DirectX Mathematics Class

305890
2010년 봄학기
3/12/2010
박경신

DirectX Naming Convention

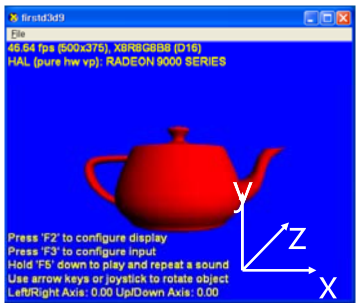
- D3D – Direct3D
- D3DX – D3D extended utility functions
- Constants and Data types
  - D3DYYY
  - D3DXYYY
  - 타입 예: typedef: D3DCOLOR, D3DXCOLOR
  - 상수 예: #define: D3D_OK, D3DXERR_INVALIDDATA
- D3DX C 함수
  - 각 단어의 첫 문자만 대문자로 시작함
  - 예: D3DXMatrixInverse
  - 주의: D3D 함수는 특별한 용도의 소수 함수만 있음
    - 예: Direct3DCreate9

DirectX Naming Convention

- D3D C++ interface
  - IDirect3DYYY
  - 예: IDirect3DDevice9 – rendering 관련된 대부분의 작업을 수행함
- D3DX C++ interface
  - ID3DXYYY
  - 예: ID3DXMesh – Mesh object를 다룰 수 있도록 하는 interface
- Interface 함수
  - 각 단어의 첫 문자만 대문자로 시작함
  - 예: IDirect3DDevice9::BeginScene, ID3DXMesh::Optimize

Vector

- 벡터는 크기(magnitude 혹은 길이 length)와 방향(direction)이 있다
- 벡터는 조명의 방향 (light source directions), 표면의 방향 (surface orientations), 물체간의 거리 (relative distance between objects) 등에서 사용되고 있다.



3D Vector

D3DXVECTOR3 class (d3dx9math.h)

```
typedef struct _D3DVECTOR {
    float x, y, z;
} D3DVECTOR;

typedef struct D3DXVECTOR3: public D3DVECTOR {
public:
    D3DXVECTOR3 () ();
    D3DXVECTOR3 (CONST FLOAT *);
    D3DXVECTOR3 (CONST D3DVECTOR&);
    D3DXVECTOR3 (FLOAT x, FLOAT y, FLOAT z);

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR3& operator += (CONST D3DXVECTOR3&);
    D3DXVECTOR3& operator -= (CONST D3DXVECTOR3&);
    D3DXVECTOR3& operator *= (CONST D3DXVECTOR3&);
    D3DXVECTOR3& operator /= (CONST D3DXVECTOR3&);
```


3D Vector

```
    // unary operators
    D3DXVECTOR3 operator +() const;
    D3DXVECTOR3 operator - () const;

    // binary operators
    D3DXVECTOR3 operator + (CONST D3DXVECTOR3&) const;
    D3DXVECTOR3 operator - (CONST D3DXVECTOR3&) const;
    D3DXVECTOR3 operator * (FLOAT) const;
    D3DXVECTOR3 operator / (FLOAT) const;

    friend D3DXVECTOR3 operator * (FLOAT, CONST struct D3DXVECTOR3&);

    BOOL operator == (CONST D3DXVECTOR3&) const;
    BOOL operator != (CONST D3DXVECTOR3&) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;
```


3D Vector

D3DXVECTOR2, D3DXVECTOR4 class (d3dx9math.h)

- D3DXVECTOR3에서와 같은 연산들이 동일하게 정의되어 있음 (외적은 예외).

```
typedef struct D3DVECTOR2 {
    FLOAT x;
    FLOAT y;
} D3DVECTOR2;

typedef struct D3DVECTOR4 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
    FLOAT w;
} D3DVECTOR4;
```


