3D Game Programming Basic with Direct3D

4. Primitive

이 장부터 여러분은 고정 기능 파이프라인에서 3D에 대한 초보적인 프로그램을 배울 것입니다. 고정 기능을 이용한 프로그램이 어느 정도 손에 익게 되면 이 후에 쉐이더도 쉽게 배울 수 있습니다. 명심해야 할 것은 우리는 게임 프로그램을 위해서 3D를 배우는 것입니다. 쉐이더, 수학, 물리를 다 배워서 좋은 게임을 만들 수 있을 것 같지만 실제로 그런 경우는 거의 없습니다. 항상 어떻게 조합을 해서 좀 더 재미 있는 게임을 만들까에 대한 고민을 하는 경우에만 좋은 게임이 나올 확률이 높아지는 것입니다. 우리는 자신이 알고 있는 것을 게임에 어떻게 적용해 볼 것인지 항상 생각하고 적용해 노력을 해야 합니다.

앞 장까지 3D 프로그램을 위해 D3D와 그래픽 파이프라인, 그리고 파이프라인에서 정점 데이터가 처리되는 과정을 배웠습니다. 프로그램을 잘 연습할 수 있도록 Framework, 인풋 시스템, Extension Utility들을 다루어 보았습니다. 이 장은 렌더링 파이프라인에 필 수로 입력해야 하는 정점을 만드는 방법, 정점들을 구성하는 프리미티브, 여러 공간에 만들어진 데이터를 출력하는 법을 배울 것입니다.
그 시작으로 정점 버퍼(Vertex Buffer)부터 설명을 시작하겠습니다.

4.1 정점 버퍼(Vertex Buffer)

정점은 사람의 몸에 비유 한다면 분자라 할 수 있습니다. 이런 정점이 규칙을 따라 생명의 기본 단위인 세포처럼 렌더링의 기본 단위로 구성이 되면 이 것을 프리미티브가 됩니다.
고속 장치와 저속 장치의 속도차이를 극복하기 위해 고속 장치의 일부 메모리 영역에 일시적으로 데이터를 보관하는 임시 기억 공간이 버퍼입니다. 정점 버퍼는 그 목적이 장치에 출력하기 위해서 가장 기본이 되는 정점 데이터를 저장하고 있는 기억 공간입니다. 이 기억 공간은 시스템 메모리, 비디오 메모리 등 어디든지 될 수 있고 D3D를 통해서 생성하거나 아니면 여러분이 new나 malloc()과 같은 동적 메모리 생성함수를 이용해서 버퍼를 만들 수 있습니다.

그런데 이 정점 버퍼는 아무렇게나 만드는 것이 아닙니다. 이 버퍼는 렌더링이 목적이므로 일정한 규칙을 따라야 합니다. 그것을 c언어의 구조체로 표현한 것이 정점 구조체 입니다.
정점 구조체는 항상 위치 값을 가지고 있어야 합니다. 그렇지 않으면 그래픽 파이프라인의 변환에서 처리가 안됩니다.
D3D의 고정 기능 파이프라인에서 정점의 위치 값은 항상 x, y, z 3차원이고 각각 32비트 float형으로 구성해야 합니다. (쉐이더는 3차원이 아니어도 되지만 반드시 위치 1차원이상 주어져야 합니

다.) 일단 정점의 위치만 확보되어 있으면 렌더링이 가능합니다.
3차원 x, y, z가 포함된 정점 구조체에 조명에 대한 반사, 색상, 매핑(Mapping)을 위한 텍스처 좌표 등 렌더링에서 필요한 데이터를 추가할 수 있습니다. 비록 순서를 지켜야 하는 단점이 있지만 렌더링에 필요한 내용을 선택할 수 있어서 이것을 "유연한 정점 포맷"(FVT: Flexible Vertex Format)이라 합니다.
다음 그림은 고정 기능 파이프라인에서 정점 구조체에 추가할 수 있는FVF 코드들입니다.

Position (untransformed or transform x, y, z)	x-coordinate (float)
	y-coordinate (float)
	z-coordinate (float)
RHW (transformed vertices only)	RHW (float)
Blending Weight Data (1,2, or 3 floats) D3DFVF_LASTBETA_UBYTE4 → DWORD	1st-5th Blend Weights
Vertex Normal (untransformed vertices only)	Normal x (float)
	Normal y (float)
	Normal z (float)
Vertex Point Size	Point size (float)
Diffuse Color	Diffuse RGBA(DWORD)
Specular Color	Specular RGBA(DWORD)
Texture Coordinate Set 1 (1,2,3, or 4 floats)	1st-4th coordinates(1,2,3,4 floats)
...	...
Texture Coordinate Set 8 (1,2,3, or 4 floats)	1st-4th coordinates(1,2,3,4 floats)

<고정 기능 파이프라인의 FVF>

가장 많이 사용되는 FVF를 초록색으로 표시해 놓았습니다.

만약 정점의 위치, 정점의 색상을 표현하는 Diffuse, 반사 하이라이트(Highlight) Specular로 정점 구조체를 만들고자 한다면 다음과 같이 작성하면 됩니다.

```
struct VtxDS
{
        D3DXVECTOR3 p;          // 위치
        DWORD   d;              // Diffuse
        DWORD   s;              // Specular
};
```

정점 구조체의 이름은 각자 이름 규칙을 가지고 만들면 되고, 대신 위 그림에서 제시한 순서와 형식대로 구조체를 작성해야 합니다.

이 번에는 텍스처 매핑을 위해 2차원 매핑을 갖는 정점 구조체를 만들어 봅시다. 그림의 Texture Coordinate Set을 보면 1st-4th Coordinate로 되어 있는데 이것은 텍스처가 1차원에서 4차원까지 가질 수 있다는 것이며 floats로 되어 있으므로 2차원일 경우에 float 형 변수 2개를 선언하거나 2차 배열을 만들어야 합니다.
그냥 간단하게 매핑 좌표를 u, v라 한다면 구조체는 앞의 예와 비슷하게 다음과 같이 구성할 수 있습니다.

```
struct VtxUV1
{
        D3DXVECTOR3 p;          // 위치
        FLOAT   u;              // 텍스처 매핑 x좌표
        FLOAT   v;              // 텍스처 매핑 y좌표
};
```

아주 잘 빠진 구조체 입니다.
여기서 깜짝 퀴즈!!! 이 "위치" + "2차원 텍스처 좌표" 구조체와 앞의 "위치"+ "Diffuse" + "Specular"로 되어 있는 구조체를 구분 할 수 있습니까?

사람은 차이를 구분할 수 있지만 컴퓨터는 하지 못합니다. 그냥 20 바이트 공간으로 인식할 뿐입니다. 따라서 이것을 그냥 컴퓨터에 넣어주면 예쁘게 출력 되지 않습니다.
컴퓨터에게 "지금 네게 준 것은 렌더링 세트 2호야 "라고 알려 주어야 합니다. 이것은 앞서 상태머신 설정에서 언급했듯이 디바이스의 SetFVF() 함수로 렌더링 파이프라인에 올라와 있는 정점 데이터의 포맷을 알립니다.

포맷 값은 D3D의 FVF 값을 "|" 비트 연산자로 연결해서 알립니다.
앞의 "위치" + "Diffuse" + "Specular" 구조체에 대한 최종 FVF 값은 다음과 같이 조합합니다.

```
DWORD FVF = (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_SPECULAR);
```

그리고 디바이스의 SetFVF() 함수를 DrawPrimitive()전에 호출합니다.

```
m_pd3dDevice->SetFVF( FVF );
m_pd3dDevice->DrawPrimitive(…);
```

"위치" + "2차원 텍스처 좌표" 로 구성된 구조체의 FVF는 다음과 같이 구성합니다.

```
#define FVF_VTXUV1      (D3DFVF_XYZ|D3DFVF_TEX1)
```

이 전에는 FVF를 변수에 저장했지만 지금은 전처리 문에 넣었습니다. 이 방법이 변수 하나라도 더 줄일 수 있다는 장점이 있습니다.

가장 많이 사용되는 정점 구조체 형식은 "위치" + "법선" + "색상" + "텍스처 1" + "텍스처 2" 입니다. 이것을 구조체와 FVF를 표현 하면 다음과 같이 됩니다.

```
struct VtxNDUV2
{
        D3DXVECTOR3     position;       // 정점 위치
        D3DXVECTOR3     normal;         // 법선 벡터
        D3DXVECTOR2     tex0;           // 0 번 텍스처
        D3DXVECTOR2     tex1;           // 1 번 텍스처

        enum { FVF=(D3DFVF_XYZ| D3DFVF_NORMAL| D3DFVF_DIFFUSE | D3DFVF_TEX2), };
};
```

일단 enum 의 FVF를 보면 예상대로 법선 벡터에 대한 FVF인 D3DFVF_NORMAL 이 포함 되어 있습니다. 그런데 텍스처를 보면 TEX2로만 되어 있습니다. 텍스처에 대한 TEX 다음에 오는 수는 텍스처의 개수입니다. D3D 는 한 정점에 총 8개의 텍스처 좌표를 설정 할 수 있습니다. 따라서 TEX 다음에 오는 최대 수는 8이 됩니다.

