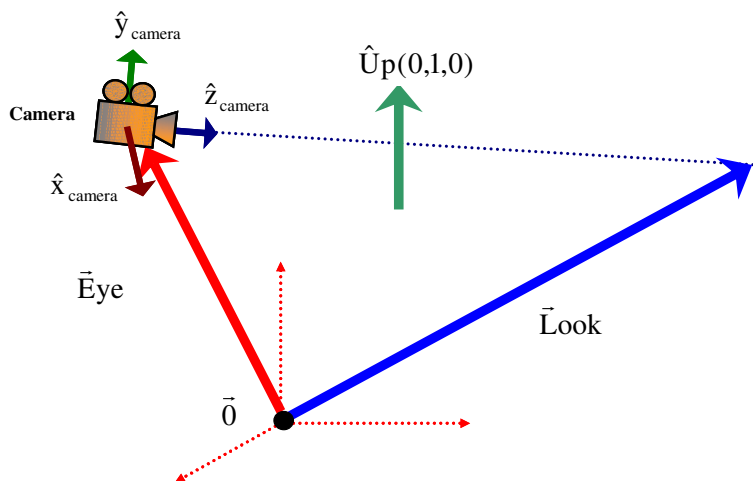
DXSDK는 뷰 변환에 대한 뷰 행렬을 만드는 D3DXMatrixLookAtLH() 함수를 제공합니다.

이 함수는 왼손 좌표계를 사용하는 D3D에서 뷰 변환 행렬을 만들 때 자주 사용이 되며 함수의 원형은 다음과 같습니다.

D3DXMATRIX\* WINAPI D3DXMatrixLookAtLH

```
( D3DXMATRIX *pOut, CONST D3DXVECTOR3 *pEye, CONST D3DXVECTOR3 *pAt,
  CONST D3DXVECTOR3 *pUp );
```

함수의 첫 번째 인수는 이 함수를 이용해서 구한 행렬의 주소입니다. 그 다음에 있는 Eye 벡터, Look 벡터, Up 값은 다음 그림처럼 Eye는 월드 좌표계에서 카메라의 위치를, Look은 월드 좌표계에서 카메라가 보고 있는 지점, 월드 좌표계에서 Up은 최초로 주어지는 카메라 Y축 값입니다.



< D3DXMatrixLookAtLH ( ) 함수 인수 의미>

주의할 것은 시선 벡터는 위 그림에서 카메라의 z축 벡터가 됩니다. 종종 Look를 카메라가 보고 있는 방향 벡터인 시선 벡터로 착각하고 프로그램을 만들어서 화면에 원하는 장면이 안 나오는 경우가 있습니다. 다시 한 번 강조하지만 Look 벡터는 카메라가 보고 있는 방향이 아니라 지점 벡터입니다.

레이싱(Racing), 비행 시뮬레이션 게임들은 카메라와 주인공을 일치시키는 경우가 많이 있고 온라인 RPG나 FPS 게임들은 Up 벡터를 (0,1,0) Y축과 일치 시키는 경우가 많이 있습니다. 이것은 Up 벡터가 움직이면 카메라도 기울어져서 어지러움을 느낄 때가 많아 게임 유저의 장시간 게임 하에 많은 피로를 주어서 카메라가 덜 흔들리도록 (0,1,0)으로 많이 설정합니다. - 물론 장시간 게임 자체가 문제겠지요?-

D3DXMatrixLookAtLH( ) 함수 내부 처리 방법은 DXSDK 도움말에 자세하게 나와 있어서 직접 함수와 동일한 함수를 구현할 수 있습니다. 이 부분 3D 기초 후반부 카메라에서 좀 더 자세하게 다루겠습니다.

D3DXMatrixLookAtLH( ) 함수로 구한 행렬을 그래픽 파이프라인의 렌더링 상태 머신 뷰 행렬 값 설정은 월드 행렬과 마찬가지로 디바이스의 SetTransform( ) 함수를 다음과 같이 이용합니다.

```

// 뷰 행렬
D3DXMATRIX m_mtView;
D3DXVECTOR3 vEyePt( 0.0f, 3.0f, -5.0f );           // 카메라의 위치
D3DXVECTOR3 vLookAt( 0.0f, 0.0f, 0.0f );           // 카메라가 보고 있는 지점
D3DXVECTOR3 vUpVec( 0.0f, 1.0f, 0.0f );            // 카메라의 업 벡터
D3DXMatrixLookAtLH(&m_mtView, &vEyePt, &vLookAt, &vUpVec ); // 뷰 행렬 만들기

m_pd3dDevice->SetTransform( D3DTS_VIEW, &m_mtView ); // 디바이스의 뷰 행렬 설정

```

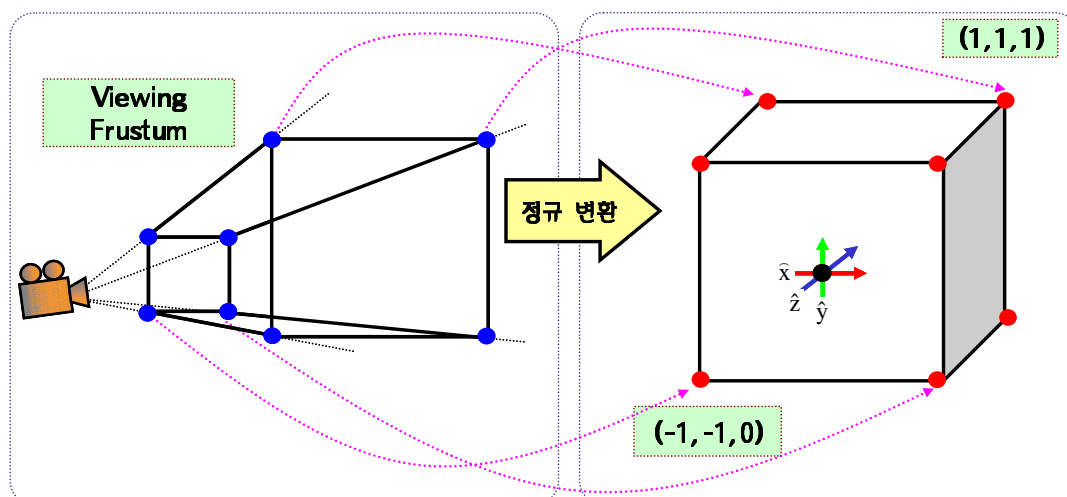
### 3.1.5 정규 변환(투영 변환: Projection Transform)

정규 변환 혹은 투영 변환은 카메라 좌표계의 Geometry를 장치(모니터)에 독립인 투영 평면으로 변환하는 것입니다. 정규 변환이라는 이름은 어떤 장치에도 독립성을 유지하기 위해 뷰 체적 안의 정점 위치는 변환 후 정규화된 값으로 변환되는 데서 나온 이름입니다.

이 정규화 값은 벡터의 정규화와 비슷해서 뷰 체적 안의 정점이 변환을 거치면 D3D의 경우 x 값은  $[-1, 1]$  y 값은  $[-1, 1]$ , z 값은  $[0, 1]$ 의 범위 값으로 결정이 됩니다. 참고로 OpenGL의 경우 z 값은  $[-1, 1]$  내에 있습니다.

또한 3차원 좌표를 2차원 화면으로 바꾸는 투영(Projection)의 역할로 인해 투영 변환이라고 합니다. 자주 사용되는 용어는 투영 변환입니다.

다음은 D3D 그래픽 파이프라인에서 뷰 체적 안의 점들이 정규 변환을 표현한 그림입니다.



< Direct3D의 뷰 체적(View Volume)과 정규변환>

3차원 공간의 점을 2차원 평면으로 투영하는 방법은 크게 투시 투영(Perspective Projection)과

평행 투영(Parallel Projection)으로 분류합니다. 투시(원근) 투영은 투영 중심점(COP: Center of Projection) 또는 소실점(Vanish Point) 수에 따라 단일점(1-Point) 투영, 이중점(2-Point) 투영, 삼중점(3-Point) 투영이 등이 있습니다.

평행 투영은 정사(직교: Orthographic) 투영, 등축(axonometric) 투영, 경사(Oblique) 투영이 있고, 직각 투영은 등각(동형: Isometric) 투영, 이중형(diametric), 삼중형(trimetric)으로 분류합니다. 경사 투영은 캐벌리어(Cavalier) 투영, 캐비닛(Cabinet) 투영 등이 있다고 합니다.

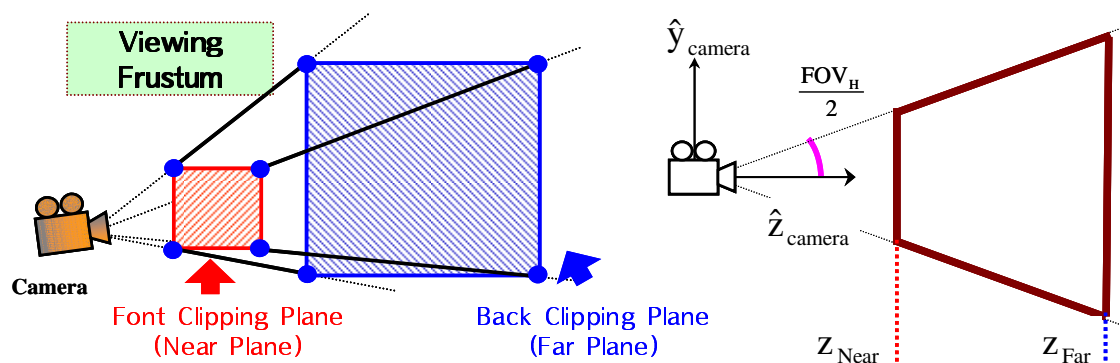
이 중에서 단일점 투시 투영은 게임에서 게임 프로그램에서 가장 많이 사용되며 줄여서 투시 투영(또는 원근 투영)으로 많이 부릅니다. 가끔 건축, 기계 설계에서 이용되는 CAD와 같은 효과를 내기 위해 직교 투영을 투영을 사용하기도 하고, Isometric은 2D 전략시물레이션이나 RPG 게임에서 광활한 지형을 표현할 때 많이 이용되기도 합니다.

DXSDK는 투영 행렬을 구해주는 많은 함수들이 있습니다. 이 중에서 D3D를 이용한 게임 프로그램에서는 단일점 투시 투영을 반영한 D3DXMatrixPerspectiveFovLH() 함수를 가장 많이 이용합니다. 이 함수의 원형은 다음과 같습니다.

**D3DXMATRIX\* WINAPI D3DXMatrixPerspectiveFovLH**

( **D3DXMATRIX** \*pOut, **FLOAT** fovy, **FLOAT** Aspect, **FLOAT** zn, **FLOAT** zf );

함수의 첫 번째 인수는 이 함수로 구한 투영 행렬의 주소입니다. 그 다음에 FOV(Field of View)는 광 시야 각 입니다. Aspect는 영상 비율(Aspect Ratio) 값으로 한 주사선 당 픽셀을 전체 주사선 수로 나눈 값으로 이 값은 화면의 가로 폭을 화면의 높이 값으로 나눈 것과 동일합니다. 다음의 zn은 카메라에서부터 Near 평면까지의 거리 값이고, zf는 카메라에서부터 Far 평면까지의 거리 입니다.



<뷰 체적 만들기>

게임 프로그램에서 FOV 값은  $\pi/4$  (45도) 값을 가장 많이 사용합니다. FOV가 크면 어안 렌즈(물고기 눈)처럼 화면에 많은 장면을 담게 되어 어지럽고, FOV가 작으면 어지러움은 덜 느끼나 화면에

서 오브젝트가 적게 표현되어 게임 유저들이 답답함을 느낍니다. 보통  $\pi/4$  기준으로 FOV를 크게 하거나 줄입니다.

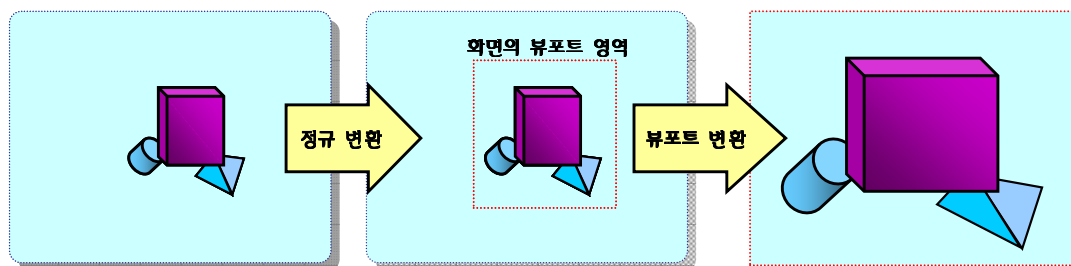
Near 평면까지의 거리는 1.0f로 설정하는 것이 좋습니다. Near값이 1이면 마우스로 렌더링 오브젝트를 선택하는 Picking을 구현할 때 유리한 점이 있습니다.