3D Vector Operations

벡터의 상등 (equal) u == v

```
D3DXVECTOR u(1.0f, 0.0f, 1.0f);
D3DXVECTOR v(0.0f, 1.0f, 0.0f);
if (u == v) return true;          // 두 벡터가 같으면
if (u != v) return true;          // 두 벡터가 다르면
```

Floating-point imprecision

```
const float EPSILON = 0.001f;
bool Equals(float lhs, float rhs)
{
    // if lhs == rhs their difference should be zero
    return fabs (lhs -  rhs) < EPSILON ? true : false;
}
```

3D Vector Operations

- 벡터의 크기 (length) length(v) $\|\vec{v}\|$

  FLOAT D3DXVec3Length(CONST D3DXVECTOR3* pV);
  D3DXVECTOR3 v(1.0f, 2.0f, 3.0f);
  Float magnitude = D3DXVec3Length(&v);   // =sqrt(14)

- 벡터의 정규화 (normalize) normalize(v) $\frac{\vec{v}}{\|\vec{v}\|}$

  D3DXVECTOR3* D3DXVec3Normalize(D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV);

3D Vector Operations

- 벡터 더하기 (addition) u + v

  D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
  D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);
  D3DXVECTOR3 sum = u + v;   // (2.0+0.0, 0.0-1.0, 1.0+5.0) = (2.0, -1.0, 6.0)

- 벡터 빼기 (subtraction) u - v

  D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
  D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);
  D3DXVECTOR3 diff = u - v;   // (2.0-0.0, 0.0+1.0, 1.0-5.0) = (2.0, 1.0, -4.0)

- 벡터 스칼라 곱 (scalar multiplication) u * *k*

  D3DXVECTOR3 u(2.0f, 0.0f, -1.0f);
  D3DXVECTOR3 scaleVec = u * 10.0f; // (2.0, 0.0, -1.0) *10.0 = (20.0, 0.0, -10.0)

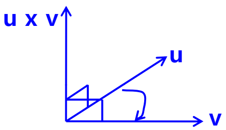
3D Vector Operations

- 벡터 내적 (dot product) $\vec{u}\cdot\vec{v} = \|\vec{u}\|\|\vec{v}\|\cos\theta = u_x v_x + u_y v_y + u_z v_z$

  FLOAT D3DXVec3Dot (CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2);
  D3DXVECTOR3 u(1.0f, -1.0f, 0.0f);
  D3DXVECTOR3 v(3.0f, 2.0f, 1.0f);
  float dot = D3DXVec3Dot(&u, &v); // 1.0*3.0 + -1.0*2.0 + 0.0*1.0 = 1.0

- 벡터 외적 (cross product)
  - 왼손 좌표계를 사용하므로 왼손 엄지 규칙을 적용
  - u x v = -(v x u)

  D3DXVECTOR3* D3DXVec3Cross (D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2);



$$\vec{u}\times\vec{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$

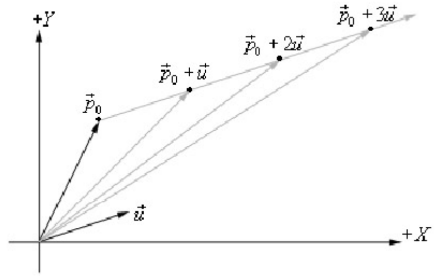
Rays, Lines, and Line Segments

- 광선 (ray) :

  $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$ where p_0 is the origin of the ray, $t \in [0, \infty)$
  u is a vector specifying the direction of the ray

- 선 (line) : $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$ where $t \in [-\infty, \infty)$
- 선분 (line segment) :

  $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{u}$ where $u = \mathbf{p}_1 - \mathbf{p}_0$, $t \in [0, 1]$