이 코드는 구조체 안에서 enum을 선언하고 있습니다. 이 코드가 동작이 되는 것은 sizeof 연산자로 enum이 있는 것과 없는 것의 크기를 비교해 보면 둘이 같음을 알 수 있습니다.
이렇게 enum으로 만들어 놓으면 SetFVF()함수에서 다음과 같이 호출합니다.

```
m_pd3dDevice->SetFVF(VtxNDUV2::FVF);
```

정점 구조체의 크기가 변하지 않음을 이용해 enum 대신 static으로 FVF 변수를 선언하기도 합니다.

```
struct VtxNDUV2
{
…
        static DWORD FVF;
```

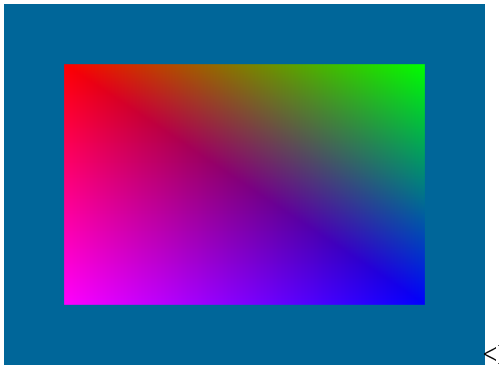
```
};

// 실행 cpp 파일
DWORD VtxNDUV2::FVF = (D3DFVF_XYZ| D3DFVF_NORMAL| D3DFVF_DIFFUSE | D3DFVF_TEX2);
```

static의 약점은 이용되는 변수를 어디선가 초기화 해야 한다는 것입니다. 이런 이유로 enum을 사용하는 것이 좋습니다.

4.1.1 RHW(Use Transformed Vertex)

m3d_04_primitive00__rhw.zip 파일을 압축을 풀고 실행을 하면 다음과 같은 사각형이 화면에 출력됩니다.


<RHW 방식의 폴리곤>

이 그림은 INT CMain::Init() 함수 안에 정점을 생성하는 pd3dDevice->CreateVertexBuffer()와 정점 데이터를 갱신하는 m_pVB->Lock()/Unlock() 함수를 통해서 이 사각형(폴리곤)에서 사용되고 있는 정점 구조체와 FVF는 VtxFmt.h 파일에 있는 다음으로 정의 되어 있음을 알 수 있습니다.

```
struct VtxRHW
{
        D3DXVECTOR4     p;              // The transformed position for the vertex
        DWORD           d;              // The vertex color
...
};

#define FVF_VTXRHW      (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

RHW는 "Reciprocal Homogeneous W"의 약어로 번역을 하면 역(逆: 반대) 동차 W로 이것의 의미는 "이미 변환(Transformed)을 끝내고 w 값으로 나누었습니다." 라는 것을 디바이스에게 알리는 것입

니다. 이렇게 RHW를 지정하면 고정 기능 그래픽 파이프라인에서는 x, y, z 점을 변환하지 않고 또한 w 값으로 나누지도 않으며 곧 바로 화면에 출력을 합니다.
앞의 그림 "<고정 기능 파이프라인의 FVF>" 에서 "transformed"은 사용자가 변환된 데이터로 파이프라인에서 변환 과정을 거치지 않는다는 것입니다.
m3d_04_primitive00__rhw.zip 예제의 어느 부분에서도 뷰 행렬, 투영 행렬을 설정하고 디바이스에 설정하기 위한 "SetTransform()" 함수는 없이 정점의 x, y 위치와 화면의 위치가 동일하게 출력이 되고 있습니다. 즉, VtxRHW 구조체 위치 p의 x, y 좌표는 CMain::Init() 함수에 구현된 다음 코드처럼 화면 좌표를 그대로 사용하고 있음을 알 수 있습니다.

```
VtxRHW pVtx[4];

pVtx[0] = VtxRHW(  100.f, 100.f, 0.f, 1.f, 0xffff0000 );
pVtx[1] = VtxRHW(  700.f, 100.f, 0.f, 1.f, 0xff00ff00 );
pVtx[2] = VtxRHW(  700.f, 500.f, 0.f, 1.f, 0xff0000ff );
pVtx[3] = VtxRHW(  100.f, 500.f, 0.f, 1.f, 0xffff00ff );
...
if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (void**)&pVertices, 0 ) ) )
        return E_FAIL;

memcpy( pVertices, pVtx, m_dwSizeofVertices);
m_pVB->Unlock();
```

RHW를 사용할 때 w 값은 변환 과정에서 행렬 변환 후의 정점 위치(x', y', z', w')를 (x'/w', y'/w', z'/w', w'/w') 와 같이 최종 w = w'/w' = 1로 한다고 정점의 변환에서 설명한 적이 있습니다. 따라서 항상 w= 1.0로 해야 합니다.
z 값은 정규 변환(투영 변환) 과정을 거치면 뷰 체적의 정점은 [0,1] 범위에 있습니다. 이 값 중에서 0을 선택합니다. 만약 여러 RHW를 그리고 순서를 정할 경우 정렬의 위치에 따라 적절한 값을 0보다 크거나 같고 1보다 작은 범위의 숫자를 z 값으로 선택하기 바랍니다.
RHW가 정점 파이프라인을 거치지 않고 통과하는 것을 이용해서 게임 프로그래머는 RHW를 2D의 Sprite 객체를 만드는데 많이 이용합니다. 이 부분은 프리미티브 과정이 끝나면 과제로 하겠습니다.

RHW를 사용하는 구조체는 다음 그림과 같은 여러 형식을 볼 수 있는데 이들은 서로 같습니다.

```
struct VtxRHW
{
    D3DXVECTOR4 p;
    ...
};
```

```
struct VtxRHW
{
    D3DXVECTOR3 p;
    float rhw;
    ...
};
```

```
struct VtxRHW
{
    float x, float y, float z, float rhw;
    ...
};
```

<RHW를 사용한 구조체>

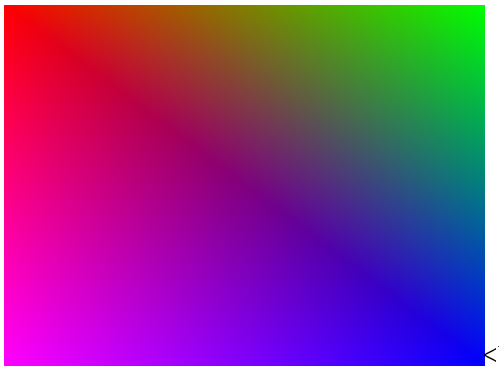
이 이유는 앞서 깜짝 퀴즈에서 언급한 것처럼 디바이스는 FVF 형식에 따라 데이터를 해석하기 때문입니다.

4.1.2 XYZ (Use Untransformed Vertex)

앞의 RHW는 그래픽 파이프라인에서 정점 변환 과정을 건너 뛰며 정점의 위치는 화면 좌표와 일치해 미리 변환이 된 정점을 사용하는 것처럼 되어 "RHW는 Transformed Vertex를 사용한다" 라고 했습니다.

Untransformed Vertex는 변환되지 않은 정점으로 장차 그래픽 파이프라인에서 변환 과정을 거치게 될 정점입니다. Untransformed Vertex는 디바이스의 변환 행렬들을 설정해야 합니다. 그런데 파이프라인의 변환 행렬을 설정하지 않는다면 무슨 일이 일어날까요?

다음 그림은 행렬을 설정하지 않았을 때 파이프라인의 디폴트 값을 가지고 출력된 이미지입니다.


<Untransformed vertex data>

D3D 파이프라인의 변환 행렬은 4x4 행렬입니다. 이 행렬들은 전부 대각선이 1이고 나머지가 0인 항등 행렬(단위 행렬) 입니다. 변환은 정점의 위치에 행렬을 곱해서 새로운 위치를 만드는 것인데 이 때 행렬이 단위 행렬이면 1을 곱한 것과 같아 원래의 위치대로 출력이 됩니다. 원래의 위치가 그대로 출력된다는 것에서 RHW와 비슷하다고 느껴질 것입니다.

그런데 RHW와 분명한 차이는 D3D 파이프라인인의 변환은 투영 변환(정규 변환)의 결과 정점의 위치는 [-1, 1] 범위를 갖는다고 했습니다.

위의 그림처럼 화면 안에 렌더링 하기 위해서는 정점의 위치는 [-1, 1] 범위를 가져야만 합니다.

m3d_04_primitive00_untransformed.zip 예제의 CMain::Init() 함수를 보면 위의 그림을 표현하기

위해서 정점의 위치가 설정되어 있음을 볼 수 있습니다.