Far 평면까지의 거리에 대한 값은 테스트 코드를 만들 때 [4000, 20000] 범위 값으로 설정합니다. 이 값이 너무 작으면 렌더링이 뷰 체적을 벗어나는 경우가 많아 화면에 아무것도 출력 못하는 일이 발생합니다. 예러도 없는 것 같은데 화면에 아무것도 나오지 않는다면 이 값을 한 번쯤은 의심해 볼 필요가 있습니다.

### 3.1.6 뷰포트 변환(Viewport Transform)

정규 변환을 거친 Geometry 데이터는 뷰포트 변환을 거쳐 화면에 출력합니다. 직접적으로 화면이라는 장치와 연관되어 있어서 이 변환은 장치 의존 변환이 됩니다.

변환의 과정은 다음 그림 처럼 해당 윈도우의 클리핑 영역에 맞게 변환을 합니다.



<그림 뷰포트 변환>

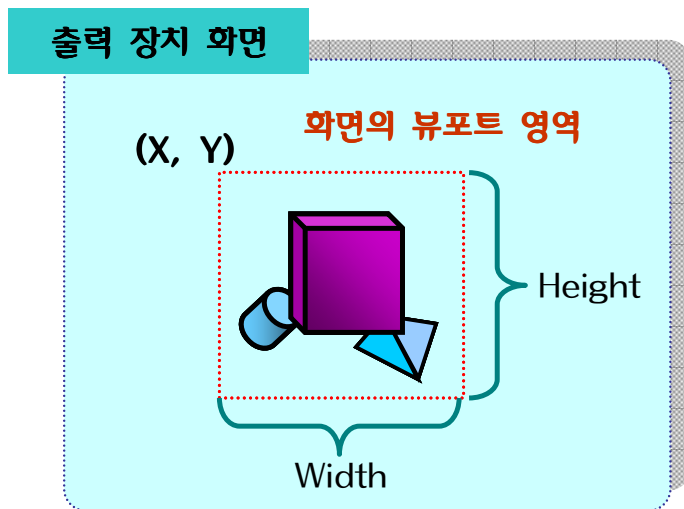
Direct3D는 뷰포트를 설정하지 않으면 디바이스를 생성할 때 설정했던 화면의 너비와 높이를 뷰포트 영역으로 설정 합니다. 만약 직접 뷰포트를 설정하려면 `IDirect3DDevice9::SetVeiwPort()` 함수를 호출해서 뷰포트를 설정합니다. 게임 제작 툴(Tool) 프로그램이 아니면 뷰포트를 설정할 일은 별로 없습니다.

가끔씩 화면에 오브젝트의 위치에 문자를 출력할 때 그래픽 파이프라인의 변환을 CPU에서 직접 연산할 일이 있습니다. 이 경우를 대비해서 뷰포트 행렬을 얻는 방법 또한 알고 있어야 합니다.

DXSDK에는 직접 뷰포트 행렬을 얻는 방법은 없습니다. 이는 프로그래머가 직접 만들어야 하는데 SDK 도움말에 뷰포트 행렬에 대해서 설명이 잘 나와 있습니다. 도움말을 이용해 뷰포트 행렬을 만들면 다음과 같습니다.

$$\text{뷰포트 행렬: } \vec{M}_{\text{viewport}} = \begin{bmatrix} \frac{W}{2} & 0 & 0 & 0 \\ 0 & -\frac{H}{2} & 0 & 0 \\ 0 & 0 & Z_{\max} - Z_{\min} & 0 \\ X + \frac{W}{2} & Y + \frac{H}{2} & Z_{\min} & 1 \end{bmatrix}$$

이 행렬에서 사용되는 변수는 그림처럼 모니터와 같은 출력 장치의 영역 값으로 설정합니다.



보통 게임에서 적용되는 범위를 본다면 X는 0 ~ Screen Width], Y는 0 ~ Screen Height 범위를 가집니다.  $Z_{\max}$ 와  $Z_{\min}$ 은 정규 변환의 Z 값의 범위 0 ~ 1로 설정합니다. 이것은 Z에 대한 Clipping이라 할 수 있습니다. 또한 항상  $Z_{\max} > Z_{\min}$ 으로 설정합니다.

뷰포트 행렬 수식에 W=Screen Width, H= Screen Height, Zmin=0, Zmax = 1을 넣고 정규변환 값[-1, 1, 0, 1], [ 1, -1, 0, 1], [-1, -1, 0, 1], [ 1, -1, 0, 1]을 벡터 \* 행렬 순으로 곱해보면 화면 영역으로 계산 결과가 나오게 됨을 볼 수 있습니다.

다음은 앞의 뷰포트 행렬 수식을 구하는 함수 입니다.

```
void D3DXMatrixViewport(D3DXMATRIX* pOut, const D3DVIEWPORT9* pV /*Viewport*/)
{
    float fW = 0; float fH = 0; float fD = 0; float fY = 0;
    float fX = 0; float fM = FLOAT(pV->MinZ);

    fW = FLOAT(pV->Width)*.5f;    fH = FLOAT(pV->Height)*.5f;
    fD = FLOAT(pV->MaxZ) - FLOAT(pV->MinZ);
```

```

fX = FLOAT(pV->X) + fW;    fY = FLOAT(pV->Y) + fH;

*pOut = D3DMATRIX( fW,  0.f,  0, 0,
                  0.f, -fH,  0, 0,
                  0.f,  0.f, fD, 0,
                  fX,   fY, fM, 1);
}

```

이 함수는 디바이스를 통해서 뷰포트를 얻어와야 합니다. 디바이스의 Get 함수들을 호출할 수 있으려면 여러분은 디바이스를 생성할 때 Behavior가 PURE가 아닌 Flag를 선택해야 Get 함수를 호출할 수 있습니다. MIXED나 SOFTWARE VERTEX PROCESSING이 가장 무난 합니다.

다음 코드는 디바이스에서 뷰포트 값을 얻고 이것을 위에서 만든 함수로 뷰포트 행렬을 구하는 예입니다.

```

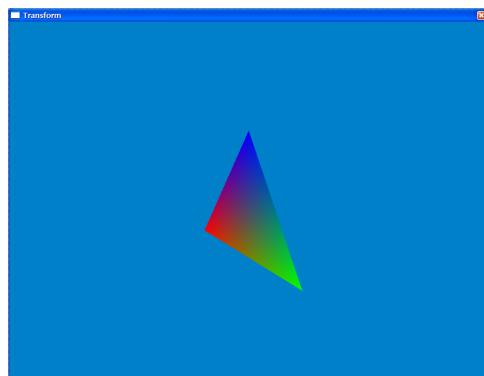
// 뷰포트 행렬 설정
D3DMATRIX      mtVp;
D3DVIEWPORT9   vp;

// Behavior of Device is not PURE
pDevice ->GetViewport(&vp);
D3DXMatrixViewport(&mtVp, &vp);

```

이제 변환에 대한 기초적인 내용은 끝났습니다. 지금까지 변환에 대한 설명에서 중간 중간에 보인 코드들은 [m3d\\_01\\_overview04\\_transform.zip](#) 을 참고 하기 바랍니다.

이 파일을 풀어 실행하면 다음 그림처럼 삼각형이 회전을 할 것입니다.

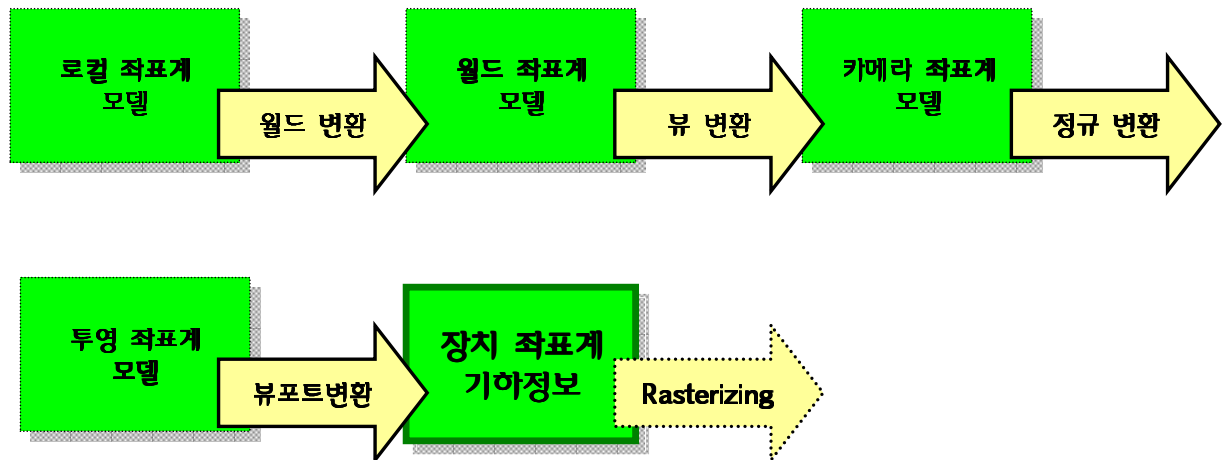


<렌더링 오브젝트 변환 예>

그래픽 파이프라인의 변환을 잘 알고 있으면 이후 셰이더 코드를 작성할 때도 상당히 유리합니다.



게임 프로그램을 배우는 여러분들은 최소한 다음 그림 정도는 꼭 기억하기 바랍니다.



< Graphic Pipe Line의 Geometry 변환>

D3D에서 변환에서 잊지 말아야 할 내용을 정리하면 다음과 같습니다.

1. D3D 디바이스의 멤버 함수 중에서 변환 행렬 적용 함수는 SetTransform()입니다.
2. 월드 행렬은 크기 \* 회전 \* 이동 행렬의 곱으로 만듭니다.
3. 월드 변환을 적용하지 않고 직접 정점의 위치를 바꿀 때에는 D3DXVec3TransformCoord() 함수를, 법선 벡터를 바꿀 때는 D3DXVec3TransformNormal() 함수를 사용합니다.
4. 뷰 행렬을 만드는 함수는 D3DXMatrixLookAtLH() 입니다.
5. 투영 행렬을 만드는 함수는 D3DXMatrixPerspectiveFovLH() 입니다.

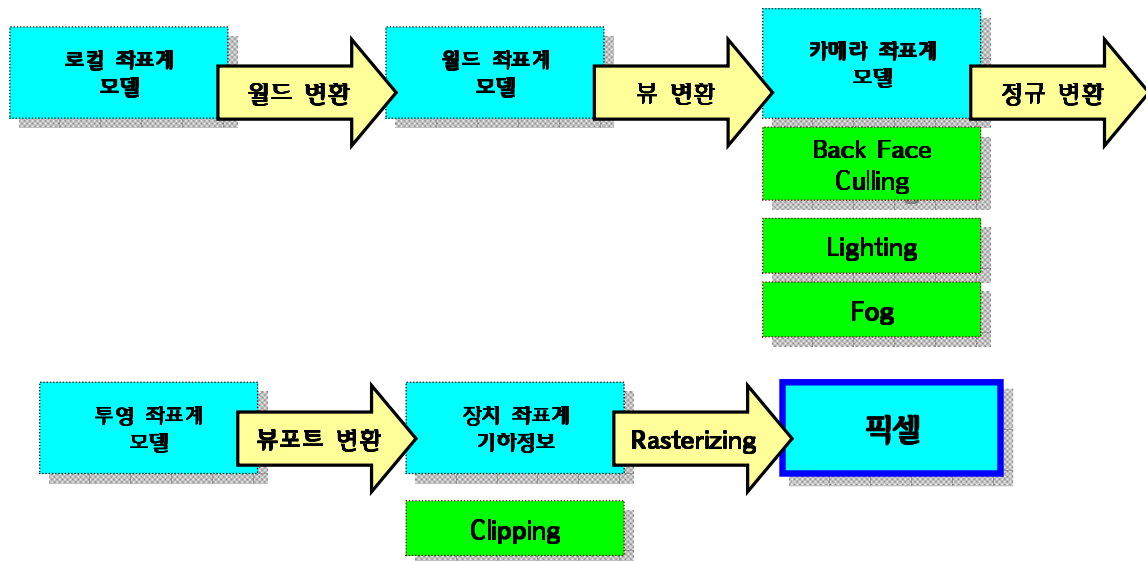
실습과제) 앞의 과제 중에서 삼각형을 이용한 화면 보호기 프로그램에서 삼각형의 이동을 행렬을 통해서 구현해 보시오.

