1 벡터와 동차 좌표

1.1 동차 좌표

1차원 좌표에서 (2), (4)가 있다고 가정 합시다. 이 둘은 당연히 같지 않습니다. 그런데 이것을 2차원으로 확장하는 동차 좌표계를 사용하면 (2) ->(2, 1), (4) ->(4,1)로 변경 될 수 있습니다. 새롭게 추가된 좌표의 값은 일반적으로 값은 1로 유지합니다. 이렇게 좌표계를 확장하더라도 원래의 좌표 값을 그대로 유지 시켜주는 것이 동차 좌표입니다.
그런데 동차 좌표계를 사용할 경우 주의해야 합니다. 예를 들어 2차원 동차좌표계에서 (2,1) ≡ (4, 2) 가 되고, 또한 (2,1)과 같은 좌표 값은 동차 좌표계에서는 무수히 많은 좌표 값을 가질 수 있습니다. 2차원의 경우 이러한 내용을 일반화 시키면 (a, b) ≡ (a/b, 1)로 표현할 수 있습니다. 예를 들면 (2,4) ≡ (3, 6) ≡ (1.5, 3) ≡ (0.5, 1)이 가능합니다.

3차원은 2차원의 확장입니다. 그러므로 2차원 좌표(x, y)를 3차원 동차 좌표계의 좌표로 바꾸면 (x, y, 1)이 되고 이것을 일반화 하면 다음과 같이 됩니다.

(C *x, C* y, C)(단, C ≠ 0)

3차원 벡터를 4차원 동차 좌표계로 확장 할 수 있습니다. 위의 과정을 그대로 적용하면 3차원 좌표에 대한 4차원 동차 좌표계 좌표는 다음과 같이 변환 됩니다.

(x, y, z) ->(x, y, z, 1) ≡ (w * x, w * y, w * z, w)

4차원 동차좌표계에서는 C 대신 w를 사용하고, 입력한 벡터의 연산의 마지막에 이 값으로 x 성분, y 성분, z 성분을 나누어 원래 차원의 크기로 바꾸어 주는데 사용됩니다. 이 때 w 값이 이용되는 과정을 요약하면 다음과 같습니다.

V(x, y, z) ->동차 좌표계로 변환 (x, y, z, 1)
 -> 행렬 변환 (x', y', z', w')
 -> 원 좌표계의 크기로 환원 (x'/w', y'/w', z'/w', w'/w')
 -> 최종 (x'/w', y'/w', z'/w')

컴퓨터 그래픽스에서는 이 w를 특별히 Reciprocal of Homogeneous W 라 부르며 줄여서 RHW 라 합니다.

문제 1-1)(2,3) 과 같은 3차원 동차 좌표계의 좌표 3개만 구하시오.

문제 1-2)(3,4,2) 와 같은 4차원 동차 좌표계의 좌표 3개만 구하시오.

풀이 1-1) (4,6), (-2, -3), (200, 3) 등
풀이 1-2) (-3, -4, -2), (-1.5, -2, -1) (6, 8, 4) 등

1.2 동차 좌표와 행렬

1) 벡터 $\vec{v} = \vec{v}$에 대한 식을 $\vec{v} = \vec{v}M$ 을 만드는 행렬 M을 동차 좌표계에서 구하도록 합시다.
행렬을 공부한 사람은 위의 수식에 대한 행렬 M은 3X3 항등 행렬 임을 알고 있을 것입니다. 그러나 우리가 구하고자 하는 것은 동차좌표계에서의 행렬이므로 먼저 주어진 벡터를 동차좌표계의 좌표로 바꾸어 주어야 합니다.

벡터 $\vec{v} = (x, y, z)$라 합시다. 이것을 4차원 동차 좌표계로 바꾸면 다음과 같이 됩니다.

(x, y, z) -> $(x, y, z, 1)$
≡ $(w*x, w*y, w*z, w)$

이것을 행렬로 나타내면

$$(x, y, z, 1) = (x, y, z, 1)\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & w \end{pmatrix}$$

이 되고, 동차 좌표계에서 행렬 M = $\begin{pmatrix} w & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & w \end{pmatrix}$ 이 됩니다.

일반 좌표계에서 항등 행렬은 거칠게 표현 한다면 동차 좌표계에서는 대각선의 값이 같고, 나머지가 0인 행렬의 특별한 값이 되는 특별한 행렬일 뿐입니다.