```
VtxD pVtxSrc[4];
pVtxSrc[0] = VtxD( -1.f, +1.f, 0.f, 0xffff0000 );
pVtxSrc[1] = VtxD( +1.f, +1.f, 0.f, 0xff00ff00 );
pVtxSrc[2] = VtxD( +1.f, -1.f, 0.f, 0xff0000ff );
pVtxSrc[3] = VtxD( -1.f, -1.f, 0.f, 0xffff00ff );

VtxD* pVertices;

m_pVB->Lock( 0, m_dwSizeofVertices, (void**)&pVertices, 0 );
memcpy( pVertices, pVtxSrc, m_dwSizeofVertices);
m_pVB->Unlock();
```

그래픽 파이프라인의 행렬을 설정하지 않고 출력하는 것은 다음처럼 파이프라인의 행렬을 단위 행렬로 초기화 해서 사용하는 것과 같습니다.

```
D3DXMATRIX      mtTM;
D3DXMatrixIdentity(&mtTM);

m_pd3dDevice->SetTransform(D3DTS_WORLD, &mtTM);
m_pd3dDevice->SetTransform(D3DTS_VIEW , &mtTM);
m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &mtTM);
```

RHW는 화면 좌표로 출력이 되며 RHW가 아닌 Untransformed Vertex는 변환 행렬이 단위 행렬로 설정 되면 [-1, 1] 범위를 가져야 출력이 됨을 꼭 기억 하기 바랍니다.

4.1.3 정점 버퍼와 렌더링

비디오 메모리에 정점 버퍼를 만들기 위해서 우리는 디바이스의 CreateVertex() 함수를 이용해야 합니다.

```
HRESULT hr;
hr = m_pd3dDevice->CreateVertexBuffer( dSizeofVertices,
                        0, VtxD::FVF, D3DPOOL_MANAGED, &m_pVB, 0 );
```

```
if(FAILED( hr ))
        return hr;
```

CreateVertexBuffer() 함수의 첫 번째 인수는 정점 버퍼의 크기 입니다. 정점 버퍼의 크기는 보통 정점 수 * sizeof("정점 구조체") 로 계산합니다.
그 다음 인수는 usage로 usage 선택에 따라 속도 차이가 있을 수 있습니다. 보통은 0으로 설정합니다.
세 번째 인수는 FVF 값으로 사용자가 D3D가 지원하는 FVF를 조합해서 설정합니다.
네 번째 인수는 메모리 풀입니다. DEFAULT 가 속도 면에서 가장 좋아 보이지만 가장 무난한 것은 MANAGED입니다.
네 번째 인수는 생성된 정점 객체의 주소 입니다.
이렇게 디바이스를 통해서 만든 정점 객체를 해제하려면 Release() 함수를 호출 하면 됩니다.
더 이상의 사용을 막기 위해서 NULL로 초기화 하면 더 좋습니다.

```
m_pVB->Release();       m_pVB = NULL;
```

만약 사용하고 있는 버텍스 버퍼의 종류, 크기, FVF 등을 알고 싶으면 IDirect3DVertexBuffer9::GetDesc() 함수를 다음과 같이 사용하면 됩니다.

```
D3DVERTEXBUFFER_DESC desc;
m_pVB->GetDesc(&desc);
```

정점 버퍼의 접근은 앞서 기초 시간에 배웠습니다. 다시 말한다면 Lock()함수로 독점권과 정점 데이터가 저장된 주소를 얻습니다. 이 주소를 이용해서 정점 데이터를 가져오거나 갱신합니다. 작업이 끝나면 독점권을 Unlock() 함수를 통해서 반환 합니다. 그리고 Lock() 함수로 얻은 주소는 Unlock()함수 이후에는 더 이상 유효하지 않는 주소라는 점을 기억하기 바랍니다.
다음은 정점 데이터의 정보를 변경하는 예제 입니다. memcpy() 함수 인수의 순서를 바꾸면 데이터를 가져오는 코드가 됩니다.

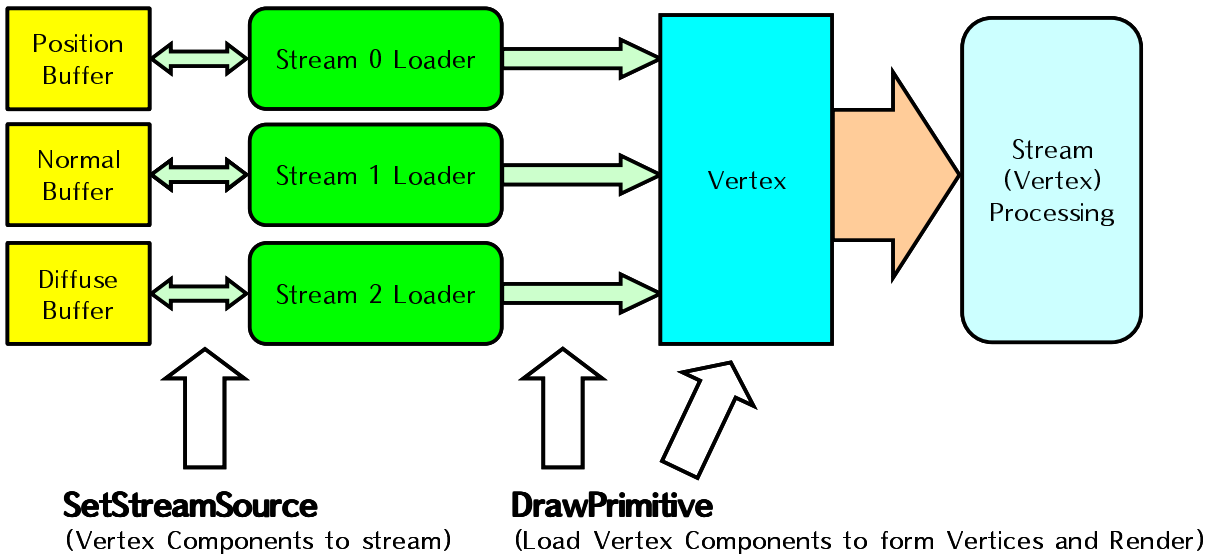
```
VtxD* pVertices;

if( FAILED( m_pVB->Lock( 0, m_dwSizeofVertices, (void**)&pVertices, 0 ) ) )
        return -1;

memcpy( pVertices, pVtxSrc, m_dwSizeofVertices);
m_pVB->Unlock();
```

정점 버퍼를 만들고 정점 데이터를 채웠다면 렌더링만 남아 있습니다. 렌더링은 정점 버퍼 연결, FVF 알림, 렌더링 명령 호출 순서로 됩니다. 상태 머신의 설정 순서는 바뀌어도 상관은 없으나 렌더링 명령 호출을 반드시 맨 나중에 해야 합니다.

파이프라인에 정점 버퍼 연결은 SetStreamSource() 함수를 이용합니다.
D3D는 정점 데이터의 위치, 법선, 색상, 텍스처 좌표 등 그 내용들을 따로 분리해서 만들 수 있습니다. 또한 분리해서 만든 데이터를 합쳐서 사용할 수 있습니다. 이렇게 분리된 데이터를 합쳐서 사용할 수 있도록 파이프라인에 해당 정보를 연결해 주는 것이 SetStreamSource() 함수 입니다.



<SetStreamSource() 함수의 역할>

이렇게 분리해서 사용하면 메모리를 절약할 수 있는 이점이 있습니다. 하지만 이런 방식을 지원하는 그래픽 카드는 천차만별입니다. 아예 분리가 안 되는 카드들도 있어서 또한 굳이 분리할 필요를 느끼지 않아 다음과 같이 하나의 단일 스트림만 사용하는 것이 대부분 입니다.

```
m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VtxD) );
```

디바이스에 FVF 알림은 SetFVF() 함수로 이것은 이전에도 설명했습니다.