실습과제) 여러 삼각형들을 적절히 조합해 비행기를 만들어 보시오. 이 비행기를 월드 행렬 변환을 적용시켜 이동을 구현하시오.

## 3.2 렌더링 상태 머신(Rendering State Machine) 설정

렌더링 상태 머신(기계)은 디바이스를 좀 더 일반화 시킨 추상적인 개념입니다. D3D가 일반적인 그래픽파이브라인 구조를 따르지만 이 것이 언제 바뀔지 모릅니다. 또한 복잡한 작업을 거쳐야만 일이 된다면 프로그래머로서 굉장히 피곤한 일이 아닐 수 없습니다. D3D는 이러한 부분을 상태 머신으로 두고 사용자는 단순히 상태 값 설정만으로 원하는 장면을 만들어 낼 수 있게 했습니다.

다음은 그래픽 파이프라인의 정점 처리과정에서 같이 처리되는 병행 작업을 나타낸 그림입니다.



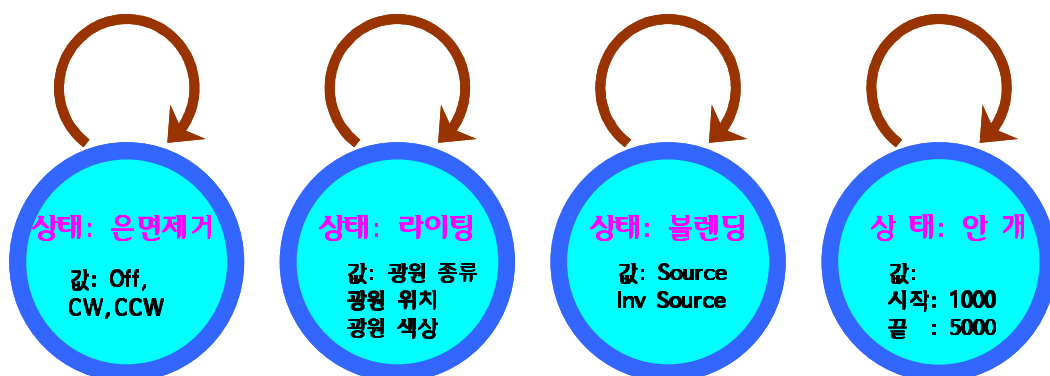
<그래픽 파이프라인의 정점 처리 병행 작업>

위의 그림에서 녹색 부분이 병행 작업이며 이 부분은 정점의 변환 과정처럼 기본 값(Default)를 가지고 있고 무조건 처리합니다.

상태 머신의 설정은 다음과 같이 합니다.

1. 상태 머신 활성화
2. 상태 머신 값 전달

상태가 한번 활성화 되면 사용자가 비활성화로 설정할 때까지 디바이스는 계속 상태를 유지합니다. 상태 머신 값 또한 사용자가 새로운 상태 값을 설정하기 전까지 이전 값들을 계속 유지 합니다.



<D3D의 다양한 상태들>

D3D는 상태 설정을 디바이스의 SetRenderState() 함수 호출로 합니다.

```
pDevice->SetRenderState( D3DRS_상태_이름, 상태_값);
```

은면 제거의 예를 들면 다음과 같습니다.

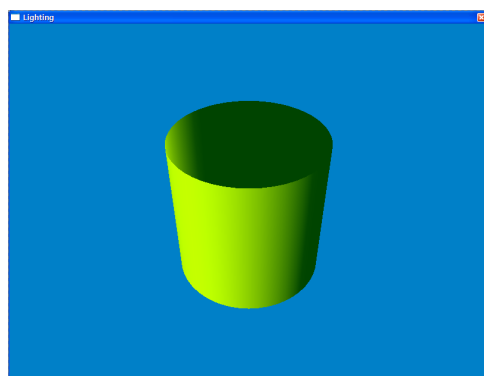
```
// 은면제거 비활성화
// 양면을 다 그릴 경우에 컬링을 끈다.
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
...
// 렌더링
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2*50-2 ); // 렌더링
...
// 은면제거 활성화
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW);
```

상태의 변경을 하고 렌더링을 마쳤으면 다시 원래의 상태로 돌려 놓아야 다른 프로그램에 영향을 주지 않습니다. 상태의 기본 값들은 게임 프로그램을 시작하기 전에 미리 약속을 하거나 아니면 D3D 기본 값을 저장해서 사용하는 것이 바람직합니다.

만약 현재 설정되어 있는 상태 값을 알고자 할 때는 GetRenderState() 함수를 호출하면 됩니다.

```
DWORD dMode=0;
pDevice->GetRenderState(D3DRS_상태이름, &dCullMode);
```

[m3d\\_01\\_overview05\\_lighting.zip](#)의 파일을 열어서 컴파일 한 다음 실행해보면 다음 그림처럼 원통이 회전하고 있는 장면을 볼 수 있습니다.



<라이팅과 상태 머신>

CMain::FrameMove() 함수 중간을 보면 렌더링 오브젝트에 조명 효과를 적용하는 다음과 같은 코드를 볼 수 있습니다.

```

D3DMATERIAL9 mtrl={0};
...
m_pd3dDevice->SetMaterial( &mtrl );

D3DXVECTOR3 vecDir;
D3DLIGHT9 light={0};
memset(&light, 0, sizeof light);
...
m_pd3dDevice->SetLight( 0, &light );
m_pd3dDevice->LightEnable( 0, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00004444 );

```

조명 효과를 적용하기 위해서 재질과, 조명이 필요합니다. D3D는 조명을 8개까지 그래픽 파이프라인에 연결할 수 있습니다. 상태 값을 설정하는 것이 길어지면 위의 코드의 SetLight(), LightEnable() 함수처럼 직접 호출해서 설정하는 경우도 있습니다.

이렇게 조명에 대한 상태 값들을 설정한 후에 파이프라인의 조명을 활성화하기 위해 CMain::Render() 함수에서 DrawPrimitive() 전에 SetRenderState( D3DRS\_LIGHTING, TRUE ) 가 호출되고 있음을 볼 수 있습니다.

```

void CMain::Render()
...
    // 조명 활성화
    m_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    ...
    m_pd3dDevice->DrawPrimitive();

```

조명 효과를 적용하는 방법은 이 후의 조명 효과에서 자세히 다루겠습니다.

## 3.3 픽셀 처리 과정과 렌더링 상태 머신

### 3.3.1 래스터 과정(Rasterization)

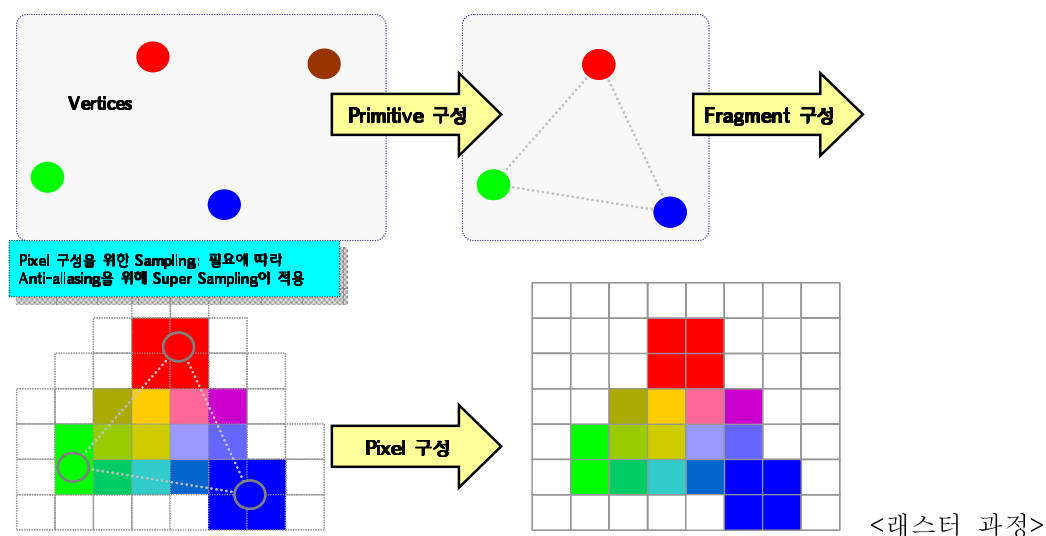
정점의 변환이 끝나면 정점 처리의 마지막 단계인 래스터 처리 과정이 남아있습니다.

래스터 과정은 다음 그림처럼 변환을 거친 기하 정보를 가지고 픽셀을 만드는 과정입니다. 여러분이 정점에 설정한 색상 값과 그래픽 파이프라인이 적용한 안개 효과, 조명 효과 등을 적절히 배합

(Blending)해서 최종 색상을 만듭니다. 이렇게 만들어진 색상들을 가지고 후면 버퍼의 색상 버퍼를 갱신하기 해서 정점과 정점 사이의 픽셀의 값들을 계산해서 만들어 냅니다.

이 픽셀 값을 계산하는 방법을 셰이딩(Shading) 모델이라 합니다. D3D의 셰이딩 모델은 플랫(Flat), 구로(Gouraud), 폰(Phone) 세가지 모델을 배경으로 구성되어 있습니다. 셰이딩 모델은 이후 조명에서 자세하게 다루겠습니다. 여기서 모델이라는 것은 3D가 실제 세계의 흉내내기(Simulation)에서 적당한 방법을 말합니다.

래스터 과정에서는 픽셀을 만드는 일은 GPU 또는 CPU에서 가장 부담 되는 작업입니다. 따라서 화면 영역에 있지 않은 픽셀들은 계산할 필요가 없습니다. 이것을 클리핑(Clipping)이라 합니다. 클리핑 알고리즘은 상당히 많이 연구되어 있는데 컴퓨터 그래픽스 책의 내용 중에서 Cohen - Sutherland 알고리즘, Liang - Barsky 알고리즘 등을 찾아 보기 바랍니다.



현재 대부분의 그래픽 카드들은 픽셀을 만드는 작업뿐만 아니라 Anti-Aliasing까지 구현하고 있습니다. 앞서 설명한 슈퍼 샘플링, 멀티 샘플링이 Anti-Aliasing 입니다.

Anti-Aliasing을 위해 D3D는 `3DPRESENT_PARAMETERS` 구조체 변수의 "MultiSampleType" 값을 하드웨어에서 지원하는 멀티 샘플 타입 값을 저장한 후에 디바이스를 만듭니다.