2) $\vec{v} = -\vec{v}$ 에 대한 식을 $\vec{v} = \vec{v}M$ 을 만드는 행렬 M을 동차 좌표계에서 구해 봅시다.

벡터 $\vec{v} = (x, y, z, 1)$라 하면 벡터 $-\vec{v} = (-x, -y, -z)$가 됩니다. 주어진 문제의 식을 4차원 동차

좌표계로 바꾸면 다음과 같이 됩니다.

$-\vec{v} = (-x, -y, -z, 1)$

이것을 $\vec{v} = \vec{v}M$ 식으로 변경하면 다음과 같이 됩니다.

$$(-x, -y, -z, 1) = (x, y, z, 1)\begin{pmatrix} -w & 0 & 0 & 0 \\ 0 & -w & 0 & 0 \\ 0 & 0 & -w & 0 \\ 0 & 0 & 0 & w \end{pmatrix}$$

따라서 행렬 M=$\begin{pmatrix} -w & 0 & 0 & 0 \\ 0 & -w & 0 & 0 \\ 0 & 0 & -w & 0 \\ 0 & 0 & 0 & w \end{pmatrix}$ 가 됩니다.

3) $\vec{v} = \vec{c}$ 에 대한 식을 $\vec{v} = \vec{v}M_c$ 을 만드는 벡터 c의 성분으로 구성된 행렬 M을 동차 좌표계에서 구해 봅시다.

벡터 $\vec{v} = (v_x, v_y, v_z, 1)$ 라 하고 $\vec{c} = (c_x, c_y, c_z, 1)$ 라 하면

$$(v_x, v_y, v_z, 1) = (c_x, c_y, c_z, 1)$$

$$= (v_x, v_y, v_z, 1)\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ w*c_x & w*c_y & w*c_z & w \end{pmatrix}$$

따라서 행렬 M_c=$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ w*c_x & w*c_y & w*c_z & w \end{pmatrix}$ 가 됩니다.

4) $\vec{v} = -\vec{c}$ 에 대한 식을 $\vec{v} = \vec{v}M_c$ 을 만드는 벡터 c의 성분으로 구성된 행렬 M을 동차 좌표계에

서 구해 보면 3)의 경우와 같으므로

벡터 $\vec{v} = (v_x, v_y, v_z, 1)$ 라 하고 $\vec{c} = (c_x, c_y, c_z, 1)$ 라 하면

$$(v_x, v_y, v_z, 1) = (-c_x, -c_y, -c_z, 1)$$

$$= (v_x, v_y, v_z, 1) \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -w*c_x & -w*c_y & -w*c_z & w \end{pmatrix}$$

따라서 행렬 $M_c = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -w*c_x & -w*c_y & -w*c_z & w \end{pmatrix}$ 가 됩니다.

주의할 것은 w 값을 아직 1로 만들지 않았다는 것입니다. 이 것은 동차좌표계의 행렬이기 때문이다. 이것을 실제로 적용하려면 위의 행렬을 w값으로 나누어야 할 것입니다.

문제1-3) $\vec{A}, \vec{B}, \vec{C}$ 가 3차원 벡터일 때 $\vec{C} = \vec{A} + \vec{B}$ 를 $\vec{C} = \vec{A}\, M_B$ 으로 나타내고자 할 때 벡터 $\vec{B}$ 의 성분으로 구성된 행렬 M_B 을 4차원 동차좌표계에서 구하시오.

문제1-4) $\vec{A}, \vec{B}, \vec{C}$ 가 3차원 벡터일 때 $\vec{C} = \vec{A} - \frac{1}{k}\vec{B}$ 를 $\vec{C} = \vec{A}\, M_B$ 으로 나타내고자 할 때 벡터 $\vec{B}$ 의 성분으로 구성된 행렬 M_B 을 4차원 동차좌표계에서 구하시오.