```
m_pd3dDevice->SetFVF( VtxD::FVF );
```

마지막에 프리미티브 정보와 렌더링 명령입니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```

렌더링 순서와 필요한 함수들 간략히 정리해 봅니다.

1. 정점 구조체, FVF 선언
2. 정점 버퍼 생성: pDevice->CreateVertexBuffer(크기, 0, FVF, "메모리 풀", &"정점 버퍼")
3. 정점 버퍼의 갱신: "정점 버퍼"->Lock()/Unlock()
4. 디바이스에 스트림 연결: pDevice->SetStreamSource("정점 버퍼")
5. FVF 알림: pDevice->SetFVF(FVF)
6. 프리미티브 정보 전달과 렌더링 명령: pdDevice->DrawPrimitive("프리미티브 정보", …)

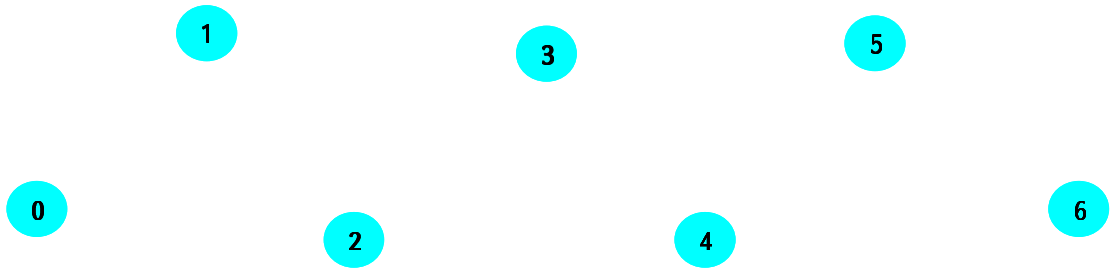
4.2 Primitive

렌더링의 최소 단위는 프리미티브(Primitive)입니다. D3D3D는 정점을 조합해서 프리미티브를 크게 점(Point), 선(Line), 삼각형(Triangle) 세 종류로 만듭니다. 각각의 프리미티브는 렌더링 방법을 도입해서 리스트(List), 스트립(Strip), 팬(Fan)으로 분리합니다.
점은 리스트만 가능해 포인트 리스트 프리미티브만 있습니다. 라인은 리스트, 스트립 두 방식의 렌더링이 가능해서 라인 리스트 프리미티브, 라인 스트립 프리미티브 두 가지가 있습니다.
삼각형은 삼각형 리스트, 삼각형 스트립, 삼각형 팬 총 세 가지를 가지게 됩니다. 이들 총 6가지가 D3D가 지원하는 프리미티브이며 OpenGL은 이 보다 더 많은 프리미티브를 지원합니다.
구체적으로 각각의 프리미티브를 자세히 알아 봅시다.

4.2.1. 포인트 리스트(Point Lists)

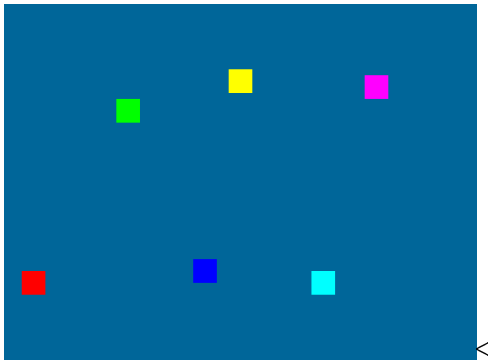
점에 대한 렌더링은 점 자체를 출력하는 방식 하나밖에 없습니다. 따라서 점에는 포인트 리스트만 존재 합니다.

여러분이 정점 버퍼에 7개의 정점을 만들었고 포인트 리스트 방식으로 렌더링 명령을 내리면 D3D는 이 정점의 순서에 따라 다음과 같은 방식으로 정점을 출력합니다.



<포인트 리스트 프리미티브>

m3d_04_primitive01_PointList.zip는 포인트 리스트 예제 입니다.
이 압축 파일을 컴파일 하고 실행 하면 다음 그림과 같이 화면에 총 6개의 정점을 출력합니다.


<포인트 리스트>

m3d_04_primitive01_PointList.zip을 컴파일 하고 실행하면 위와 같은 점 6개가 출력되는 화면을 볼 수 있습니다.

다른 프리미티브는 정점 버퍼 이외에 시스템 메모리의 힙 공간, 스택 공간의 정점도 출력이 가능합니다. 하지만 포인트 리스트 프리미티브는 오직 디바이스로 만든 정점 버퍼로만 렌더링이 가능합니다. 이 부분이 포인트 리스트의 가장 중요한 부분입니다.

```
hr = m_pd3dDevice->CreateVertexBuffer(  m_dwSizeVertices
                                      , 0     // D3DUSAGE_POINTS
                                      , FVF_VTXDRHW, D3DPOOL_MANAGED, &m_pVB, 0 );
```

이렇게 만든 정점 버퍼를 렌더링 하면 점의 크기가 작아 화면에 출력이 안 되는 것처럼 보일 수 있습니다. 이럴 때 점의 크기를 키워야 합니다. 점 크기의 변경은 디바이스의 렌더링 상태 함수로 설정합니다.

```
float    fPointSize = 40.F;
m_pd3dDevice ->SetRenderState( D3DRS_POINTSIZE, *((DWORD*)&fPointSize));
```

SetRenderState()함수는 DWORD형 이외의 변수를 그냥 전달하면 데이터를 잃어 버립니다. 이를 위해 *((DWORD*)&("변수"))와 같은 캐스팅이 필요합니다.
40 픽셀이면 상당히 큰 값이라 화면에 잘 출력이 될 것입니다.

포인트 리스트는 파티클(입자) 효과 등에서 자주 사용됩니다. 보통 입자 하나를 만들기 위해서 2개의 삼각형이 필요하고 정점 4개가 들지만 포인트 리스트는 1개만 있어도 됩니다. 따라서 메모리를 많이 사용하는 파티클 효과에서 아주 유용한 방식이 됩니다.

프리미티브 나머지를 끝내고 DXSDK의 파티클 예제인 Point Sprite 예제도 꼭 실행해 보기 바랍니다.

이 프리미티브를 파이프라인에 알리는 방법은 DrawPrimitive() 함수 첫 번째 인수에 다음과 같이 합니다.

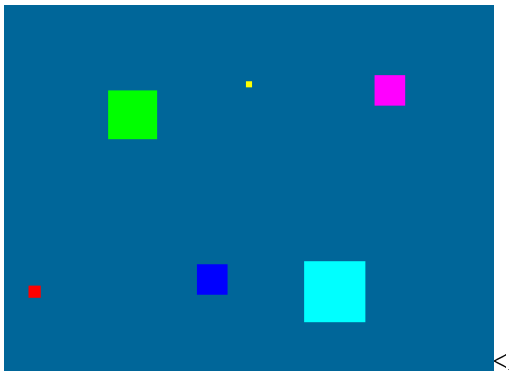
```
m_pd3dDevice->DrawPrimitive( D3DPT_POINTLIST, …);
```

그림 <고정 기능 파이프라인의 FVF>을 보면 정점 구조체에 Vertex Point Size가 있고 float형으로 되어 있습니다.
이 내용을 가지고 앞의 예제의 구조체와 FVF를 다음과 같이 변경한 후에

```
struct VtxRHWD
{
        D3DXVECTOR4     p;
        FLOAT           s;              // Point Size
        DWORD           d;
…
};
#define FVF_VTXDRHW     (D3DFVF_XYZRHW|D3DFVF_PSIZE|D3DFVF_DIFFUSE)
```

CMain::Init() 함수의 정점 갱신을 다음과 같이 하면

```
VtxRHWD pVtx[6];
pVtx[0] = VtxRHWD(  50.f,  470.f, (1+rand()%10)*10.f,  0xFFFF0000 );
…
pVtx[5] = VtxRHWD( 630.f,  140.f, (1+rand()%10)*10.f,  0xFFFF00FF );
```

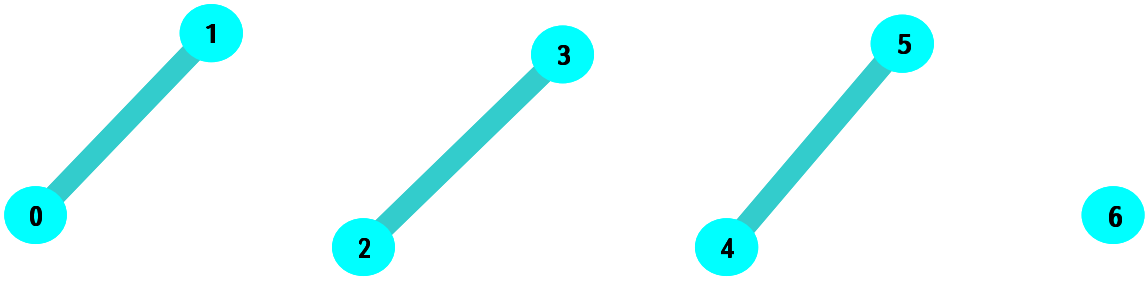

<포인트 크기가 들어간 정점 구조체 >

화면에 6개 정점의 크기가 달라져 있음을 볼 수 있습니다.
이것은 구조체에서 정점 크기에 대한 정보 없으면 디바이스에 설정된 크기 값으로 정점이 출력됩니다.

이외에 텍스처를 붙여서 출력하는 포인트 스프라이트(Point Sprite)도 있으나 지금은 프리미티브의 종류를 배우기 때문에 이 부분은 생략하겠습니다. 이전 그림의 코드는 m3d_04_primitive01_PointList2.zip을 참고 하기 바랍니다.

4.2.2 라인 리스트(Line Lists)

라인 리스트 프리미티브는 두 개의 정점을 하나의 쌍으로 설정해서 라인을 그리는 방식입니다.
만약 총 7개의 정점의 다음그림과 같이 있다면 실제 화면에는 3개의 선이 그려집니다.



<라인 리스트 프리미티브>

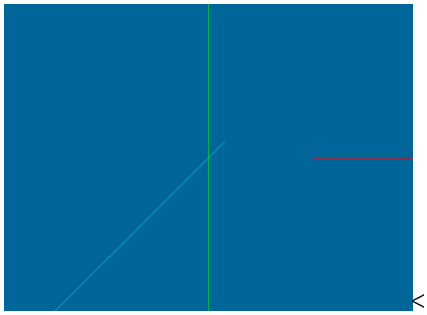
이 n개의 프리미티브에 대한 정점의 수는 프리미티브에 2배가 필요합니다.

필요한 정점의 수 = 프리미티브 수 * 2

라인 리스트 프리미티브는 정점 리스트와 마찬가지로 디바이스의 DrawPrimitive() 함수의 첫 번째 인수에 설정합니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_LINELIST, 정점 시작 인덱스, 프리미티브의 개수);
```

m3d_04_primitive01_LineList.zip 을 실행하면 x, y, z축을 선을 그리고 있는 위와 같은 화면이 출력됩니다.
여러분들도 라인 리스트를 이용해서 각각 축을 그려 넣는 것이 좋습니다. 어떤 프로그래머는 x, z 축 평면에 바둑판처럼 일정한 폭의 선을 그려 넣어서 게임 오브젝트의 위치와 방향을 눈으로 확인하는 분들도 있습니다.
최소한 3개의 축만 각각 다른 색으로 표시해도 렌더링이 잘 되고 있는 지 파악 될 수 있습니다.