그리고 나서 렌더링에서 다음과 같이 렌더링 상태 설정 함수를 이용해 Anti-Aliasing을 활성화 시킵니다.

```
// 적절한 멀티 샘플링 값을 찾는다.
pD3D->CheckDeviceMultiSampleType();
...
3DPRESENT_PARAMETERS m_d3dpp;
...
```

```
d3dpp.MultiSampleType =SamplingType;
```

```
// 디바이스 생성
```

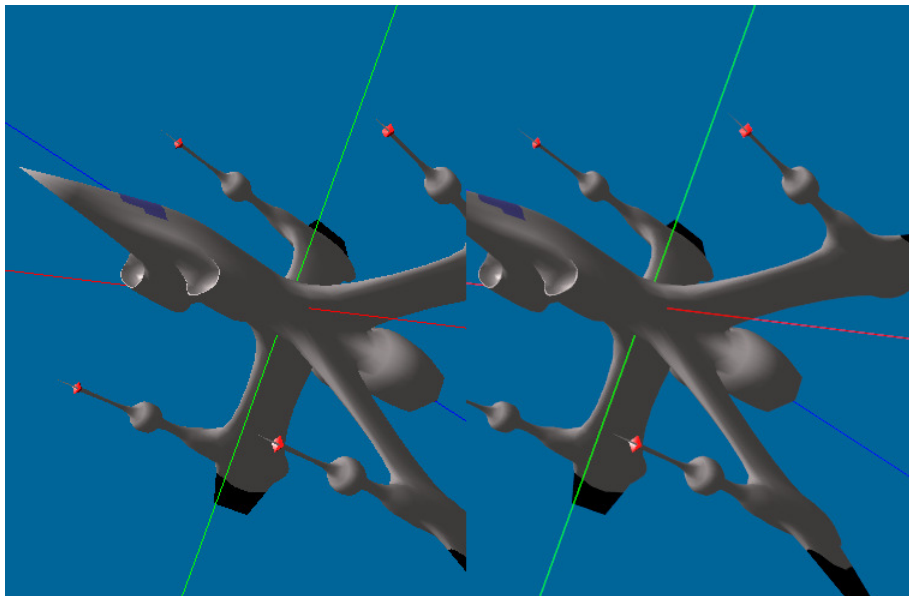
```
pD3D->CreateDevice();
```

```
// Anti - Aliasing 활성화
```

```
pDevice->SetRenderState (D3DRS_MULTISAMPLEANTIALIAS, FALSE);
```

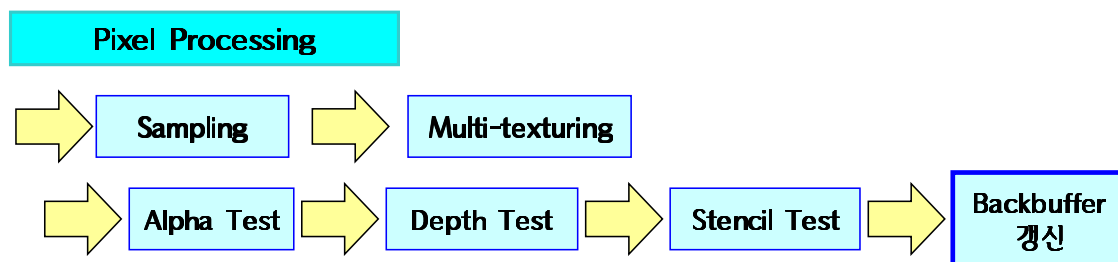
```
pDevice->SetRenderState (D3DRS_ANTI_ALIASABLE, FALSE);
```

전체 코드에 대한 내용은 [m3d\\_08\\_aux3\\_antialias.zip](#)을 참고 하기 바랍니다.



<Aliasing, Anti-Aliasing 비교>

### 3.3.2 픽셀 처리과정



<픽셀 처리 과정 개요>

D3D의 그래픽 파이프라인은 크게 정점 처리 과정과 픽셀 처리 과정 두 부분으로 구성되어 있다고 했습니다. 위의 그림은 픽셀 처리 과정을 간략하게 표현한 것인데 꼭 기억해 두기 바랍니다.

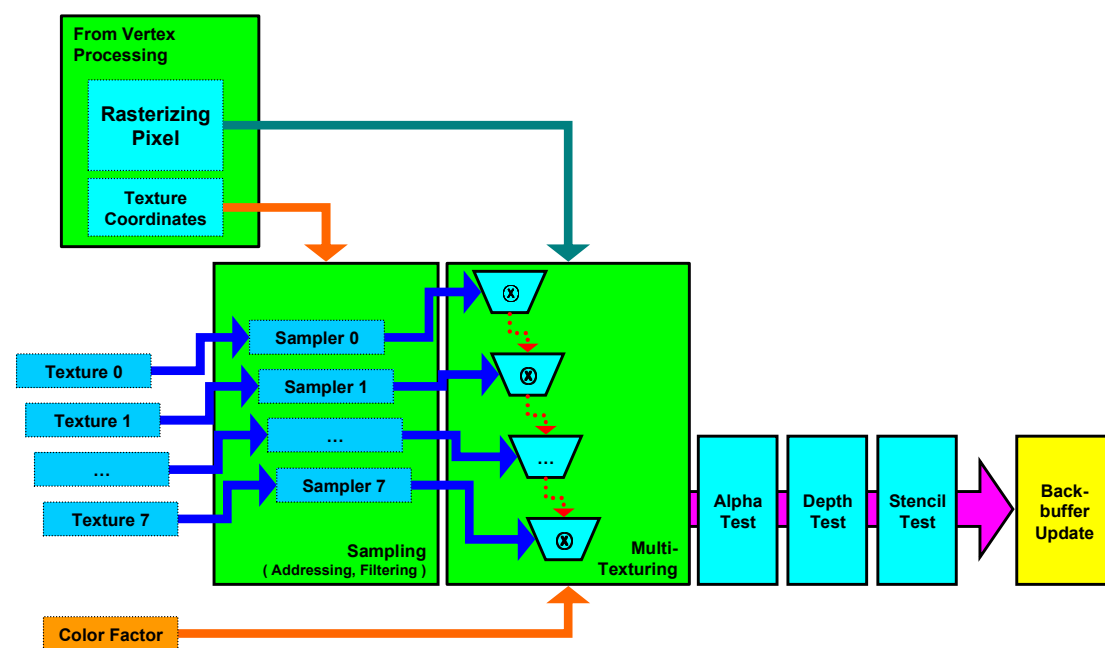
픽셀 처리과정은 단어 그대로 픽셀을 처리해서 모니터에 출력할 수 있도록 후면 버퍼의 내용을 갱신하기 위한 작업입니다.

이 과정을 좀 더 세밀하게 표현하면 다음 그림과 같습니다.

픽셀 처리과정에서 입력 받는 값은 정점 처리 과정의 래스터과정에서 만든 픽셀들입니다. 때로는 렌더링 객체를 좀 더 원하는 모양으로 표현하기 위해 텍스처를 매핑 하기도 합니다. 매핑을 위한 텍스처 좌표(UV 좌표) 값은 사용자가 지정한 텍스처에서 색상을 추출(Sampling)하는 좌표로 사용합니다.

이런 경우에 당연히 텍스처도 같이 연결해야 합니다. D3D는 총 8개의 텍스처를 파이프라인에 연결할 수 있습니다.

정해진 순서에 따라서 래스터 과정에서 만든 픽셀, 텍스처에서 샘플링한 픽셀, 그리고 사용자가 지정한 Color Factor 값을 적절히 섞는 블렌딩 작업인 Multi-Texturing 작업을 수행 합니다.



#### <픽셀 처리과정>

Multi-Texturing 작업을 끝내면 비로소 완전한 색상이 만들어지며 이후 알파, 깊이, 스텐실 테스트를 통과하면 후면버퍼의 픽셀 값을 변경하게 됩니다.

픽셀을 추출하는 샘플링, 멀티 텍스처링은 이 후에 자세하게 설명되어 있습니다. 여기서는 "아 이런 것이구나" 하는 정도로만 이해 하기 바랍니다.

[m3d\\_01\\_overview06\\_texture.zip](#) 예제를 컴파일 해서 실행하면 다음 그림과 같이 출력됩니다.

여러분이 이 예제에서 눈 여겨 봐야 할 것은 D3D가 텍스처에서 픽셀을 샘플링할 수 있도록 정점 구조체의 모양을 다음과 같이 변경해야 하는 것입니다.

```
struct VtxUV                                // 정점에 텍스처 매핑을 위해 UV좌표를 구성
{
    D3DXVECTOR3    p;                        // 정점 위치
    FLOAT          u, v;                    // 텍스처 좌표 (매핑 좌표)
    enum { FVF = (D3DFVF_XYZ|D3DFVF_TEX1), }; // TEX1의 1은 텍스처 좌표 1개 의미.
};
```

텍스처는 2D에서와 마찬가지로 LPDIRECT3DTEXTURE9 변수에 텍스처 데이터 포인터를 저장합니다.

```
LPDIRECT3DTEXTURE9 m_pTx;                // 텍스처 데이터에 대한 주소
```

텍스처 생성은 다음 코드는 D3DXCreateTextureFromFile() 함수를 사용하지만 여러분은 이 함수 보다 D3DXCreateTextureFromFileEx()를 사용하십시오.

```
// 텍스처 생성
```

```
D3DXCreateTextureFromFile( m_pd3dDevice, "Texture/Env_Sky.jpg", &m_pTx );
```

렌더링에서 텍스처 포인터를 설정합니다.

```
m_pd3dDevice->SetTexture(0, m_pTx);      // 디바이스에게 m_pTx 텍스처를 사용하겠다고 알림.
```

```
...
```

```
m_pd3dDevice->SetFVF(VtxUV::FVF );      // 정점 포맷을 설정
```

```
m_pd3dDevice->DrawPrimitive(...);        // 렌더링
```

```
m_pd3dDevice->SetTexture(0, NULL);       // 사용이 끝나면 상태 머신의 텍스처 포인터를 해제
```



<텍스처 출력 예>

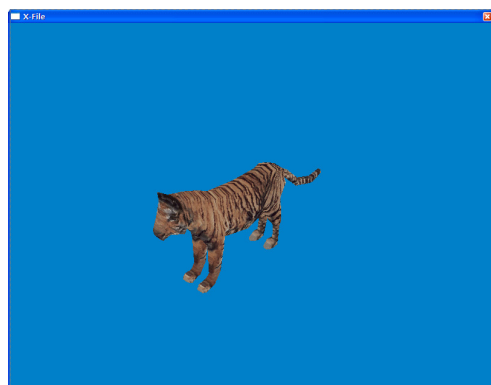


렌더링 후에 디바이스의 텍스처 포인터를 해제하는 것을 볼 수 있습니다. 이 것은 그래픽 카드의 성능과 구조가 천차만별이기 때문에 정확하게 자신이 설정한 것은 자신이 원래대로 되돌려 놓아야만 문제의 발생을 줄일 수 있다는데 있습니다. 특히 텍스처의 경우 가장 많은 영향을 주는 요소입니다. 꼭 위와 같이 렌더링이 끝나고 나서 NULL로 설정하기 바랍니다.

주요 함수:

1. 텍스처 생성 - D3DXCreateTextureFromFileEx()
2. 렌더링 머신에 텍스처 포인터 연결: pDevice->SetTexture(stage, pointer)
3. 텍스처 포인터 연결 해제: pDevice->SetTexture( stage, NULL)

D3D의 tutorial 예제 중에는 Microsoft가 제안한 X-File이라는 Geometry 모델 파일을 파일에서 읽어와 화면에 출력하는 예제가 있습니다. 많은 회사들이 자체적으로 만든 모델 파일을 이용하지만 처음 3D 프로그램을 하는 사람에게는 X-File이 쉽습니다. 이 [m3d\\_01\\_overview07\\_xFile.zip](#) 예제도 같이 살펴 보기 바랍니다.

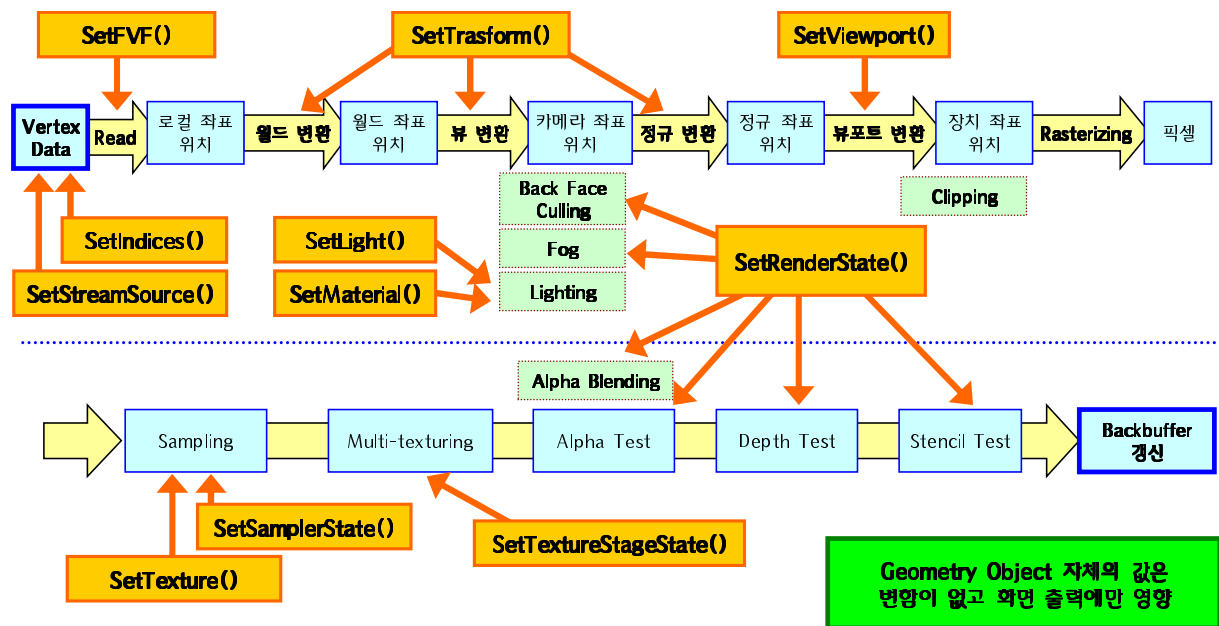


< m3d\_01\_overview07\_xFile.zip 예제 >

### 3.3.3 주요 렌더링 상태 설정 함수

다음은 상태 함수들이 파이프라인에 적용 되는 범위를 나타낸 그림입니다. D3D는 함수와 변수에 대한 일정한 이름 규칙(Naming Rule)이 있습니다.