풀이 1-3) 동차 좌표계로 풀면

$$\vec{A} = \vec{A}\,, \quad (A_x, A_y, A_z, 1) = (A_x, A_y, A_z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\vec{A}=\vec{B},\ (A_x, A_y, A_z, 1)=(A_x, A_y, A_z, 1)\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\B_x&B_y&B_z&1\end{pmatrix}$$

$$\vec{C}=\vec{A}+\vec{B}$$

$$=\vec{A}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}+\vec{A}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\B_x&B_y&B_z&1\end{pmatrix}$$

$$=\vec{A}\left(\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\B_x&B_y&B_z&1\end{pmatrix}\right)\quad=\vec{A}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\B_x&B_y&B_z&1\end{pmatrix}$$

$$\therefore M_B=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\B_x&B_y&B_z&1\end{pmatrix}\ \text{또는}\ \therefore M_B=\begin{pmatrix}w&0&0&0\\0&w&0&0\\0&0&w&0\\w*B_x&w*B_y&w*B_z&w\end{pmatrix}$$

위의 풀이 과정을 보면 동차좌표의 W 값은 연산에서 같은 값이어야 하고, 또한 행렬의 덧셈에는 관여하지 않음을 주의해야 합니다.

풀이 1-4) 동차 좌표계로 풀면

$$\vec{A}=\vec{A},\ \vec{A}=\vec{A}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

$$-\frac{1}{k}\vec{B}\quad=\vec{A}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\-\frac{1}{k}B_x&-\frac{1}{k}B_y&-\frac{1}{k}B_z&1\end{pmatrix}\quad=\vec{A}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\-B_x&-B_y&-B_z&k\end{pmatrix}$$

연산을 위해서 $\vec{A} = \vec{A}\begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & k \end{pmatrix}$ 로 바꿉니다.

$$\vec{C} = \vec{A} - \frac{1}{k}\vec{B}$$

$$= \vec{A}\begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & k \end{pmatrix} + \vec{A}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -B_x & -B_y & -B_z & k \end{pmatrix} = \vec{A}\begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ -B_x & -B_y & -B_z & k \end{pmatrix}$$

$$\therefore M_B = \begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ -B_x & -B_y & -B_z & k \end{pmatrix}$$

1.3 벡터의 내적과 행렬의 특별한 표현

3차원벡터 $\vec{A}, \vec{B}, \vec{C}$ 에서 $(\vec{A} \cdot \vec{B})\vec{C}$ 를 행렬로 표현해 봅시다.

$\vec{A} = (A_x, A_y, A_z)$, $\vec{B} = (B_x, B_y, B_z)$, $\vec{C} = (C_x, C_y, C_z)$ 라 한다면

$$(\vec{A} \cdot \vec{B})\vec{C}$$

$$= (\quad (A_xB_x + A_yB_y + A_zB_z)C_x ,$$
$$(A_xB_x + A_yB_y + A_zB_z)C_y ,$$
$$(A_xB_x + A_yB_y + A_zB_z)C_z)$$

$$= (\quad A_xB_xC_x + A_yB_yC_x + A_zB_zC_x ,$$
$$A_xB_xC_y + A_yB_yC_y + A_zB_zC_y ,$$
$$(A_xB_xC_z + A_yB_yC_z + A_zB_zC_z))$$

$$= (A_x, A_y, A_z) \begin{pmatrix} B_xC_x & B_xC_y & B_xC_z \\ B_yC_x & B_yC_y & B_yC_z \\ B_zC_x & B_zC_y & B_zC_z \end{pmatrix}$$

$(\vec{A}\cdot\vec{B})\vec{C} = \vec{A}\cdot\vec{B}\vec{C}$ 라 하고 이때 $\vec{B}\vec{C} = \begin{pmatrix} B_xC_x & B_xC_y & B_xC_z \\ B_yC_x & B_yC_y & B_yC_z \\ B_zC_x & B_zC_y & B_zC_z \end{pmatrix}$ 인 행렬로 만들 수 있습니다.

물리에서는 $\vec{B}\vec{C}$와 같이 종종 두 벡터의 조합으로 이루어진 행렬을 사용하므로 의미를 기억하도록 합시다.

$(\vec{A}\cdot\vec{B})\vec{C} = (\vec{B}\cdot\vec{A})\vec{C} = \vec{B}\cdot\vec{A}\vec{C}$ 이 가능하므로 $\vec{A}\vec{C} = \begin{pmatrix} A_xC_x & A_xC_y & A_xC_z \\ A_yC_x & A_yC_y & A_yC_z \\ A_zC_x & A_zC_y & A_zC_z \end{pmatrix}$ 행렬을 얻을 수 있습니다.