<라인 리스트 프리미티브>

선의 색상을 출력하기 위해 정점 구조체와 FVF 를 다음과 같이 설정합니다.

```
struct VtxD
{
        D3DXVECTOR3     p;      // 정점 위치
        DWORD           d;      // 정점의 색상
        …
        enum {FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE), };
};
```

렌더링에서 주의할 점은 정점의 색상을 출력할 때 조명 효과가 활성화 되면 색상이 검정색으로 나올 수 있습니다. 이것은 D3D가 조명 효과를 디폴트로 활성화 해 놓았는데 만들어진 정점은 법선 값이 없고, 조명 또한 설정을 안 했기 때문입니다. 따라서 다음과 같이 조명을 꺼 놓으면 정점의 원래 색상이 출력 됩니다.

```
pDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

다음으로 먼저 오브젝트들이 화면에 출력이 되야 하니까 은면 제거를 처음에는 비활성화 시킵니다.

```
m_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
```

지금 화면에 출력하는 정점에 텍스처가 필요 없는 경우 파이프라인에게 텍스처 포인터가 없다고 무조건 알리는 것이 좋습니다.

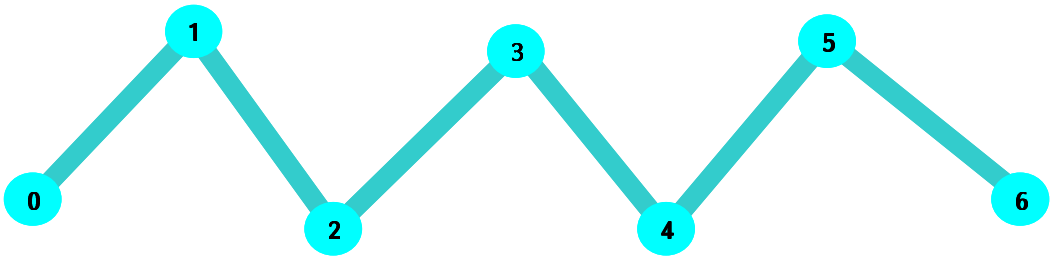
```
m_pd3dDevice->SetTexture( 0, NULL);     // 텍스처 사용을 안 함
```

실습 과제) x, y, z 축에 대해서 길이 4000에 대한 선을 각각 붉은 색, 초록 색, 파란 색으로 출력하시오.

실습 과제) x-z 평면에 10 픽셀 간격으로 -100~ 100 범위를 바둑판 모양의 그리드(Grid)를 출력하시오.

4.2.3 라인 스트립(Line Strip)

때로는 다음 그림과 같이 직선들이 정점을 연속으로 연결 된 형태도 있을 수 있습니다. 이런 구조라면 리스트 방식보다 선을 그리는데 필요한 정점의 숫자는 절반 정도 될 것입니다.



<라인 스트립>

라인 스트립은 연속된 정점을 이어서 그리기 때문에 수학 함수의 선들을 표시하거나 운동의 예상 경로를 표시하는데 사용 되기도 합니다.
이 n개의 프리미티브에 대한 정점의 수는 프리미티브 + 1 이 됩니다.

필요한 정점의 수 = 프리미티브 수 + 1

5개의 직선이 연결되어 있다면 리스 방식으로 그릴 때 정점은 10개가 필요하다. 그런데 연속된 직선의 경우 다음 직선의 시작점은 이전 직선의 끝나는 지점과 일치를 한다. 이러한 경우 직선이 계속 이어지는 상황에서 메모리를 줄여주는 방식이 라인 스트립 방식이다.

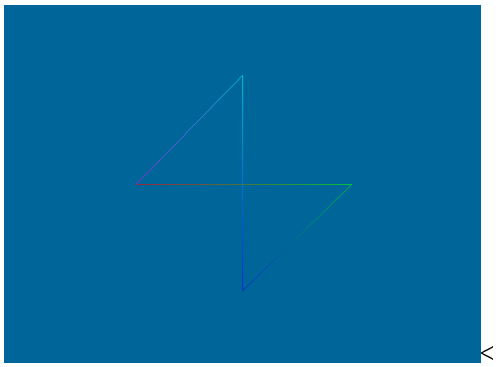
pDevice->DrawPrimitive(D3DPT_LINESTRIP, 정점 시작 인덱스, 프리미티브의 개수);

Line Strip 방식으로 라인을 그리면 그려야 할 라인과 필요한 정점의 수는 다음과 같다.

필요한 정점의 수 = 프리미티브 수 +1

m3d_04_primitive02_LineStrip.zip 을 실행하면 그림처럼 4개의 선들이 출력 되는데 라인 리스트와 차이는 4개의 선을 표현하기 위해서 5개의 정점을 사용했고, DrawPrimitive() 함수에 라인 스트립 옵션을 넣은 것입니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_LINESTRIP, 0, 4);
```

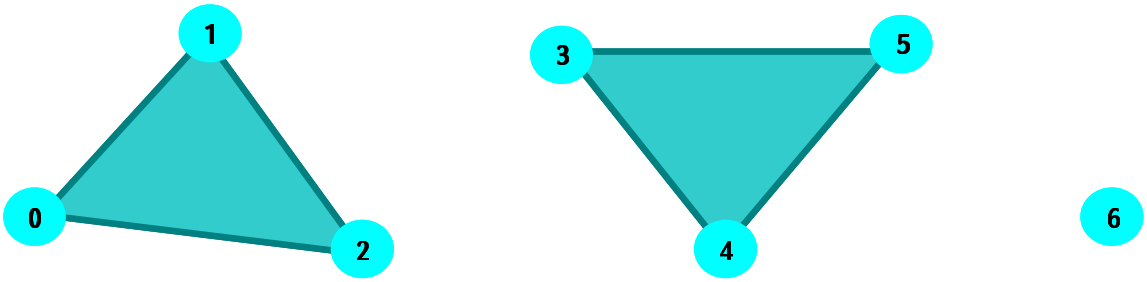

<라인 스트립 프리미티브>

4.2.4 Triangle List

점, 선 두 종류의 도형에서 우리는 보통 상식으로 렌더링 하는 방법은 리스트(List)라는 것을 알게 되었습니다. 삼각형도 마찬가지 입니다.
가장 일반적으로 3점을 하나의 삼각형으로 구성해서 렌더링 하는 것을 삼각형 리스트(Triangle List)라 합니다. 삼각형들(보통 폴리곤이라 합니다.)을 그릴 때 필요한 정점의 수를 계산하면 삼각형 개수 * 3이 됩니다.

필요한 정점의 수 = 프리미티브 수 * 3

삼각형 리스트는 패턴이 없는 일반적인 폴리곤에서 가장 많이 사용이 됩니다.
예를 들어 파티클 이펙트(Particle Effect)를 삼각형으로 표현한다면 이 삼각형들은 연속이 아닌 서로 떨어져 있어서 삼각형 리스트로 그려야 합니다. 또한 렌더링 오브젝트가 삼각형을 많이 가지고 있지 않은 경우에도 종종 이 프리미티브가 사용됩니다.



<삼각형 리스트>

```
pDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 정점 시작 인덱스, 프리미티브의 개수);
```

색상을 가지는 두 개의 삼각형을 출력해 보겠습니다.
먼저 정점 구조체를 3차원 위치 벡터와 Diffuse 색상에 대한 DWORD 두 개로 구성하는 작업은 많이 해왔기 때문에 아주 쉽습니다.