Matrix

- 다음과 같이 사각형 형태로 표기한 숫자 배열을 행렬 M (***r*** x ***c*** matrix) 라고 한다.
  - 가로로 배열된 행렬을 **행 (row)**
  - 세로로 배열된 행렬을 **열 (column)**
  - Mij는 행 i 와 열 j에 있는 **원소 (element)**

$$\boldsymbol{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \quad \}\ r(3)\ \text{rows}$$

c(3) columns

D3DX Matrix

- D3DX Matrix
  - Direct3D에서는 4x4 행렬 (matrix)과 1x4 벡터 (vector)를 사용한다.
  - $v' = v_{1x4}\ T_{4x4}$ (not $T_{4x4}\ v_{1x4}$)
- D3DMatrix
  - _ij: i는 행(row) number이고 j는 열(column) number이다.

```
typedef struct _D3DMATRIX {
    union {
      struct {
          float _11, _12, _13, _14;
          float _21, _22, _23, _24;
          float _31, _32, _33, _34;
          float _41, _42, _43, _44;
      };
      float m[4][4];
    };
} D3DMATRIX;
```


D3DX Matrix Operations

- D3DXMATRIX

```
typedef struct D3DXMATRIX: public D3DMATRIX {
    public:
        D3DXMATRIX() {};
        D3DXMATRIX(CONST FLOAT*);
        D3DXMATRIX(CONST D3DMATRIX&);
        D3DXMATRIX(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                   FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                   FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                   FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

        // access grants
        FLOAT& operator () (UNIT Row, UNIT Col);
        FLOAT operator () (UNIT Row, UNIT Col) const;

        // casting
        operator  FLOAT*();
        operator CONST FLOAT* () const;
```


D3DX Matrix Operations

```
// assignment operators
D3DXMATRIX& operator *= (CONST D3DXMATRIX&);
D3DXMATRIX& operator += (CONST D3DXMATRIX&);
D3DXMATRIX& operator -= (CONST D3DXMATRIX&);
D3DXMATRIX& operator *= (FLOAT);
D3DXMATRIX& operator /= (FLOAT);

// unary operators
D3DXMATRIX operator + () const;
D3DXMATRIX operator - () const;

// binary operators
D3DXMATRIX operator * (CONST D3DXMATRIX&) const;
D3DXMATRIX operator + (CONST D3DXMATRIX&) const;
D3DXMATRIX operator - (CONST D3DXMATRIX&) const;
D3DXMATRIX operator * (FLOAT) const;
D3DXMATRIX operator / (FLOAT) const;
```

D3DX Matrix Operations

```
friend D3DXMATRIX operator * (FLOAT, CONST D3DXMATRIX&);

BOOL operator == (CONST D3DXMATRIX&) const;
BOOL operator != (CONST D3DXMATRIX&) const;

} D3DXMATRIX, *LPD3DXMATRIX;
```


Matrix Operations

- 행렬의 연산 (arithmetic) ==, +, -, *, /

```
D3DXMATRIX A(....);          // A의 초기화
D3DXMATRIX B(....);          // B의 초기화
D3DXMATRIX C = A * B;// C = AB
```

- 행렬의 항목에 접근은 괄호연산자()를 사용한다.

```
D3DXMATRIX M;
M(0, 0) = 5.0f;              // _11 = 5.0f
```

- 단위행렬 (identity matrix) D3DXMatrixIdentity I

```
D3DXMATRIX* D3DXMatrixIdentity(D3DXMATRIX* pOut);
D3DXMATRIX M;
D3DXMatrixIdentity(&M);      // identity matrix
```


Matrix Operations

- 전치행렬 (transpose) D3DXMatrixTranspose M^T

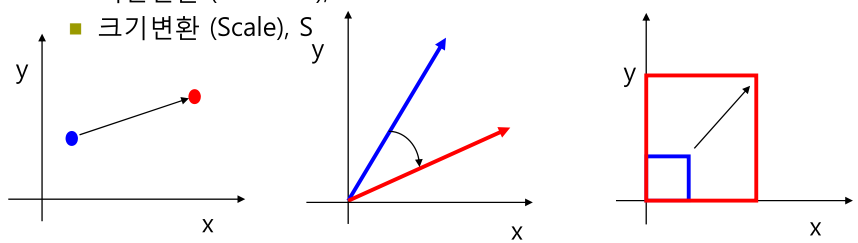
```
D3DXMATRIX* D3DXMatrixTranspose(D3DXMATRIX* pOut,
                                CONST D3DXMATRIX* pM);
D3DXMATRIX A(....);          // A 초기화
D3DXMATRIX B;
D3DXMatrixTranspose(&B, &A); // B = transpose(A)
```