문제 1-5) $\vec{P}, \vec{A}, \vec{B}, \vec{C}$가 3차원 벡터일 때 $\vec{P} = \vec{A} + \frac{D}{k}\vec{B} + -\frac{(\vec{A}\cdot\vec{C})}{k}\vec{B}$ 를 $\vec{C} = \vec{A}\,M_{B,C}$ 식을 만족하는 벡터 $\vec{B}, \vec{C}$ 의 성분으로 구성된 행렬 M_{BC} 을 4차원 동차좌표계에서 구하시오.(단, D, k는 스칼라)

문제에서 k값으로 나누고 있으므로 이것을 w 값으로 사용합니다.

$$\vec{A} = \vec{A}\begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & k \end{pmatrix}$$

$$\frac{D}{k}\vec{B} = \vec{A}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ D*B_x & D*B_y & D*B_z & k \end{pmatrix}$$

$$-\frac{(\vec{A}\cdot\vec{C})}{k}\vec{B} \quad = \vec{A}\cdot\left(-\frac{1}{k}(\vec{C}\vec{B})\right) \quad = \vec{A}\begin{pmatrix} -C_xB_x & -C_xB_y & -C_xB_z & 0 \\ -C_yB_x & -C_yB_y & -C_yB_z & 0 \\ -C_zB_x & -C_zB_y & -C_zB_z & 0 \\ 0 & 0 & 0 & k \end{pmatrix}$$

$$\vec{P} = \vec{A}\begin{pmatrix} k & 0 & 0 & 0 \\ 0 & k & 0 & 0 \\ 0 & 0 & k & 0 \\ 0 & 0 & 0 & k \end{pmatrix} + \vec{A}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ D*B_x & D*B_y & D*B_z & k \end{pmatrix} + \vec{A}\begin{pmatrix} -C_xB_x & -C_xB_y & -C_xB_z & 0 \\ -C_yB_x & -C_yB_y & -C_yB_z & 0 \\ -C_zB_x & -C_zB_y & -C_zB_z & 0 \\ 0 & 0 & 0 & k \end{pmatrix}$$

$$\vec{P} = \vec{A}\begin{pmatrix} k-C_xB_x & -C_xB_y & -C_xB_z & 0 \\ -C_yB_x & k-C_yB_y & -C_yB_z & 0 \\ -C_zB_x & -C_zB_y & k-C_zB_z & 0 \\ D*B_x & D*B_y & D*B_z & k \end{pmatrix}$$

$$\therefore M_{BC} = \begin{pmatrix} k-C_xB_x & -C_xB_y & -C_xB_z & 0 \\ -C_yB_x & k-C_yB_y & -C_yB_z & 0 \\ -C_zB_x & -C_zB_y & k-C_zB_z & 0 \\ D*B_x & D*B_y & D*B_z & k \end{pmatrix}$$

또는 $$M_{BC} = \begin{pmatrix} -k+C_xB_x & C_xB_y & C_xB_z & 0 \\ C_yB_x & -k+C_yB_y & C_yB_z & 0 \\ C_zB_x & C_zB_y & -k+C_zB_z & 0 \\ -D*B_x & -D*B_y & -D*B_z & -k \end{pmatrix}$$

행렬 M_{BC} 는 평행광의 그림자를 만드는 행렬로 그림자 행렬(Shadow Matrix) 라 합니다.

2 충돌

충돌이란 거칠게 표현한다면 A∩B≠Φ을 의미합니다. 즉, 두 충돌 대상이 공유하는 어떠한 요소가 있어야 충돌이 이루어지기 때문입니다. 3D 게임은 현실을 프로그램으로 컴퓨터에서 흉내를 낸 것입니다. 따라서 일반적인 수학의 원리를 적용하기 보다는 특수한 부분으로 나누어서 문제들을 해결합니다. 충돌을 수학의 집합 등과 같이 표현한다면 사람은 편리하지만 프로그램으로 작성하기가 무척 어렵습니다. 일반적인 상황이 아닌 좀 더 지역적이고 특수한 상황으로 만든다면 프로그램이

어느 정도 가능하게 됩니다. 게임에서는 충돌을 다음과 같이 몇 가지 경우로 나누어서 프로그램을 작성합니다.

1. Point vs. Others
2. Sphere vs. Others
3. Line vs. Others
4. Triangle vs. Others
5. Box vs. Others
6. 기타

Others는 점(또는 구), 선(휴한 직선, 무한직선), 삼각형, 상자(AABB, OBB) 가 대응 됩니다. 기타에는 실린더, 다각형 충돌 등이 있습니다. 먼저 점에 대한 충돌을 살펴 봅시다.