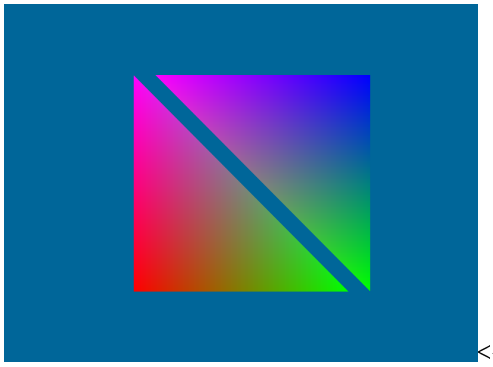
```
struct VtxD
{
        D3DXVECTOR3     p;
        DWORD           d;              // Diffuse Color
        …
        enum { FVF = (D3DFVF_XYZ|D3DFVF_DIFFUSE),};
};
```

삼각형 2개를 렌더링 하므로 정점 생성은 6개를 합니다.

```
m_dwSizeVertices = sizeof(VtxD) * 6;
m_pd3dDevice->CreateVertexBuffer(m_dwSizeVertices, …);
```

렌더링에서 삼각형 리스트, 프리미티브 2개를 출력하도록 명령을 내립니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2);
```


<삼각형 리스트: m3d_04_primitive04_List.zip>

위의 삼각형 리스트에 텍스처를 매핑 해 보겠습니다. 텍스처 매핑에 대한 자세한 내용은 다음에 나오니 여기서는 매핑 좌표만 설정하고 렌더링 정도만 시도하겠습니다.
텍스처 매핑을 위해서 구조체를 수정해야 합니다. 지금까지 사용한 색상은 버리고 대신 텍스처 좌표 설정할 수 있도록 다음과 같이 구조체를 수정합니다.

```
struct VtxUV1
```

```
{
        D3DXVECTOR3     p;
        FLOAT           u, v;           // 텍스처 매핑 2차원 좌표
        enum { FVF =(D3DFVF_XYZ|D3DFVF_TEX1), };
};
```

매핑 좌표를 좌 하단은 (0, 1), 좌 상단은 (0, 0), 우 상단은 (1, 0), 우 하단은 (1, 1)로 설정하겠습니다.

```
pVtx[0] = VtxUV1( -50.f, -50.f, 0.f,     0.f, 1.f );
pVtx[1] = VtxUV1( -50.f,  50.f, 0.f,     0.f, 0.f );
pVtx[2] = VtxUV1(  50.f, -50.f, 0.f,     1.f, 1.f );

pVtx[3] = VtxUV1(  60.f,  50.f, 0.f,     1.f, 0.f );
pVtx[4] = VtxUV1(  60.f, -50.f, 0.f,     1.f, 1.f );
pVtx[5] = VtxUV1( -40.f,  50.f, 0.f,     0.f, 0.f );
```

이렇게 매핑은 끝냈습니다. 다음으로 텍스처를 생성합니다.

```
LPDIRECT3DTEXTURE9  m_pTx;
...
D3DXCreateTextureFromFile(m_pd3dDevice, "Texture/env3.bmp", &m_pTx);
```

렌더링에서 디바이스의 SetTexture()함수로 연결합니다.

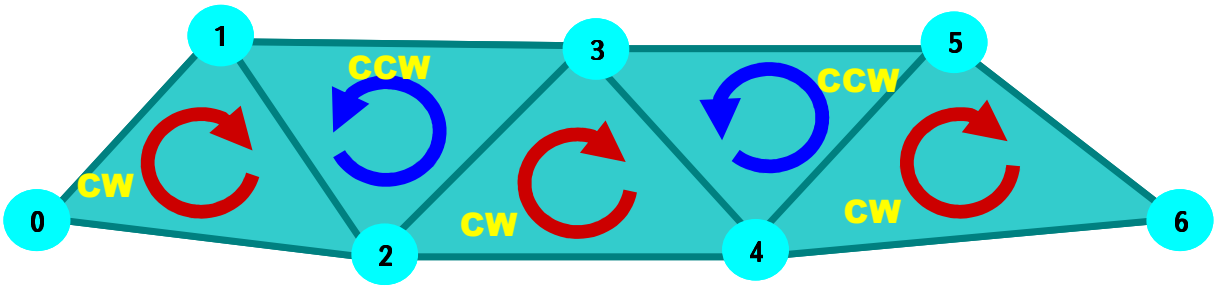
```
m_pd3dDevice->SetTexture( 0, m_pTx);
m_pd3dDevice->SetStreamSource( 0, m_pVB, 0, sizeof(VtxUV1) );
m_pd3dDevice->SetFVF( VtxUV1::FVF );
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2);
m_pd3dDevice->SetTexture( 0, NULL);
```

텍스처 사용이 끝나면 SetTexture(0, NULL)로 디바이스의 텍스처 포인터를 해제합니다.

< 삼각형 리스트: m3d_04_primitive04_List2.zip>

4.2.5 Triangle Strip

라인 스트립과 비슷하게 삼각형 스트립(Triangle Strip)도 연속된 삼각형을 표현하는 방법입니다. 그런데 삼각형 스트립은 다음 그림처럼 인접한 삼각형끼리 공유 되는 점들의 감는 방법입니다.



<삼각형 스트립>

정점을 스트립 구조에 맞게 구성하려면 최초의 삼각형에 대한 정점은 시계 방향으로 감아 쥐는 방향으로 정점을 배치해야 합니다.(0, 1, 2)
두 번째 삼각형은 첫 번째 삼각형의 두 번째 정점을 시작으로 시계 반대 방향으로 감아 쥐면서 정점을 추가해 삼각형을 만듭니다. 마치 톱니바퀴처럼 감아 쥐는 순서를 반복해서 정점을 배치하면 삼각형 스트립이 됩니다.
위 그림에 대한 각각의 삼각형은 v0->v1->v2, v1->v2->v3, v2->v3->v4, v4->v5->v6로 구성이 됩니다.

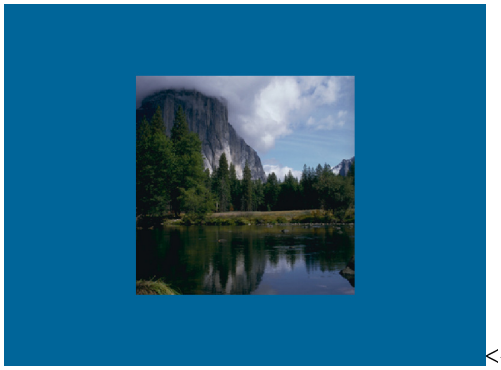
삼각형 스트립에 필요한 정점의 수를 계산 해 봅시다. 위 그림으로 유추해 볼 때 다음과 같이 결정이 된다.

필요한 정점의 수 = 프리미티브 수 + 2

이것을 삼각형 리스트와 비교해 보면 정점의 수가 1/3로 줄어들고 있습니다. 정점이 1/3로 줄어든

다는 것은 굉장한 일입니다. 2000개의 삼각형에 위치, 법선, 색상, 이 차원 텍스처 좌표 구조를 가진 3D 모델을 생성한다고 할 때 삼각형 스트립은 대략 2000*(3 * sizeof(float) + 3* sizeof(float) + sizeof(DWORD) + 2 * sizeof(float)) = 72kb 정도가 필요합니다. 이 모델을 500개 정도 화면에 출력한다면 대략 36Mb가 필요합니다. 그런데 삼각형 리스트로 하면 3배정도가 더 소모가 되므로 100Mb가 넘어 가게 됩니다.
스트립이 메모리를 적게 사용하고 적게 사용하는 만큼 그만큼 렌더링 속도는 증가합니다.

삼각형 스트립의 감아 쥐는 순서가 익숙하지 않을 수 있습니다. 어느 정도 익숙해질 때까지 스트립 방식으로 모델 구성 연습을 틈틈이 하기 바랍니다. 다은 삼각형 스트립을 이용해서 사각형을 만든 그림입니다.


<삼각형 스트립: m3d_04_primitive05_Strip.zip>

m3d_04_primitive05_Strip.zip의 INT CMain::Init() 함수에서 정점의 설정을 보면 4개의 정점에 다음과 같이 좌 하단-> 우 하단 ->좌 상단 -> 우 상단 순으로 삼각형을 구성하고 있음을 볼 수 있습니다.