- 역행렬 (inverse) D3DXMatrixInverse M^{-1}

```
D3DXMATRIX* D3DXMatrixInverse(D3DXMATRIX* pOut,
                              FLOAT* pDeterminant,
                              CONST D3DXMATRIX* pM);
D3DXMATRIX A(....);          // A 초기화
D3DXMATRIX B;
D3DXMatrixInverse(&B, 0, &A); // B = inverse(A)
                             // pDeterminant는 필요한 경우에 이용되며
                             // 그렇치 않으면 NULL을 전달한다.
```


Transformation

- 기하변환 (geometric transformation)이란 점들(points)을 한곳에서 다른 곳으로 옮겨주는 함수를 의미한다.
- 2D transformation
  - 이동변환 (Translation), T
  - 회전변환 (Rotation), R
  - 크기변환 (Scale), S



Transformation

- Direct3D에서는 변환을 표현하기 위해 4x4 행렬과 1x4 벡터를 사용한다.
  - v = (2, 6, -3, 1)
  - T = x-축으로 10-단위 이동
  - **v' = v T** = (12, 6, -3, 1)

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

$$x' = (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41})$$
$$y' = (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42})$$
$$z' = (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43})$$

Transformation

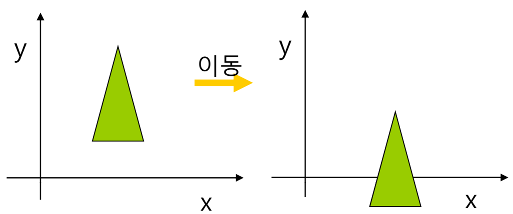
- 왜 4x4 행렬을 사용하는가?
  - 우리가 원하는 모든 변환(이동, 투영, 반사등을 포함하여)을 행렬로 표현할 수 있기 때문
  - 또한 변환 수행을 위한 벡터-행렬 곱을 일정하게 할 수 있기 때문
- Non-homogeneous/Homogeneous coordinates convert
  - (x, y, z) → (x, y, z, 1)
  - (x/w, y/w, z/w) ← (x, y, z, w)

Translation

- 이동행렬 (Translation) D3DXMatrixTranslation
  - No translation when w=0
  - $T^{-1}(p)=T(-p)$

```
D3DXMATRIX* D3DXMatrixTranslation(D3DXMATRIX* pOut,
                                  FLOAT px,
                                  FLOAT py,
                                  FLOAT pz);
```

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ px & py & pz & 1 \end{bmatrix}$$



Rotation

- 회전행렬 (Rotation) D3DXMatrixRotationX/Y/Z
  - $R^{-1}(p)=R^{T}(p)$
  - angle은 radian값으로 넣을 것

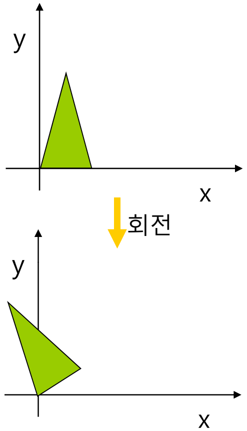
```
D3DXMATRIX* D3DXMatrixRotationX(D3DXMATRIX* pOut, FLOAT angle);
D3DXMATRIX* D3DXMatrixRotationY(D3DXMATRIX* pOut, FLOAT angle);
D3DXMATRIX* D3DXMatrixRotationZ(D3DXMATRIX* pOut, FLOAT angle);
```