이 이름 규칙은 자원을 할당할 때는 Create + "목적 객체" 또는 유틸리티 함수에서는 D3DX + "목적 객체"가 되는 것은 이전 시간에 이야기 한 적이 있으니 잘 알고 있는 내용입니다.



#### <주요 렌더링 설정 함수>

비슷하게 상태 또는 값을 "설정"을 할 때는 Set + "목적 대상"으로 합니다.

이 목적 대상의 이름은 그래픽 파이프라인에서 사용하는 단어들로 정합니다. 예를 들어 렌더링 상태의 경우 "RenderState", 변환에 대해서는 "Transform", 텍스처의 샘플링 단계에 대해서는 "SamplerState"로 이름을 정해서 좀 더 사용하기 편리하게 만들어 놓았습니다.

이러한 이유로 파이프라인을 대충 알아도 여러모로 도움이 많이 됩니다.

실습과제) DXSDK의 Sample 예제들을 하나씩 실행해 보시오.

실습과제) DXSDK의 Sample의 Tutorial 예제들을 C++의 클래스를 이용해서 재 구성 하시오.

### 3.3 Framework

3D 프로그램을 만들다 보면 자기 컴퓨터에서는 잘 실행되다가도 다른 사람의 컴퓨터에서는 잘 안 되는 경우가 많이 있습니다. 요즘은 많이 나아 졌지만 D3D는 예전부터 하드웨어 사양을 많이 타서 프로그래머들이 불평하는 일들이 많이 있었습니다. 이것은 D3D의 문제라기 보다는 충분한 테스트 없이 시간의 압박에 쫓겨 프로그래머가 순서나 값을 제대로 지키지 못해서 발생한 것인데, 그래픽 카드에서 프로그래머가 잘못된 부분을 넘어가도록 하드웨어가 만들어졌기 때문입니다. 그래서 저 같은 몇몇 프로그래머들은 아직도 특정 회사의 그래픽카드를 아직도 고집합니다. 지금 사용하고 있는 그래픽 카드는 과거 "A"사의 제품인데 이 회사의 그래픽카드를 계속 사용하게 된 것은 "N"사는 일부 렌더링을 잘못 설정해도 문제가 잘 나타나지 않는 반면에 "A"사에서 실행하면 컴퓨터가 Re-booting되어 "아 내가 잘못했구나" 문제를 바로 몸으로 깨닫게 해주기 때문입니다. 그렇다고 "N"사 제품이 더 낫다고 할 수도 없습니다. 2D가 3D 그래픽 파이프라인을 거치지 않는다면 포그(Fog)나 텍스처가 2D에 영향을 주지 않아야 합니다. "N"사의 일부 그래픽 카드는 글자를 출력하거나 그림을 2D로 출력할 때 포그, 조명을 비활성화 해야 합니다. 이것은 역으로 2D도 내부에서는 3D로 처리 되고 있다고 볼 수 있지요. 또 "N" 제품은 같은 계열이라도 렌더링의 Default 값이 약간 다를 때도 있습니다. 여러 군데서 각기 다른 환경의 컴퓨터를 가지고 강의를 하다 보니 이런 것을 알게 되었는데 여러분들도 좀 여유가 있으면 경쟁 관계에 있는 GPU를 만드는 회사의 제품에서 각각 테스트 하는 것이 좋습니다. 회사에서는 하나의 팀에서 사양이 다른 컴퓨터로 개발하는 것은 당연합니다.

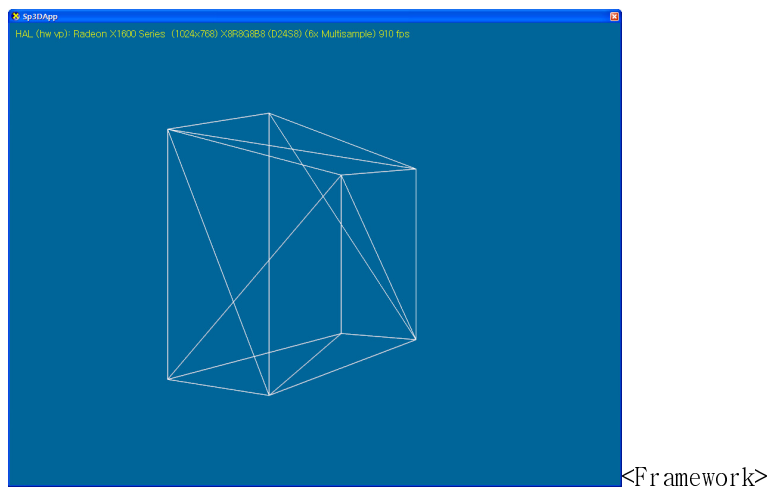
Code Framework은 좀 더 많은 컴퓨터에서 자신이 만든 프로그램이 잘 동작할 수 있도록 렌더링에 대한 기본적인 틀을 만드는 것입니다. DXSDK의 Sample Code Framework은 테스트가 잘되어 있는 코드입니다. 제가 잘해도 전세계의 모든 컴퓨터와 그래픽 카드를 상대로 전쟁을 벌이는 Microsoft사의 프로그래머가 만든 것보다 잘 만들 수 있다는 자신도 없고 해서 저 같은 경우 게임을 만들 때 Framework에 대한 코드 중에 D3D와 디바이스 생성, 윈도우 모드 변환(풀 모드(Full Mode) ← → 창 모드(Window Mode)) 등 디바이스가 안정적으로 처리해야 할 일들에 대한 코드는 SDK의 Sample 코드를 조금 수정해서 사용하고 있습니다.

2003 Summer 버전까지 Visual Studio에서 Wizard로 간편하게 Framework을 만들 수 있었는데 근래에는 이것이 없어진 것 같습니다. (있으면 알려주십시오.)

[m3d\\_02\\_auxiliary01\\_sample\\_framework.zip](#)은 DXSDK 2003 Summer 버전에서 Wizard로 만든 Framework를 수정한 코드입니다. Visual Studio 2005에서 DXSDK 2009 August 버전에서도 컴파일과 렌더링 테스트를 했습니다. (잘 동작 하니 염려 불들어 매 놓아도 됩니다.)

이 예제에 여러분은 키보드 마우스에 대한 인풋 시스템(Input System)을 추가해서 렌더링을 테스트 할 수 있도록 해야 합니다. 또한 Framework 코드에서 3D를 연습하고 배우면서 사운드-미디(Sound-Midi), 네트 워크 시스템을 등 게임에 필요한 시스템들을 추가시켜 가면서 점차 게임 코드로 만들기 바랍니다.

다음 그림은 [m3d\\_02\\_auxiliary01\\_sample\\_framework.zip](#)의 렌더링 화면입니다.



Framework으로 사용하려는 이 코드는 다음과 같은 클래스 구조로 구성되어 있습니다. 여기서 CD3DApplication은 윈도우, D3D, 디바이스, 2D Sprite, 시간 등을 관리합니다. 또한 윈도우 모드 변환에 안정적인 동작이 되도록 프로그램의 기초적인 시스템을 구성하고 있습니다.

주요 변수들은 UML로 표기해 놓았습니다. m\_hInst는 HINSTANCE 값입니다. m\_hWndFocus는 디바이스 생성의 윈도우 핸들입니다. 이 값은 메인 프로그램 윈도우의 m\_hWnd 값을 복사해서 사용하고 있습니다. 윈도우 핸들인 HWND 변수 값입니다. m\_pd3D는 D3D 객체, m\_pd3dDevice는 디바이스 객체, m\_d3dapp는 PRESENT PARAMETER 구조체 값입니다.

간편하게 2D를 출력하기 위해 m\_pd3dSprite를, 후면 버퍼의 색상 버퍼 주소를 m\_pd3dBackBuffer에 저장합니다.

m\_dwCreationWidth, m\_dwCreationHeight 변수는 DC의 너비와 높이, 즉 화면의 크기입니다. 800x600, 1024x768 값이 가장 많이 사용되는 값입니다.

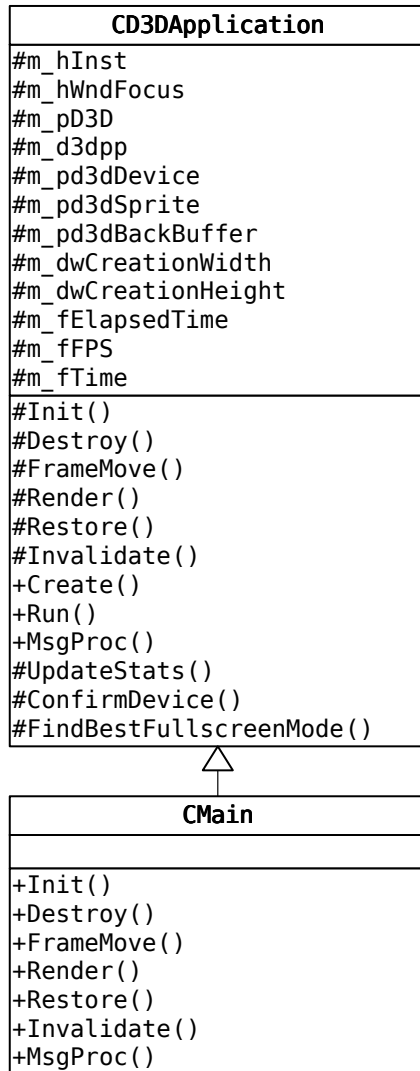
프로그램이 시작하고 나서부터 총 시간을 알려면 m\_fTime을 이용하고, 직전의 프레임 시간은 m\_fElapsedTime를 이용하면 됩니다. m\_ffPS는 초당 프레임인데 16번 렌더링을 한 후의 평균 값입니다.

Init(), Destroy(), FrameMove(), Render(), Restore(), Invalidate(), MsgProc() 함수는 Virtual 함수입니다. 이 함수는 CMain 클래스에서 재정의(Overriding) 해서 사용합니다.

CD3DApplication 클래스를 상속 받는 CMain 클래스에서 반드시 구현해야 할 것은 가상함수에 대한 재정의입니다.

Init() 함수는 디바이스가 만들어지고 나서 호출하게 되어 있습니다. 여러분은 게임 객체를 생성

할 때 이 함수 안에서 구현 해야 합니다.



Destroy() 함수는 Init() 함수와 대응 되는 게임 객체를 해제하기 위한 함수입니다. D3D 디바이스를 통해서 만든 모든 객체는 디바이스가 해제(Release)되기 전에 자원이 해제되어야 합니다. Framework구조는 이 함수를 디바이스 해제 전에 호출하고 있어서 여러분은 반드시 이 함수 안에서 게임 객체를 해제해야 합니다.

FrameMove(), Render() 함수는 Run() 함수의 while 루프안에서 호출 되는 함수로 FrameMove() 함수는 게임 자료의 갱신을 처리하기 위한 함수이고, Render() 함수는 화면에 출력하기 위한 함수입니다.

이렇게 자료의 갱신, 화면 출력을 따로 가져가는 이유는 렌더링이 자료의 갱신보다 처리에 시간이 많이 들어서 평등하지 않은 게임 유저의 컴퓨터 환경에서 발생하는 렌더링 속도 차이를 줄이고자 할 때에 느린 컴퓨터에서 일부 장면을 화면에 출력하지 않고 건너 뛸 수 있도록 갱신과 출력을 분리해 놓은 것입니다.

윈도우 모드 변환이 발생할 때 디바이스를 리셋(Reset) 해야 합니다. 만약 여러분이 자원의 메모리 풀을 D3DPPOOL\_DEFAULT 로 만들었다면 디바이스의 리셋에도 대비해야 하는데 이렇게 리셋에 영향을 받는 자원은 Restore() 함수에서 생성해야 합니다.

<Framework 구조>

Restore() 함수는 디바이스의 생성 후 Init() 함수를 처리한 후에 호출되는 함수 입니다. 또한 디바이스의 리셋 후에도 호출 되는 함수 입니다.

Restore()에서 생성한 자원은 Invalidate() 함수에서 해제하도록 합니다. Invalidate() 함수는 디바이스가 해제하기 전에 호출이 되고, 또한 윈도우 모드의 변환에서 디바이스 리셋 전에 호출되는 함수 입니다.