```
pVtx[0] = VtxUV1( -50.f, -50.f, 0.f,     0.f, 1.f );
pVtx[1] = VtxUV1( -50.f,  50.f, 0.f,     0.f, 0.f );
pVtx[2] = VtxUV1(  50.f, -50.f, 0.f,     1.f, 1.f );
pVtx[3] = VtxUV1(  50.f,  50.f, 0.f,     1.f, 0.f );
```

나머지는 삼각형 리스트를 그릴 때와 같고 마지막 DrawPrimitive() 함수 호출 부분입니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2);
```

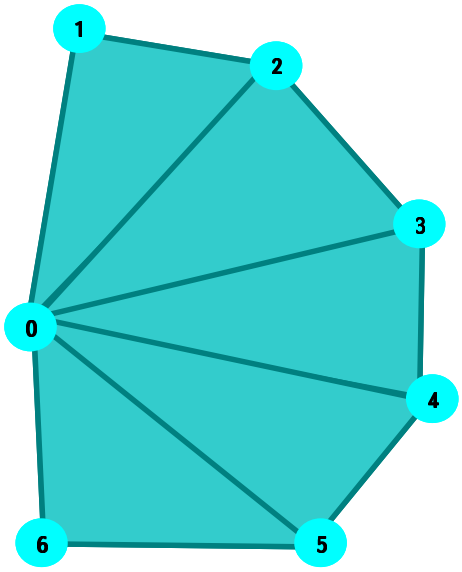
삼각형으로 장면을 연출해도 이 삼각형들을 선으로 출력할 수 있습니다. D3D의 Fill Mode를 다음과 활용하면 간편하게 선으로 삼각형을 출력합니다.

```
if(GetAsyncKeyState('W') & 0x8000)
```

```
        m_pd3dDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
else
        m_pd3dDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
```

4.2.6 Triangle Fan

삼각형에는 팬(Fan)이라는 프리미티브 하나가 더 이 있습니다. 다음 그림처럼 부채 살 모양으로 삼각형이 구성될 때 이 프리미티브를 이용합니다.



<삼각형 팬 (Triangle Fan)>

렌더링 순서는 1->2->0, 2->3->0, 3->4->0, 4->5->0, 5->6->0으로 합니다.
이 삼각형 팬 구조의 정점 수를 계산해 봅시다.

필요한 정점의 수 = 프리미티브 수 + 2

정점의 수는 스트립과 동일합니다.
이 프리미티브는 게임에서 파티클을 대신하는 원뿔 모양의 이펙트나 폭발할 때 충격파 효과, 하늘을 원추형으로 만들 때 많이 이용됩니다. 사각형 하나를 표현할 때도 삼각형 스트립보다 삼각형 팬이 적용하기가 쉽고 실수할 확률이 낮습니다.

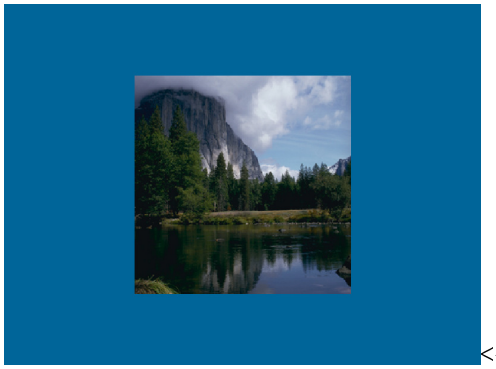
m3d_04_primitive06_Fan.zip 예제의 CMain::Init() 함수에서 정점 설정하는 부분을 보면 스트립 때보다 설정하기 쉽다는 것을 알 수 있습니다.

```
pVtx[0] = VtxUV1( -50.f, -50.f, 0.f,     0.f, 1.f );
pVtx[1] = VtxUV1( -50.f,  50.f, 0.f,     0.f, 0.f );
```

```
pVtx[2] = VtxUV1(  50.f,  50.f, 0.f,     1.f, 0.f );
pVtx[3] = VtxUV1(  50.f, -50.f, 0.f,     1.f, 1.f );
```

DrawPrimitive() 함수에서 삼각형 팬으로 그리도록 다음과 같이 호출합니다.

```
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 2 );
```


<삼각형 팬: m3d_04_primitive06_Fan.zip>

4.3 인덱스 버퍼(Index Buffer)

지금까지 렌더링에 관련된 프리미티브를 전부 살펴 보았습니다. 여러 프리미티브 중에서 가장 많이 사용되는 것은 아마도 삼각형 리스트 입니다.
그런데 오브젝트를 어떤 수단을 발휘해서 삼각형 스트립이나 또는 팬 방식으로 그릴 수 있도록 한다면 리스트보다 1/3 정도 메모리를 줄일 수 있지만 이렇게 만들기가 쉬운 일은 아닐 것입니다.
렌더링을 해야 할 대부분의 오브젝트는 스트립과, 팬과 같은 패턴이 없습니다. 이것을 조금이라도 극복해 주는 것이 인덱스 리스트(Index List) 입니다.

인덱스 리스트는 프리미티브를 구성하는 정점에 대한 인덱스를 리스트로 가지고 있는 객체 입니다. 즉, 이 인덱스는 정점에 대한 번호 리스트라 할 수 있습니다.

예를 들어 앞의 삼각형 스트립에서 사용했던 폴리곤을 인덱스로 구성한다면 다음과 같이 만들 수 있습니다.

v1->v2->v3 ➔ index(1,2,3)
v2->v3->v4 ➔ index(2,3,4)
v3->v4->v5 ➔ index(3,4,5)
v4->v5->v6 ➔ index(4,5 6)

삼각형 팬에서 사용했던 폴리곤도 인덱스로 구성하면 다음과 같이 만들 수 있습니다.

v0->v1->v2 ➔ index(1,2,0)
v0->v2->v3 ➔ index(2,3,0)
v0->v3->v4 ➔ index(3,4,0)
v0->v4->v5 ➔ index(4,5,0)
v0->v6->v6 ➔ index(5,6,0)
인덱스가 있으면 삼각형을 구성할 때 필요한 정점의 수는 다음과 같이 됩니다.

필요한 정점의 수 ~ 프리미티브 수 + alpha

이것은 거의 스트립과 팬에서 구한 정점의 수와 비슷합니다.
또한 인덱스도 기억공간을 차지하므로 이것도 개수를 알고 있어야 합니다. 리스트의 개수는 다음과 같이 됩니다.

필요한 인덱스 수 = 프리미티브 수 * 3

여기에 인덱스가 기억 공간을 적게 사용하도록 하기 위해서 이 인덱스들을 WORD (2Byte)로 만들면 삼각형 하나당 6Byte가 필요하게 됩니다.

앞에서 2000개를 가진 삼각형이 위치, 법선, 색상, 이 차원 텍스처 좌표 구조로 된 삼각형 2000개를 가진 모델을 계산해 보았습니다.
하나의 정점만 하더라도 총 36Byte가 필요합니다. 삼각형 3개라면 108Byte가 소모 됩니다. 하지만 인덱스는 삼각형 1개당 6Byte만 필요합니다. 즉, 삼각형 리스트를 직접 렌더링 하지 않고 인덱스를 구성해서 그리게 되면 정점의 숫자를 줄일 수 있고, 줄어든 정점의 숫자만큼 렌더링은 빨라지리라는 것은 당연할 것입니다.

인덱스를 구성해서 렌더링 하는 인덱스 리스트 방식의 속도는 거의 Strip에 가깝다고 실험적으로 보고되고 있습니다. 그리고 스트립이 표현 못하는 대부분의 오브젝트에 적용이 되는데, 특히 3DS Max의 경우 오브젝트는 인덱스와 정점들로 구성되어 있어서 이 방식으로 렌더링 해야 합니다.

인덱스 리스트 방식으로 렌더링 하기 위해 필요한 것은 이 인덱스를 저장할 공간입니다. 이것을 D3D는 인덱스 버퍼라 합니다.
인덱스 버퍼 객체를 저장할 변수는 다음과 같이 선언합니다.

```
LPDIRECT3DINDEXBUFFER9  m_pIB;
```

인덱스 버퍼도 버텍스 버퍼처럼 디바이스를 통해서 다음과 같은 형식으로 만듭니다.

```
인덱스 크기 = (3 * sizeof(WORD) ) * 프리미티브 수;

pDevice->CreateIndexBuffer( 인덱스 크기
        , 0
        , D3DFMT_INDEX16
        , D3DPOOL_MANAGED, &m_pIB, 0 );
```

여기서 D3DFMT_INDEX16은 인덱스가 16비트 WORD형일 때 사용하고 D3DFMT_INDEX32는 32비트 DWORD 일 때 사용합니다.
16비트는 정점의 최대 인덱스가 65535 까지라는 것이고 32비트는 2의 32 -1 까지 인덱스를 사용할 수 있다는 것입니다. 2의 32 - 1까지 인덱스를 쓸 수 있다고 해서 D3DFMT_INDEX32을 잘 사용하지 않습니다. 왜냐하면 모델을 그래픽 툴을 통해서 작업을 해야 하는데 16비트 보다 커지면 분리해서 작업하는 것이 더 효율이 좋기 때문에 아직까지 D3DFMT_INDEX16을 많이 사용하고 입니다.