3D Rotation Matrix

$$R_x(\theta)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & \sin\theta & 0\\ 0 & -\sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R_y(\theta)=\begin{bmatrix}\cos\theta & 0 & -\sin\theta & 0\\ 0 & 1 & 0 & 0\\ \sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

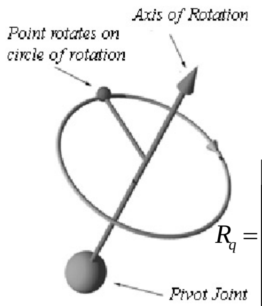
$$R_z(\theta)=\begin{bmatrix}\cos\theta & \sin\theta & 0 & 0\\ -\sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$



Rotation

- 회전행렬 (Rotation) D3DXMatrixRotationAxis
  - angle은 radian값으로 넣을 것

```
D3DXMATRIX* D3DXMatrixRotationAxis(D3DXMATRIX* pOut,
                                   CONST D3DXVECTOR3* pV,
                                   FLOAT angle);
```



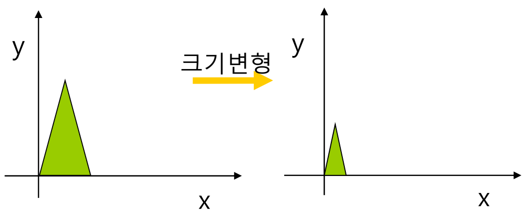
$$R_q=\begin{bmatrix}\cos\theta+x^2(1-\cos\theta) & xy(1-\cos\theta)+z\sin\theta & xz(1-\cos\theta)-a_y\sin\theta & 0\\ xy(1-\cos\theta)-z\sin\theta & \cos\theta+y^2(1-\cos\theta) & yz(1-\cos\theta)+x\sin\theta & 0\\ xz(1-\cos\theta)+y\sin\theta & yz(1-\cos\theta)-x\sin\theta & \cos\theta+z^2(1-\cos\theta) & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Scaling

- 크기변형행렬 (Scaling) D3DXMatrixScaling
  - $S^{-1}(sx, sy, sz)=S(1/sx, 1/sy, 1/sz)$

```
D3DXMATRIX* D3DXMatrixScaling(D3DXMATRIX* pOut,
                              FLOAT sx,
                              FLOAT sy,
                              FLOAT sz);
```

$$S=\begin{bmatrix}sx & 0 & 0 & 0\\ 0 & sy & 0 & 0\\ 0 & 0 & sz & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$



Inverse Transformation Matrix

$$T^{-1}(p) = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -px & -py & -pz & 1\end{bmatrix}$$

$$S^{-1}(s) = \begin{bmatrix}1/sx & 0 & 0 & 0\\ 0 & 1/sy & 0 & 0\\ 0 & 0 & 1/sz & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R_X^{-1}(\theta) = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_Y^{-1}(\theta) = \begin{pmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_Z^{-1}(\theta) = \begin{pmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Composing Transformation

- 예를 들어 벡터 p=[5, 0, 0, 1]을 모든 축으로 1/5 크기로 배율을 변경한 후, y-축으로 π/4만큼 회전시킨 다음, x-축으로 1단위, y-축으로 2단위, z-축으로 -3단위만큼 이동
- Q = S(1/5, 1/5, 1/5) R_y(PI/4) T(1,2,-3)
- pQ = [1.707, 2, -3707, 1]

$$SR_yT = \begin{pmatrix} 1/5 & 0 & 0 & 0 \\ 0 & 1/5 & 0 & 0 \\ 0 & 0 & 1/5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} .707 & 0 & -.707 & 0 \\ 0 & 1 & 0 & 0 \\ .707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 2 & -3 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} .1414 & 0 & -.1414 & 0 \\ 0 & 1 & 0 & 0 \\ .1414 & 0 & .1414 & 0 \\ 1 & 2 & -3 & 1 \end{pmatrix} = Q$$