다시 강조한다면 D3DPPOOL\_DEFAULT로 만든 자원, 셰이더의 ID3DEffect 등은 Restore()에서 만들고 Invalidate()에서 해제합니다.

MsgProc() 함수는 윈도우 메시지 처리를 위한 함수 입니다. 게임에서 윈도우 메시지 시스템을 이용하는 경우는 별로 없지만 종종 네트워크의 Async-Select 모델을 사용할 때 윈도우 메시지가 이

용되므로 상속을 받아서 구현하도록 Framework가 구성되어 있습니다.

만약 여러분이 올바른 셰이더 버전이 선택 되도록 하려면 `CD3DApplication::ConfirmDevie()` 함수가 내용을 수정하십시오. 사용에 대한 예제를 주석으로 처리해 놓았습니다.

이 예제는 전체 모드(Full Mode)에서 바탕화면의 해상도로 설정하지 않고 화면이 그대로 늘어나는 구조로 되어 있습니다. 만약 이 부분이 맘에 들지 않거나 와이드(Wide) 화면을 구성하고 싶다면 `FindBestFullScreenMode()` 함수에 디바이스 선택 부분을 수정해야 합니다. 지금은 이 부분이 중요하지 않아 설명은 생략하겠습니다.

`UpdateStats()` 함수는 화면의 전환(Flipping)후에 시간, 프레임, 렌더링 속도 등을 처리하는 함수입니다. 특히 시간은 가장 중요하므로 시간 관련에서 추가하거나 고칠 필요가 있다면 이 함수의 내용을 수정해 사용하기 바랍니다.

지금까지 Framework 구조를 살펴 보았습니다. 여러분은 이 Framework 사용은 선택입니다. 안정적인 Framework를 만드는데 시간이 아깝거나 귀찮다면 이 Framework를 권합니다.

Framework을 코드들은 DXSDK 2003 Summer 버전에서 만들었습니다. 요즘 게임은 팀을 만들어서 제작을 많이 합니다. 개성이 강한 본인 만의 Framework을 고집하지 말고 팀원과 잘 협의해서 구조를 먼저 설계하기 바라며 시스템의 안정성을 바란다면 DXSDK Sample 예제들을 잘 분석 하기 바랍니다.



<후면 버퍼의 DC에 이미지와 문자를 출력한 예, DXSDK Wizard로 만든 코드를 수정해서 제작함, 이 방법은 게임에서 전혀 이용되지 않음>

## 3.4 Input System

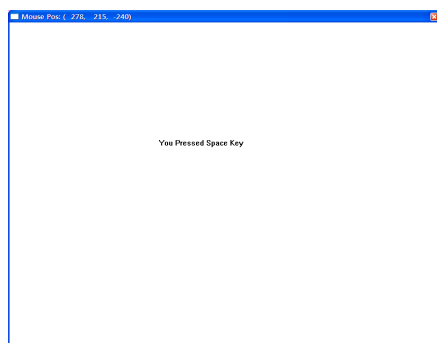
### 3.4.1 윈도우 API를 이용한 인풋 시스템

2D 게임 프로그램 응용에서 Window API를 이용해 키보드와 마우스를 배웠고 클래스로 구현했습니다. 내용을 정리하면 윈도우의 프로그램 방식은 이벤트 방식이 기본입니다. 따라서 키보드나 마우스와 같은 입력 장치들은 이벤트가 발생하면 메시지 풀에 이 내용을 저장합니다. 가장 간단한 코드는 다음과 같이 메시지 프로시저(Message Procedure)함수에서 처리하는 것입니다.

```
LRESULT CMain::MsgProc(...)  
...  
    switch( msg )  
    {  
        case WM_MOUSEWHEEL :  
            ...  
            return 0;  
        case WM_KEYDOWN :  
            ...  
            return 0;  
        case WM_LBUTTONDOWN :  
            ...  
            return 0;  
    }
```

이 방식이 가장 프로그램 하기가 쉽지만 문제는 인풋 클래스를 만들었다 하더라도 메시지 프로시저에서 이 클래스의 인스턴스를 호출해야 한다는 것입니다.

[m3d\\_02\\_auxiliary02\\_Input\\_A.zip](#)는 메시지 프로시저에서 가급적 키보드 마우스에 대한 이벤트를 적게 처리하기 위해서 키보드의 이벤트는 GetKeyboardState() 함수로 얻어서 처리합니다. 마우스의 위치는 GetCursorPos() 함수와 ScreenToClient()로 구현한 예제입니다.



<Window API를 이용한 마우스 이벤트 처리>

여러분은 마우스의 이벤트 중에서 왼쪽 버튼, 오른쪽 버튼, 가운데 버튼은 다음과 같이 가상키로 정의되어 있어서 GetKeyboardState() 함수로 처리된 키보드 배열의 인덱스나 GetAsyncKeyState() 직접 얻을 수 있습니다.

```
// Window에서 정의한 마우스에 대한 의 가상 키
#define VK_LBUTTON      0x01
#define VK_RBUTTON      0x02
#define VK_MBUTTON      0x04

// GetKeyboardState()에서 얻은 가상 키 이벤트 중에서 마우스 이벤트를 저장
buttonEventL = m_KeyCur[VK_LBUTTON];    // Left Button
ButtonEventR = m_KeyCur[VK_RBUTTON];    // Right Button
ButtonEventM = m_KeyCur[VK_MBUTTON];    // Middle Button
```

이렇게 GetKeyboardState() 함수로 마우스의 이벤트를 얻고, GetCursorPos() 함수와 ScreenToClient() 함수를 조합해서 마우스의 위치를 처리하면 더 이상 문제가 없을 것이라 생각이 들지만 남아 있는 문제는 마우스의 휠(wheel)에 대한 위치입니다. 이 휠에 대한 위치를 구하기 위해서 여러분은 필연적으로 메시지 프로시저 안에서 다음과 같은 코드를 작성해야만 합니다.

```
switch( msg )
{
    case WM_MOUSEWHEEL:
    {
        INT c= HIWORD(wParam);
        INT d= LOWORD(wParam);
        c = short( c );
        m_vcCur.z += FLOAT(c);
        break;
    }
}
```

또한 이 구문이 완성되기 위해서는 \_WIN32\_WINNT 정의가 0x0400 이상 값으로 windows.h 앞에 와야 합니다.

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
```



이렇게 API를 이용해서 좀 더 다듬은 코드가 [http://3dapi.com/bs12\\_2d\\_app/Sp15\\_Input.zip](http://3dapi.com/bs12_2d_app/Sp15_Input.zip) 입니다.

### 3.4.2 Direct Input

인풋을 프로시저에서 완전히 분리하고자 한다면 Direct Input(이하 DInput)을 사용하십시오. DInput은 키보드 마우스뿐만 아니라 조이스틱 등 입력에 대한 컴퓨터의 주변장치들에 대해서 아주 편리하게 자원을 가져다 쓸 수 있습니다.

DInput은 D3D처럼 COM을 상속 받아 구현되어 있습니다. 따라서 D3D의 디바이스처럼 장치의 생성과 해제는 비슷하게 구성되어 있습니다. 또한 키보드, 마우스, 조이스틱 등 입력들에 대해서 장치의 생성, 데이터 얻기, 입력 장치 해제는 거의 같은 코드로 구성되어 있어서 하나의 장치에 대해서 작성한 코드를 일부만 수정해서 다른 장치에도 그대로 적용할 수가 있습니다. (이 부분이 아주 매력적입니다.)

DInput을 통한 코드 작성과 처리 과정은 항상 다음 순서대로 진행 됩니다.

0. DInput 생성: DirectInput8Create(&"DInput 객체")
1. 입력 장치 생성: "DInput 객체"->CreateDevice("GUID장치이름", &"장치 객체")
2. 데이터 포맷 설정: "장치 객체"->SetDataFormat()
3. 응용 프로그램과 입력 장치의 관계 설정:  
"장치 객체"->SetCooperativeLevel()
4. 입력 장치 획득/반납  
"장치 객체"->Acquire()  
"장치 객체"->Unacquire()
3. 입력 장치로부터 데이터 얻기  
"장치 객체"->GetDeviceState()
4. 입력 장치 해제
5. DInput 해제: "장치 객체"->->Release()

이제 본격적으로 위의 순서를 이용해서 코드를 작성해 봅시다. 제일 처음으로 라이브러리 링크입니다.

```
#pragma comment(lib, "dinput8.lib")
#pragma comment(lib, "dxguid.lib")
```

DXSDK 9.0 버전은 여전히 8.0에서 만든 인풋 시스템을 그대로 사용 있는데 이것은 키보드 마우스 같은 입력 장치들이 과거에 비해서 크게 달라진 것이 거의 없기 때문입니다. 위의 두 라이브러리 중에서 "dinput8.lib"는 장치 자체에 대한 코드입니다. 윈도우 운영체제는 장치들을 GUID(구이드)

이름으로 관리합니다. 이 가이드 이름을 사용해야만 장치를 얻어 올 수 있습니다.

다음으로 다음과 같이 DInput 버전을 명시하고 헤더 파일을 포함해야 합니다.

```
#define DIRECTINPUT_VERSION    0x0800
#include <dinput.h>
```

위의 코드처럼 DIRECTINPUT\_VERSION을 명시적으로 하지 않으면 컴파일러는 현재사용하고 있는 SDK의 최종 버전으로 컴파일 합니다. 현재로서는 문제가 없지만 이후에도 지금 만든 코드들을 그대로 사용하고 싶다면 DInput에 대한 버전을 명시하는 것이 좋습니다.

## - 0. DInput 생성/해제

DInput 객체는 입력 장치들을 관리하기 위한 객체 입니다. DInput을 통해서 사용하려는 장치의 존재 여부와 종류 등을 파악할 수 있습니다. 이 객체의 생성은 다음과 같습니다.

```
LPDIRECTINPUT8  m_pDInput;           //DInput Instance
...
HINSTANCE        hInst = (HINSTANCE) GetModuleHandle(NULL);
DirectInput8Create(hInst,DIRECTINPUT_VERSION,IID_IDirectInput8, (void **)&m_pDInput,NULL);
```

DirectInput8Create() 함수의 첫 번째 인수는 HINSTANCE값입니다. 현재 프로그램의 HINSTANCE는 WinAPI 함수인 GetModuleHandle() 함수를 통해서 얻을 수 있습니다. 두 번째 인수는 DInput 버전입니다. 세 번째 인수는 DirectInput에 대한 가이드 번호입니다. 네 번째 인수는 DInput 객체의 주소입니다.

DInput 객체는 COM을 상속 받으므로 Release() 함수를 통해서 해제합니다.

```
m_pDInput->Release();
```

## - 1. 입력 장치 생성/ 해제

키보드, 마우스 등 필요한 장치들은 DInput의 CreateDevice() 함수를 통해서 생성합니다. 다음 코드는 키보드와 마우스 장치를 생성하는 예제 입니다.

```
LPDIRECTINPUTDEVICE8  m_pDiKey;           // Keyboard Device Instance
LPDIRECTINPUTDEVICE8  m_pDiMs;           // Mouse Device Instance
...
```

```
// Keyboard Device 생성
```

```
m_pDInput->CreateDevice(GUID_SysKeyboard, &m_pDiKey, NULL);
```

```
// Mouse Device 생성
```

```
m_pDInput->CreateDevice(GUID_SysMouse, &m_pDiMs, NULL);
```

키보드와 마우스 모두 같은 `IDirectInputDevice8` 인터페이스를 사용합니다. 만약 여러분이 또 다른 입력객체를 추가하고 생성할 때는 그냥 `IDirectInputDevice8` 인터페이스를 추가하고 `DInput` 객체의 `CreateDevice("해당 장치의 가이드 번호")` 함수를 호출하면 객체가 만들어집니다.

장치 인스턴스들도 COM을 상속받으므로 해제는 당연히 `Release()` 함수로 합니다

```
m_pDiKey->Release();
```

```
m_pDiMs->Release();
```

## - 2. 데이터 포맷 설정

텍스처를 생성할 때 `D3DFMT` 열거를 이용해서 적절한 포맷을 설정해야 화면에 올바르게 출력 되듯이 장치 또한 적절한 데이터 포맷을 설정해야만 장치로부터 얻은 데이터를 올바르게 해석하고 이용할 수 있습니다.

같은 종류의 입력 장치라도 다른 형식을 가질 수 있습니다. 여러 포맷 중에서 가장 기본적인 포맷은 키보드에 대해서는 `c_dfDIKeyboard`, 마우스는 `c_dfDIMouse`, 조이스틱은 `c_dfDIJoystick` 입니다.

장치들은 크게 변환된 것이 없어서 기본적인 이 3개 포맷 정도만 다음과 같이 사용해도 충분합니다.