이렇게 인덱스 버퍼를 만들고 다음으로 인덱스 버퍼를 접근해서 내용을 채우거나 가져와야 합니다. 이 것은 정점 버퍼와 비슷합니다. Lock() 함수를 통해서 독점권과 인덱스 데이터가 저장된 주소를 얻어와서 이 주소를 가지고 필요한 작업을 한 다음 Unlock으로 접근해 대한 권한을 반납합니다.

```
WORD dwIndices[] = {0, 1, 2, 3, 2, 1};

WORD* pIndices;
if( FAILED( m_pIB->Lock( 0, 0, (void**)&pIndices, 0 ) ) )
   return -1;

// 인덱스 버퍼 값을 갱신한다.
memcpy( pIndices, dwIndices, dSizeIdx );

m_pIB->Unlock();
```

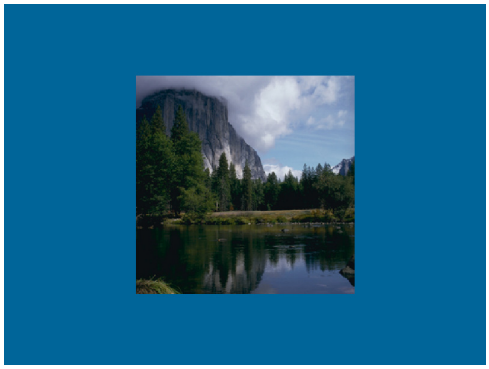
이렇게 인덱스 버퍼를 만든 다음 렌더링은 렌더링에서 다음과 같이 합니다. 먼저 인덱스 버퍼를 파이프라인에 연결합니다.

```
pDevice->SetIndices(m_pIB);
```

다음으로 DrawIndexedPrimitive()함수를 호출 합니다.

```
pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST
                               , 0 // Base Vertex Index
                               , 0 // Minimum vertex Index
                               , 4 // number of vertices
                               , 0 // Start Index
                               , 2 // number of primitives
                               );
```

주의할 것은 인덱스 리스트로 렌더링 할 때 프리미티브의 첫 번째 값은 항상 D3DPT_TRIANGLELIST 입니다.



인덱스 버퍼를 이용해서 그린 삼각형:
m3d_04_primitive07_Indexed.zip

때로는 정점 구조체처럼 인덱스 구조체를 다음과 같이 만들어서 사용하기도 합니다.

```
struct VtxIdx
{
        WORD a, b, c;
        ...
        enum { IDX = D3DFMT_INDEX16, };
};
```

이 구조체를 사용하면 인덱스 버퍼를 만들 때 약간 편리합니다.

```
INT             nTriangle = 2;
if( FAILED( m_pd3dDevice->CreateIndexBuffer( sizeof(VtxIdx) * nTriangle, 0
                                        , (D3DFORMAT)VtxIdx::FMT
                                        , D3DPOOL_MANAGED, &m_pIB, 0 ) ) )
```

```
        return -1;

VtxIdx  idx[2];
idx[0] = VtxIdx(0, 1, 2);
idx[1] = VtxIdx(3, 2, 1);

WORD* pIndices;
if( FAILED( m_pIB->Lock( 0, 0, (void**)&pIndices, 0 ) ) )
   return -1;

memcpy( pIndices, dwIndices, sizeof(VtxIdx) * nTriangle );
m_pIB->Unlock();
```

전체 코드는 m3d_04_primitive07_Indexed.zip 을 참고 하기 바랍니다.

4.4 User memory pointer (UP)

지금까지 우리는 정점 버퍼와 인덱스 버퍼를 이용했습니다. 이렇게 디바이스를 이용해 만든 버퍼뿐만 아니라 사용자가 전역, 지역, heap 등의 시스템 메모리에 할당한 연속 메모리 공간을 정점 버퍼로 사용할 수 있습니다. 이 때 이 공간을 UP(User Memory Pointer: 읽을 때 "유피"라 읽습니다.)라 합니다.

UP를 사용하면 시스템 메모리를 사용하게 되어 비디오 메모리만 이용하는 것보다 많은 메모리 공간을 이용하는 장점이 있습니다. 단점은 렌더링 할 때 시스템 메모리를 버스를 통해서 비디오 카드로 전송해야 합니다. 이 때 성능의 차이가 비디오 카드가 아닌 시스템 자체의 성능에 영향을 받는 다는 것입니다.
과거에는 UP와 정점 버퍼를 사용해서 렌더링 하는 것이 확실이 차이가 많이 났었지만 지금은 시스템이 많이 좋아져 둘의 속도 차이는 거의 없습니다.
정점 버퍼의 메모리를 접근하는 경우 시스템 복사가 자주 일어나고 이것은 버스를 통해서 데이터가 전송되는 경우이므로 차라리 UP를 이용하는 것이 더 나을 수도 있습니다.

UP에 대한 제약은 단일 스트림 밖에 지원을 안 한다는 것입니다. 이것은 어디까지나 제 추측인데 스트림을 하나밖에 안 갖는 이유는 D3D 자체의 문제라기 보다 컴퓨터 구조상의 문제와 관련 있다고 생각합니다. 비디오 카드의 메모리는 GPU의 사양에 맞게 여러 개의 스트림을 만들 수 있지만 시스템 메모리에서 전송되는 데이터는 데이터 버스를 이용해야 하고 이 버스의 제약에 따라 단일 스트림을 가질 수밖에 없지 않나 하는 것입니다.

앞서 D3D가 여러 버퍼의 정점을 모아서 처리할 수 있다고 했지만 대부분 그냥 정점 버퍼 하나를 사용한다고 했습니다. 이러한 경우라면 정점과 같은 데이터는 UP를 이용하고 텍스처 또는 DEFAULT 메모리 풀이 필요한 객체만 D3D를 이용하는 것입니다. 이렇게 UP를 이용한다면 좀 더 많은 컴퓨터에서 프로그램이 동작할 확률이 높고 제가 자주 사용하는 방법입니다.

m3d_04_primitive08_UP_List.zip파일 열면 CMain 클래스의 멤버 변수 m_pVtx는 삼각형 리스트로 출력하기 위해서 6개의 정점으로 구성된 배열로 선언하고 있음을 볼 수 있습니다. 이 변수의 내용은 Init() 함수에서 설정하고 있습니다.
이렇게 시스템에 만든 정점 데이터는 다음과 같이 SetStreamSource() 함수를 사용하는 것이 아닌 DrawPrimitiveUP() 함수를 통해서 파이프라인에 연결하고 다음과 같이 렌더링 명령을 내립니다.

```
pDevice->DrawPrimitiveUP(프리미티브 타입, 프리미티브 수
                         , 정점 Stream 시작 주소, 한 정점의 크기);
```

CMain::Render() 함수 안에 다음 코드를 확인해 보기 바랍니다.

```
m_pd3dDevice->DrawPrimitiveUP( D3DPT_TRIANGLELIST, 2, m_pVtx, sizeof(VtxUV1));
```

참 간편 합니다.
또한 다음과 같이 함수 안에서 만들어 놓고 바로 출력해도 됩니다.

```
VtxUV1  pVtx[6];
pVtx[0] = VtxUV1( -50.f, -50.f, 0.f, 0.f, 1.f );
...
m_pd3dDevice->DrawPrimitiveUP( D3DPT_TRIANGLELIST, 2, pVtx, sizeof(VtxUV1));
```

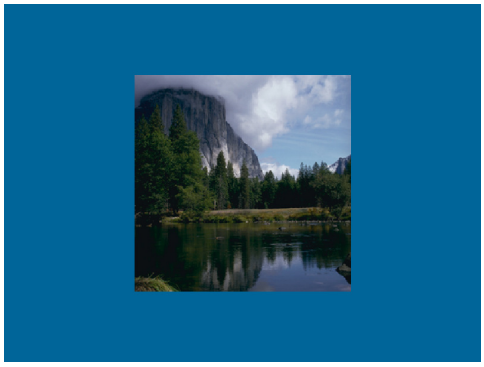

<UP 사용 예제: m3d_04_primitive08_UP_List.zip>

UP 방식 또한 인덱스 리스트를 제공합니다. 인덱스와 같이 있는 경우 DrawIndexedPrimitiveUP()함수에 프리미티브, 인덱스 포인터, 정점 포인터, 인덱스 포맷, 정점의 크기 등을 다음과 같이 인수

로 전달해서 렌더링을 합니다.

```
pDevice->DrawIndexedPrimitiveUP(
        D3DPT_TRIANGLELIST      // 항상 Triangle List
        , 최소 정점 인덱스      // 보통 0
        , 인덱스의 수           // 삼각형의 경우 삼각형 * 3
        , 프리미티브 수         // 삼각형의 개수
        , 인덱스 데이터 포인터
        , 인덱스 데이터 포맷    // D3DFMT_INDEX16 or D3DFMT_INDEX32
        , 정점 Stream의 시작 주소
        , 한 정점의 크기
        );
```

주의할 점은 프리미티브 타입은 D3DPT_TRIANGLELIST 입니다..



전체 코드는 m3d_04_primitive09_UP_Indexed.zip을 참고 하기 바랍니다.

실습과제) 정점 버퍼, 인덱스 버퍼를 생성하는 템플릿 함수를 만드시오.

실습 과제) 다음 그림과 같이 직접 텍스처를 그려서 2차원 로봇을 만드시오.

실습 과제) 색상이 있는 육면체 클래스를 UP 방식으로 인덱스를 이용해서 만드시오.

실습 과제) 위의 육면체 클래스에 행렬을 적용해서 로봇을 만드시오.