Transformation

- D3DXVec3Transform 벡터 pV를 행렬 pM로 변환

```
D3DXVECTOR4* WINAPI D3DXVec3Transform(
        D3DXVECTOR4* pOut,
        CONST D3DXVECTOR3* pV,
        CONST D3DXMATRIX* pM);
```

- D3DXVec3TransformCoord 벡터 pV를 행렬 pM로 변환
  - 벡터의 네번째 성분이 1로 인식

```
D3DXVECTOR3* WINAPI D3DXVec3TransformCoord(
        D3DXVECTOR3* pOut,
        CONST D3DXVECTOR3* pV,
        CONST D3DXMATRIX* pM);
```

- D3DXVec3TransformNormal 벡터 pV를 행렬 pM로 변환
  - 벡터의 네번째 성분이 0으로 인식

```
D3DXVECTOR3* WINAPI D3DXVec3TransformNormal(
        D3DXMATRIX* pOut,
        CONST D3DXVECTOR3* pV,
        CONST D3DXMATRIX* pM);
```

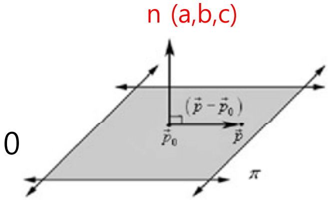

Plane

- 평면은 하나의 법선 벡터 (normal vector) n과 평면 상의 점 p_0으로 표현된다: n = (a, b, c), p=[n, d]

  ax + by + cz + d = 0

  n•p + d = 0

  d = -n•p



- 평면 위의 점 p에 대해, n•(p - p_0) =0
- 점 p과 평면 (n, d)의 공간 관계
  - 만약 n•p + d = 0라면, p는 평면에 있다.
  - 만약 n•p + d > 0라면, p는 평면의 바깥쪽에 있다.
  - 만약 n•p + d < 0라면, p는 평면의 안쪽에 있다.
- 만약 평면의 법선 벡터 n이 단위 길이라면, n•p + d로 평면에서 점 p까지의 부호를 가진 가장 짧은 거리 (the shortest signed distance)를 얻을 수 있다: d =-n•p

Plane

- D3DXPLANE

```
typedef struct D3DXPLANE{
#ifdef __cplusplus
    public:
        D3DXPLANE() {};
        D3DXPLANE(CONST FLOAT*);
        D3DXPLANE(CONST D3DFLOAT16*);
        D3DXPLANE(FLOAT a, FLOAT b, FLOAT c, FLOAT d);

        // casting
        operator  FLOAT*();
        operator CONST FLOAT* () const;

        // assignment operators
        D3DXPLANE& operator  *= (FLOAT);
        D3DXPLANE& operator /= (FLOAT);

        // unary operators
        D3DXPLANE operator + () const;
        D3DXPLANE operator - () const;
```

Plane

```
// binary operators
D3DXPLANE operator * (FLOAT) const;
D3DXPLANE operator / (FLOAT) const;

friend D3DXPLANE operator * (FLOAT, CONST D3DXPLANE&);

BOOL operator == (CONST D3DXPLANE&) const;
BOOL operator != (CONST D3DXPLANE&) const;
#endif    // __cplusplus
FLOAT a, b, c, d;
} D3DXPLANE, *LPD3DXPLANE;
```


Relationship between Point and Plane

- 점 p과 평면 (n, d)의 공간 관계
  - 만약 n•p + d = 0라면, p는 평면에 있다.
  - 만약 n•p + d > 0라면, p는 평면의 바깥쪽에 있다.
  - 만약 n•p + d < 0라면, p는 평면의 안쪽에 있다.
- D3DXPlaneDotCoord
  - 평면(a, b, c, d)와 벡터 (x, y, z)에서 a*x + b*y + c*z + d*1을 준다.

```
D3DXPLANE p(0.0, 1.0, 0.0, 0.0);
D3DXVECTOR3 v(3.0, 5.0, 2.0);
float x = D3DXPlaneDotCoord(&p, &v);
if (x approximately equals 0.0) // 평면상에 있다
if (x > 0) // 평면 밖에 있다
if (x < 0) // 평면 안에 있다

- Approximately equal
const float EPSILON = 0.001f;
boolean Equals (float lhs, float rhs) { return fabs(lhs – rhs) < EPSILON ? true : false;}
```