```
m_pDiKey->SetDataFormat(&c_dfDIKeyboard);
```

```
m_pDiMs->SetDataFormat(&c_dfDIMouse);
```

## - 3. 응용 프로그램과 입력 장치의 관계 설정

협력 레벨(Cooperative Level)이라는 것은 프로그램과 운영체제에 대해 해당 장치에 대한 사용 관계를 말합니다.

게임 프로그램은 운영체제의 자원을 독점하려는 경향이 있습니다. 기존의 메시지 기반 방식으로는 이런 독점은 불가능합니다. DirectX는 협력 레벨의 설정에 따라서 해당 자원을 독점할 수도 있습니다.

협력레벨은 윈도우의 상태에 따라 화면에서 해당 프로그램이 맨 앞에 오는 경우(FORE GROUND)와 그 이외 경우(BACK GROUND), 그리고 해당 장치의 접근과 이에 대한 자원의 독점여부를 플래그를 조합해서 SetCooperativeLevel() 함수를 이용해 결정할 수 있습니다.

```
m_pDiKey->SetCooperativeLevel(m_hWnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);  
m_pDiMs->SetCooperativeLevel(m_hWnd  
    , DISCL_FOREGROUND | DISCL_NONEXCLUSIVE | DISCL_NOWINKEY);
```


협력 레벨 플래그는 '|' 연산자로 결합해서 사용합니다. 플래그에 대한 설명은 다음과 같습니다.

DISCL\_EXCLUSIVE: 다른 프로그램들은 해당 장치를 이용 못하게 하는 장치에 대한 배타적 독점권을 행사합니다. 윈도우는 메시지 기반으로 작동합니다. 이 플래그를 설정하면 다른 프로그램에서는 해당 장치의 어떠한 이벤트도 받지 못하는 결과를 만듭니다. 물론 해당 프로그램도 장치에 대한 이벤트는 Message Procedure 함수에서 처리되지 못하고 오직 DInput을 통해서만 처리가 됩니다.

DISCL\_NONEXCLUSIVE: 장치에 대한 비 배타적이며 가장 일반적인 방법입니다.

DISCL\_FOREGROUND: 협력 레벨로 설정한 해당 윈도우(hWnd) 프로그램이 가장 앞에 올 때만 장치를 이용합니다.

DISCL\_BACKGROUND : 해당 윈도우(hWnd)의 상태에 상관 없이(Active 상태가 아니어도) 항상 동작하게 합니다.

DISCL\_NOWINKEY: 키보드를 보면  Window logo key가 있습니다. 이 logo 키와 조합하는 것을 중지 합니다.

위의 코드 "DISCL\_FOREGROUND | DISCL\_NONEXCLUSIVE DISCL\_FOREGROUND"는 프로그램이 가장 앞에 올 때만 장치를 이용하며 장치를 독점하지 않고 다른 프로그램들과 공유해 사용하겠다는 것입니다.

#### - 4. 데이터 얻기

장치 객체에서 데이터를 얻기 위해서는 사전에 데이터 접근 승인이 필요합니다.

접근 승인은 IDirectInputDevice8::Acquire() 함수를 이용합니다.

```
m_pDiKey->Acquire();  
m_pDiMs->Acquire();
```

승인에 대한 반납은 IDirectInputDevice8::Unacquire() 함수를 사용합니다.

```
m_pDiKey-> Unacquire();  
m_pDiMs-> Unacquire ();
```

승인이 되면 데이터를 가져올 수 있습니다. 키보드에 대한 데이터는 unsigned char 256 바이트 배열에 다음과 같이 가져옵니다.

```
BYTE    KeyCur[256];  
m_pDiKey->GetDeviceState(sizeof(KeyCur), (LPVOID)KeyCur);
```

마우스는 DIMOUSESTATE 구조체를 이용해서 가져옵니다.

```
m_pDiMs->GetDeviceState(sizeof(DIMOUSESTATE), &MsStCur);
```

때로는 마우스의 버튼 8개가 지원이 되는 DIMOUSESTATE2 구조체를 이용할 수 있습니다. 이 때는 마우스에 대한 데이터 포맷을 다음과 같이 변경해야 합니다.

```
m_pDiMs->SetDataFormat(&c_dfDIMouse2);
```

원칙적으로 Acquire() 함수를 장치 객체를 만들고 한 번만 호출하면 계속 사용할 것 같지만 DInput은 협력 레벨 설정에 따라서 자동으로 승인이 반납될 수 있습니다. 따라서 다음과 같이 데이터 가져오기가 실패하면 다시 승인 받고 데이터를 가져와야 합니다. 이 부분이 실패가 되면 프로그램의 이상이 있는 경우이니 종료 하는 것이 좋습니다.

```
// Keyboard  
if (FAILED(m_pDiKey->GetDeviceState(sizeof(KeyCur), (LPVOID)KeyCur)))  
{  
    memset(KeyCur, 0, sizeof(KeyCur));  
  
    if (FAILED(m_pDiKey->Acquire()))  
        return -1;    // Acquire가 안 되면 프로그램 문제  
  
    if (FAILED(m_pDiKey->GetDeviceState(sizeof(KeyCur), (LPVOID)KeyCur)))  
        return -1;    // Acquire가 되어도 GetDeviceState() 실패면 프로그램 문제  
}  
  
// Mouse  
if (FAILED(m_pDiMs->GetDeviceState(sizeof(DIMOUSESTATE), &MsStCur)))  
{  
    if (FAILED(m_pDiMs->Acquire()))
```

```

        return -1;
    if (FAILED(m_pDiMs->GetDeviceState(sizeof(DIMOUSESTATE), &MsStCur)))
        return -1;
}

```

키보드에 대해서 윈도우 API는 "VK\_" 접두어가 붙은 가상 키를 사용했습니다. DInput은 "DIK\_" 접두어를 붙여서 사용합니다. 또한 알파벳 A~Z 까지 DIK\_A ~DIK\_Z 가 선언되어 있습니다. 주의할 것은 "VK\_"와 "DIK\_"의 키 값이 일치 하지 않습니다. 이것은 DInput에서는 "DIK\_" 접두어를 붙여서 사용해야 문제가 생기지 않는 것입니다.

EX) KeyCur[VK\_LEFT] ≠ KeyCur[DIK\_LEFT], KeyCur['A'] ≠ KeyCur[DIK\_A]

마우스는 DIMOUSESTATE 구조체를 이용합니다. 이 구조체 변수 lX, lY, lZ 에는 마우스의 위치가 저장 되는데 이 위치 값은 절대적인 값이 아니라 이전 위치에 대한 상대적인 값입니다. 따라서 여러분은 이 값들을 계속 누적을 시켜서 사용해야 합니다.

다음과 같이 프로그램이 시작될 때 마우스의 위치를 특정 위치에서 시작하도록 합니다.

```

RECT rct={0};
::GetClientRect(m_hWnd, &rct);

m_vcMsCur.x = float( (rct.right - rct.left)/2 );
m_vcMsCur.y = float( (rct.bottom- rct.top )/2 );

POINT Point={ int(m_vcMsCur.x), int(m_vcMsCur.y)};
::ClientToScreen(m_hWnd, &Point);
::SetCursorPos(Point.x, Point.y);

```

다음으로 GetDeviceState() 함수 호출 후에 마우스의 위치를 수정합니다.

```

// 현재 위치 += D3DXVECTOR3(st.lX, st.lY, st.lZ);
m_pDiMs->GetDeviceState(sizeof(DIMOUSESTATE), &MsStCur);
m_vcMsCur.x += FLOAT(MsStCur.lX);
m_vcMsCur.y += FLOAT(MsStCur.lY);
m_vcMsCur.z += FLOAT(MsStCur.lZ);

```

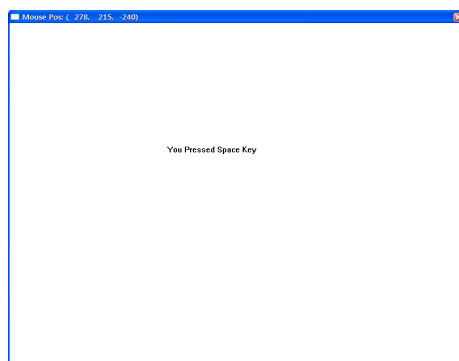
마지막에 커서의 위치를 다시 수정합니다.

```
POINT Point={ int(m_vcMsCur.x), int(m_vcMsCur.y)};
::ClientToScreen(m_hWnd, &Point);
::SetCursorPos(Point.x, Point.y);
```

이렇게 현재위치를 계속 누적 시키는 방법이 마음에 들지 않는다면 휠 부분만 누적하고 나머지 부분은 WinAPI의 함수 GetCursorPos(), ScreenToClient() 함수로 마우스의 현재 위치를 정하는 것도 좋습니다.

```
POINT MsPos;
::GetCursorPos(&MsPos);
::ScreenToClient(m_hWnd, &MsPos);
m_vcMsCur.x = FLOAT(MsPos.x);
m_vcMsCur.y = FLOAT(MsPos.y);
m_vcMsCur.z += FLOAT(MsStCur.lZ);
```

지금까지 DInput을 사용해서 입력 장치인 키보드와 마우스에 대해서 객체를 생성하고 데이터를 가져오는 법을 살펴 보았습니다. 전체 코드는 [m3d\\_02\\_auxiliary03\\_Input\\_D.zip](#)을 참고 하기 바랍니다.



<Direct Input을 이용한 키보드 마우스 처리 예>

처음에 DInput을 사용하면 D3D처럼 하드웨어 가속을 받아 빠르게 동작할 것처럼 생각되지만 컴퓨터 시스템에서 입력 장치는 비디오 카드나 하드 디스크 보다 훨씬 느리게 작동하는 장치 입니다. 따라서 WinAPI 함수를 이용해서 작성한 코드와 DInput로 작성한 코드의 실제 성능은 차이가 눈에 보일 정도로 크게 나지 않습니다. DInput은 메시지 프로시저 함수에서 호출하지 않고, 또한 독점 모드를 설정할 수 있는 장점이 있는 반면에 이것은 언제까지나 DInput을 사용할 수 있는 환경에서만 가능하다는 것입니다.

모바일과 같은 환경에서 아직까지 DInput는 널리 사용되고 있지 않습니다. 만약 PC, Mobile 등의 시스템에서 윈도우 운영 체제가 사용되고 있다면 API를 이용해서 작성하는 것이 이식성에서 도움이 될 수도 있습니다. 그래도 DInput을 배웠으니 한 번쯤 이 것을 가지고 입력 시스템을 만드는 것이 좋습니다.

실습 과제) DInput를 이용해서 키보드와 마우스에 대한 Input 클래스를 만드시오.  
실습 과제) 이 Input 클래스를 이용해서 삼각형을 상, 하, 좌, 우로 이동시키시오.

## 3.5 Extension Utility

D3DXCreateFileFromEx() 함수를 통해서 우리는 파일에서 텍스처를 만들었습니다. 사실 이 함수의 내부는 먼저 fopen() 함수 등을 이용해 파일을 열고, 파일 헤더에 저장된 이미지 정보를 바탕으로 픽셀을 들을 읽은 다음 pDevice->CreateTexture() 함수로 텍스처를 만들고 파일에서 읽은 픽셀을 텍스처에 채우는 코드들로 구성되어 있습니다.

Extension Utility는 D3D의 기능들을 조합해서 자주 사용되는 반복적인 코드들을 유틸리티로 만든 것입니다. 이 유틸리티 안에는 함수도 있고 객체도 있습니다. "D3DX" 접두어로 시작하는 모든 함수는 Extension Utility 인데 이 함수들은 여러분이 만든 함수로도 얼마든지 대체 가능하다는 것이기도 합니다.

유틸리티 객체는 함수와는 좀 다릅니다. 이 말 그대로 객체이기 때문에 인스턴스를 만들어야 사용할 수 있습니다. 인스턴스 생성은 "D3DX" + Create + "유틸리티 객체"(디바이스,..., &"유틸리티 객체")로 구성된 함수로 만듭니다. 그리고 유틸리티 객체들도 COM을 상속 받아서 반드시 Release() 함수를 호출해서 해제합니다. ("유틸리티 객체"->Release().)

유틸리티 객체를 2D와 3D 두 종류로 나누어서 살펴 봅시다.

### 3.5.1 2D Extension Utility

자주 사용되는 2D 유틸리티는 ID3DXSprite, ID3DXFont, ID3DXLine 세 가지가 있습니다.

ID3DXSprite는 텍스처를 화면에 출력해 주는 함수로 이미 2D 게임 프로그램에서 사용해 봤던 객체입니다.

#### - 1. ID3DXSprite 객체

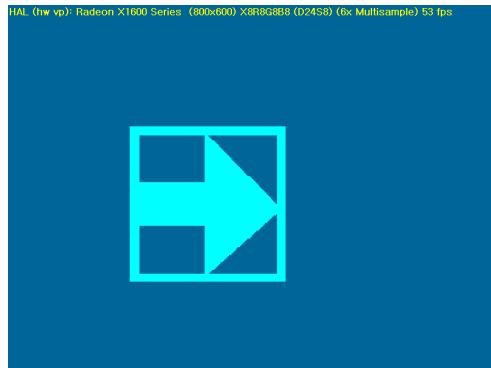
이 객체의 사용은 2D 게임부분에 잘 되어있으니 객체를 생성하고 활용하는 부분만 정리해 봅시다.