Relationship between Point and Plane

- D3DXPlaneDot
  - 평면(a, b, c, d)와 벡터 (x, y, z, w)에서 a*x + b*y + c*z + d*w을 준다.

```
FLOAT D3DXPlaneDot(CONST D3DXPLANE* pP,
                   CONST D3DXVECTOR4* pV);
```

- D3DXPlaneDotCoord
  - 평면(a, b, c, d)와 벡터 (x, y, z)에서 a*x + b*y + c*z + d*1을 준다.

```
FLOAT D3DXPlaneDotCoord(CONST D3DXPLANE* pP,
                        CONST D3DXVECTOR3* pV);
```

- D3DXPlaneDotNormal
  - 평면(a, b, c, d)와 벡터 (x, y, z)에서 a*x + b*y + c*z + d*0을 준다.

```
FLOAT D3DXPlaneDotNormal(CONST D3DXPLANE* pP,
                         CONST D3DXVECTOR3* pV);
```


Plane Construction

- 법선벡터 (normal) n과 거리 (signed distance) d
  - D3DXPLANE p(a, b, c, d)
- 법선벡터 (normal) n과 평면 상의 한 점 p_0
  - $d = -n \cdot p_0$

```
D3DXPLANE* D3DXPlaneFromPointNormal(D3DXPLANE* pOut,
                                    CONST D3DXVECTOR3* pPoint,
                                    CONST D3DXVECTOR3* pNormal);
```

- 평면 상의 세 개의 점 p_0, p_1, p_2
  - $u = p_1 - p_0$; $v = p_2 - p_0$; $n = u \times v$; $d = -n \cdot p_0$

```
D3DXPLANE* D3DXPlaneFromPoints(D3DXPLANE* pOut,
                               CONST D3DXVECTOR3* pV1,
                               CONST D3DXVECTOR3* pV2,
                               CONST D3DXVECTOR3* pV3);
```

Plane Normalization

- 평면의 정규화 (normalization)
  - 평면의 법선 벡터 (normal)를 정규화
  - 법선 벡터의 길이가 상수 d에 영향을 주기 때문에, d도 역시 정규화

$$\frac{1}{\|\mathrm{n}\|}(\mathrm{n},\mathrm{d}) = \left(\frac{n}{\|n\|}, \frac{d}{\|n\|}\right)$$

```
D3DXPLANE* D3DXPlaneNormalize(D3DXPLANE* pOut,
                              CONST D3DXPLANE* pP);
```


Plane Transformation

- 평면변환
  - 정규화된 평면이 v=(n, d)라면 이 평면을 변환 행렬 T로 변환한 평면은 $v(T^{-1})^T$이다.

```
D3DXPLANE* D3DXPlaneTransform(D3DXPLANE* pOut,
                              D3DXPLANE* pP,
                              CONST D3DXMATRIX* pM);

D3DXMATRIX T(....); // 변환행렬 초기화
D3DXMATRIX inverseOfT;
D3DXMATRIX inverseTransposeOfT;
D3DXMatrixInverse(&inverseOfT, 0, &T);
D3DXMatrixTranspose(&inverseTransposeOfT, &inverseOfT);
D3DXPLANE p(....);          // 평면의 초기화
D3DXPlaneNormalize(&p, &p);      // 정규화
D3DXPlaneTransform(&p, &p, &inverseTransposeOfT);
```


Plane Transformation

- 평면변환 예제

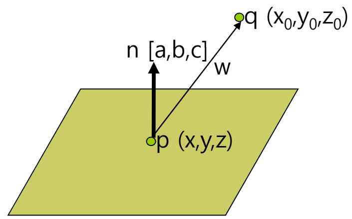
```
D3DXPLANE planeNew;
D3DXPLANE plane(0, 1, 1, 0);
D3DXPlaneNormalize(&plane, &plane);

D3DXMATRIX matrix;
D3DXMatrixScaling(&matrix, 1.0, 2.0, 3.0);
D3DXMatrixInverse(&matrix, 0, &matrix);
D3DXMatrixTranspose(&matrix, &matrix);

// Transform to a new plane = (0, 0.343, 0.235, 0)
D3DXPlaneTransform(&planeNew, &plane, &matrix);
```


Computing a Distance from Point to Plane

- 공간에 하나의 점 q를 가지고 있고, 점 q에서 평면 (n, d)과 가장 가까운 거리를 구하라
  - n은 평면의 법선 벡터 (normal vector)이고, *D* 는 평면의 한 점 p와 점 q와의 거리

$$w = [x_0 - x, y_0 - y, z_0 - z]$$

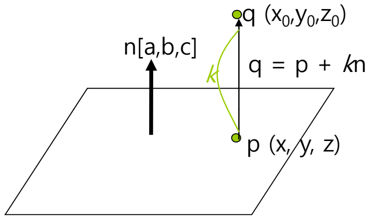
$$D = \frac{|n \bullet w|}{\|n\|} = \frac{|a(x_0 - x) + b(y_0 - y) + c(z_0 - z)|}{\sqrt{a^2 + b^2 + c^2}} = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$$



Projecting w onto n: $w_{\parallel} = n\frac{w \cdot n}{\|n\|^2}$ & $\|w_{\parallel}\| = \frac{|w \cdot n|}{\|n\|}$

Closest Point on the Plane

- 공간에 하나의 점 q를 가지고 있고, 점 q에서 가장 가까운 평면 (n, d)상의 점 p를 구하라
  - p = q – *k*n (*k*는 q에서 plane과의 the shortest signed distance)
  - n이 단위벡터(unit vector)인 경우, *k* = n•q + d
  - p = q – (n•q + d)n



$$\text{Distance}(\text{q}, \text{plane}) = \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$$

$$where\ q(x_0, y_0, z_0)\ \ and\ \ Plane\ ax + by + cz + d = 0$$

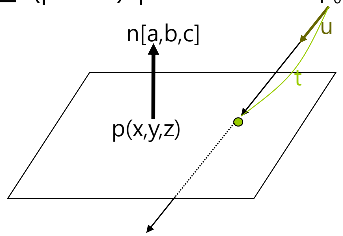
Intersection of Ray and Plane

- 광선 (ray) **p**(*t*) = **p**$_0$ + *t***u** & 평면 (plane) p•n + d = 0
- 광선/평면의 교차점:

$$(\mathrm{p}_0 + t\mathrm{u}) \cdot \mathrm{n} + d = 0$$

$$t\mathrm{u} \cdot \mathrm{n} = -d - \mathrm{p}_0 \cdot \mathrm{n}$$

$$t = \frac{-(\mathrm{p}_0 \cdot \mathrm{n} + d)}{\mathrm{u} \cdot \mathrm{n}}$$



- 만약 광선이 평면과 평행하다면, denominator **u**•**n**=0 따라서 광선은 평면과 교차하지 않는다.
- 만약 *t* 값이 범위 [0, ∞)내에 있지 않으면, 광선은 평면과 교차하지 않는다.
- $\mathrm{p}\left(\frac{-(\mathrm{p}_0 \cdot \mathrm{n} + d)}{\mathrm{u} \cdot \mathrm{n}}\right) = \mathrm{p}_0 + \frac{-(\mathrm{p}_0 \cdot \mathrm{n} + d)}{\mathrm{u} \cdot \mathrm{n}}\mathrm{u}$