```
객체 저장: LPD3DXSPRITE m_pd3dSprite; // 2D Sprite 객체
객체 생성: D3DXCreateSprite(m_pd3dDevice, &m_pd3dSprite); // Sprite create
객체 해제: m_pd3dSprite->Release();
로스트 상태: 디바이스 리셋 전에 호출 ➔ m_pd3dSprite->OnLostDevice();
리셋: 디바이스 리셋 후 호출 ➔ m_pd3dSprite->OnResetDevice();
```



렌더링: Begin()/End() 함수 안에서 Draw("텍스처 포인터") 함수 호출로 렌더링

```
m_pd3dSprite->Begin(D3DXSPRITE_ALPHABLEND);
    m_pd3dSprite->Draw(m_pTx2D, &rc, NULL, &vcPos, D3DCOLOR(1,1,1,1));
m_pd3dSprite->End();
```

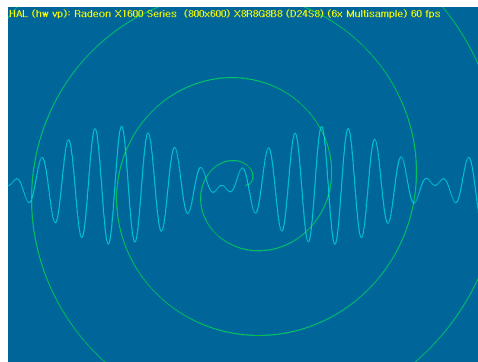


< m3d\_02\_extension02\_DxSprite.zip>

이 객체는 3D 기초 과정이 끝나고 나서 직접 구현해 봐야 하는 객체이기도 합니다. 객체의 인터페이스 구조들을 잘 연구하기 바랍니다.

## - 2. ID3DXLine 객체

ID3DXLine 객체는 화면에 다음 그림과 같은 2차원 선들을 그려주는 객체 입니다.



< m3d\_02\_extension01\_DxLine>

객체의 사용법은 ID3DXSprite와 거의 유사 합니다.

다음과 같이 먼저 객체를 저장할 변수를 선언합니다.

```
LPD3DXLINE    m_pd3dLine;
```

객체를 D3DXCreateLine() 함수를 이용해서 만듭니다.

```
D3DXCreateLine(m_pd3dDevice,&m_pd3dLine);
```

윈도우 모드 전환에 대해서 적절히 OnResetDevice() 함수와 OnLostDevice() 함수를 호출합니다.

```
m_pd3dLine->OnResetDevice();  
...  
m_pd3dLine->OnLostDevice();
```

객체의 Begin()/End() 함수 안에서 2차원 벡터 배열을 주어서 선을 그립니다.

```
VEC2 Line[2];  
Line[0] =VEC2( 0, 300);  
Line[1] =VEC2(1000, 300);  
  
m_pd3dLine->Begin();  
...  
m_pd3dLine->Draw(Line, 2, 0xff00ff00);  
...  
m_pd3dLine->End();
```

객체를 해제 하려면 Release() 함수를 호출합니다.

```
m_pd3dLine->Release();
```

전체 코드는 [m3d\\_02\\_extension01\\_DxLine.zip](#)을 참고 하기 바랍니다.

### - 3. ID3DXFont 객체

ID3DXFont 객체의 예는 이미 위의 두 예제에서 화면 상태를 출력 문자열에서 볼 수 있습니다. 다른 2D 객체와 비슷하게 동작합니다. 정리하면 다음과 같습니다.

```
객체 저장: LPD3DXFONT m_pD3DXFont; // 2D 폰트 객체  
객체 생성: D3DXCreateFont{Indirect}(m_pd3dDevice,..., & m_pD3DXFont); // 폰트 생성  
객체 해제: m_pD3DXFont->Release();  
로스트 상태: 디바이스 리셋 전에 호출 ➔ m_pD3DXFont->OnLostDevice();  
리셋: 디바이스 리셋 후 호출 ➔ m_pD3DXFont->OnResetDevice();
```

문자열 출력: DrawText(). Begin()/End() 함수 없음.

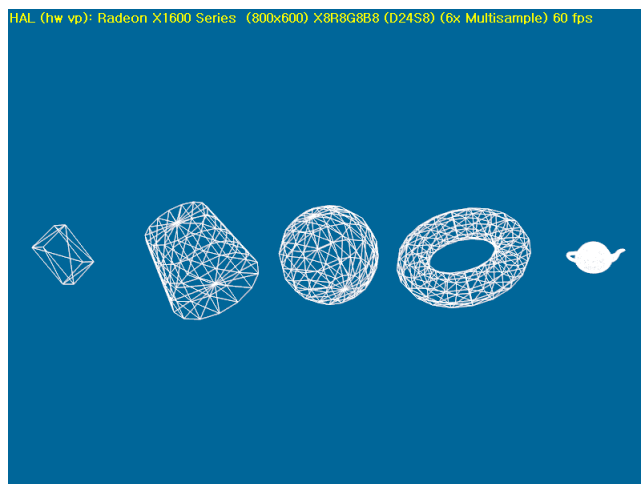
```
m_pD3DXFont->DrawText(NULL, szMsg, -1, &rc, 0, D3DCOLOR(1,1,0,1) );
```

### 3.5.2 3D Extension Utility

3D에서 장면을 연출하기 위해 반드시 필요한 것이 정점 데이터 입니다. 자주 사용되는 Geometry Object를 매 번 코딩 한다는 것은 고통이 아닐 수 없습니다.

DXSDK는 육면체(Box), 구(Sphere), Torus, 원통(Cylinder), 주전자(Teapot)에 대한 기하 오브젝트 (Geometry Object)를 제공합니다. 이 오브젝트들은 간편한 테스트에서 자주 사용이 되기도 합니다.

다음 그림은 DXSDK의 Geometry 객체의 렌더링 화면입니다.



<3D 확장 기하 객체>

이 확장 객체들은 LPD3DXMESH 변수에 객체를 저장합니다.

```
LPD3DXMESH      m_pBox;  
LPD3DXMESH      m_pCylinder;  
LPD3DXMESH      m_pSphere;  
LPD3DXMESH      m_pTorus;  
LPD3DXMESH      m_pTeapot;
```

객체의 생성은 D3DXCreate+"객체종류"(...) 형식의 함수로 다음과 같이 생성 합니다.

```
D3DXCreateBox(m_pDev, 1, 2, 3,&m_pBox,NULL);  
D3DXCreateCylinder(m_pDev, 2, 3, 5, 15, 4, &m_pCylinder, NULL);  
D3DXCreateSphere(m_pDev, 3, 16, 8, &m_pSphere, NULL);
```

```
D3DXCreateTorus(m_pDev, 1, 3, 9, 20, &m_pTorus, NULL);  
D3DXCreateTeapot(m_pDev, &m_pTeapot, 0);
```

D3DXCreateBox() 함수의 2 번째부터 4 번째 인수는 직육면체의 x축, y축, z축에 대한 길이입니다. D3DXCreateCylinder() 함수는 z축 방향으로 원통을 만드는데 2 번째 인수는 -z의 반지름이고, 3 번째 인수는 +z의 반지름 입니다. 4 번째 인수는 z축에 대한 길이를, 5 번째와 6 번째 인수는 원통의 해상도에 대한 원호의 단편을 지정하는 인수입니다. D3DXCreateSphere() 함수는 구를 만들어 주는 함수입니다. 2 번째 인수부터 반지름, 구의 동경에 대한 단편의 수, 구의 위도에 대한 단편의 수를 지정하는 인수입니다. D3DXCreateTeapot() 함수는 하나의 완전한 주전자를 만들어 냅니다.

이 모든 3D 확장 객체의 해제는 Release() 함수로 합니다.

```
m_pBox->Release();  
m_pCylinder->Release();  
m_pSphere->Release();  
m_pTorus->Release();  
m_pTeapot->Release();
```

확장 객체들은 "객체"->DrawSubset(0); 형태로 렌더링 합니다.

그런데 이 객체들은 모두 자신의 지역 좌표로 정점이 생성되어 있습니다. 따라서 이 확장 객체의 렌더링을 위해서 월드 변환 행렬이 필요합니다. 즉, 다음과 같이 하나의 객체 마다 월드 행렬을 설정하고 렌더링 해야 합니다.

```
// Draw Box.  
m_mtWld._41 = -16.f;    m_mtWld._42 = 2.f;    m_mtWld._43 = 20.f;
```

```
m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);  
m_pBox->DrawSubset(0);
```

```
//Draw Cylinder  
m_mtWld._41 = -8.f;    m_mtWld._42 = 2.f;    m_mtWld._43 = 20.f;
```

```
m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);  
m_pCylinder->DrawSubset(0);
```

```
//Draw Sphere
```

```

m_mtWld._41 = -0.f;      m_mtWld._42 = 2.f;      m_mtWld._43 = 20.f;
m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);
m_pSphere->DrawSubset(0);

```

//Draw Torus

```

m_mtWld._41 = 8.f;      m_mtWld._42 = 2.f;      m_mtWld._43 = 20.f;

m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);
m_pTorus->DrawSubset(0);

```

//Draw Teapot

```

m_mtWld._41 = 16.f;      m_mtWld._42 = 2.f;      m_mtWld._43 = 20.f;

m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);
m_pTeapot->DrawSubset(0);

```

// 파이프라인의 월드 행렬을 단위 행렬로 초기화(중요)

```

D3DXMatrixIdentity(&m_mtWld);
m_pDev->SetTransform(D3DTS_WORLD, &m_mtWld);

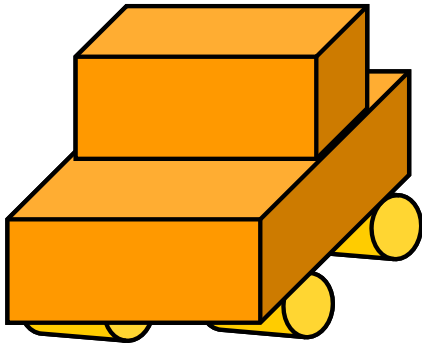
```

이 코드는 하나의 행렬을 가지고 매번 새로 설정해서 사용하고 있습니다. 여러분은 이렇게 하지 말고 반드시 모든 객체에 행렬을 따로 두어 렌더링을 하기 바랍니다. 위의 코드에서 가장 중요한 부분은 그래픽 파이프라인의 월드 행렬을 설정했다면 반드시 초기화를 해 놓아야 다른 렌더링에 영향을 주지 않는다는 것입니다.

마지막 2 줄에 행렬을 단위 행렬로 만들고 이 행렬을 디바이스의 월드 행렬에 연결하고 있음을 잘 기억하기 바랍니다.

전체 코드는 m3d\_02\_extension03\_DxGeometry.zip을 참고 하세요.

실습 과제) 다음 그림과 같이 Direct3D의 Extension Utility를 이용해서 박스 2, 원기둥 4개로 구성된 자동차를 완성하시오.



실습 과제) 위의 과제를 수정해서 자동차에 대한 월드 행렬을 만들고 이 행렬을 가지고 자동차를 이동 시키시오.

실습 과제)Extension Utility의 구를 가지고 월드 행렬을 이용해서 태양, 지구, 달의 운동을 표현해 보시오. 태양 달 지구의 자전과 공전은 전부 반 시계 방향이며 특히 지구는 23.5도 기울어져 있는 상태에서 자전을 해야 합니다.