2.1 점의 충돌

2.1.1 점 과 점 충돌

점과 점의 충돌은 너무나 간단합니다. 두 개의 위치가 같기만 하면 충돌이기 때문입니다. 충돌 시험을 할 두 점 V1, V2 가 있다면

```
if(v1.x == v2.y && v1.y == v2.y && v1.z == v2.z)
        충돌;
```

게임 프로그램에서는 때로는 적당히 작은 편차 값 Epsilon 을 이용해서 점 사이가 편차 값 안에 있으면 충돌 판정을 내리기도 합니다.

```
if( |V2.x - V1.x| <Epsilon && |V2.y - V1.y| <Epsilon && |V2.z - V1.z| < Epsilon)
        충돌;
```

이 것을 구현 하면 다음과 같습니다.

```
INT CollisionPointToPoint(const D3DXVECTOR3* V1
                        , const D3DXVECTOR3* V2
                        , FLOAT fEpsilon=0.0001f )
{
        INT hr=-1;
```

```
        if( fabsf(V2->x - V1->x) <fEpsilon &&
            fabsf(V2->y - V1->y) <fEpsilon &&
            fabsf(V2->z - V1->z) <fEpsilon)
                hr = 0;

        return hr;
}
```

2.1.2. 점과 선의 충돌

점과 선의 충돌은 직선이 유한 직선, 무한 직선에 따라 달라집니다. 직선이 무한 직선이라면 구와 무한 직선의 충돌과 동일한데 이 부분은 구와 직선의 충돌에서 다시 다루겠습니다.
만약 유한 직선과 점의 충돌이라면 점이 선 안에 있는 지만 판단하면 됩니다. 이 판단은 두 가지 방법이 있는데 하나는 길이를 이용하는 것입니다. 만약 점이 선 내부에 있다면 점과 선의 양끝 점의 길이의 합은 전체 선의 길이와 같아야 할 것입니다. 만약 밖에 있는 경우라면 전체의 길이는 유한 직선의 길이보다 더 길어 질 수 밖에 없습니다.
직선의 두 점을 P1, P2 라 하고 충돌 시험을 할 점을 V라 하면 다음과 같은 의사 코드(Pseudo Code)를 만들 수 있습니다.

```
Float fEpsilon=0.0001f;
Vector3 vcT = V - P1;
Vector3 vcL = P2 - P1;

Vector3 vcH = Cross(vcT, vcL);
Float fHsq = LengthSquare(vcH);

if( fHsq > fEpsilon)
        return 충돌 아님;

Float L  = |P2.x - P1.x| + |P2.y - P1.y| + |P2.z - P1.z|;
Float L1 = |V.x - P1.x| + |V.y - P1.y| + |V.z - P1.z|;
Float L2 = |V.x - P2.x| + |V.y - P2.y| + |V.z - P2.z|;

if( (L1 + L2)< (L+fEpsilon))
        충돌;
```

이 것을 DirectX SDK를 이용해서 구현 하면 다음과 같이 작성될 수 있습니다.

```
INT CollisionPointToLine(const D3DXVECTOR3* V
                         , const D3DXVECTOR3* P1
                         , const D3DXVECTOR3* P2
                         , FLOAT fEpsilon=0.0001f )
{
        INT hr=-1;

        FLOAT fHsq;
        D3DXVECTOR3 vcT = *V - *P1;
        D3DXVECTOR3 vcL = *P2 - *P1;

        D3DXVECTOR3 vcH;
        D3DXVec3Cross(&vcH, &vcT, &vcL);
        fHsq = D3DXVec3LengthSq(&vcH);

        if( fHsq > fEpsilon)
                return hr;

        FLOAT L =       fabsf( P2->x - P1->x) +
                        fabsf( P2->y - P1->y) +
                        fabsf( P2->z - P1->z);

        FLOAT L1 =      fabsf( V->x - P1->x) +
                        fabsf( V->y - P1->y) +
                        fabsf( V->z - P1->z);

        FLOAT L2 =      fabsf( V->x - P2->x) +
                        fabsf( V->y - P2->y) +
                        fabsf( V->z - P2->z);

        if( (L1 + L2)< (L+ fEpsilon))
                hr =0;
```

```
        return hr;
}
```

벡터의 내적(Dot Product)을 이용한다면 좀 더 간결하게 코드를 구성할 수 있습니다. 점이 선의 내부에 있다면 이 점에서 두 벡터 V - P1, V - P2를 먼저 구합니다. 만약 점이 선의 내부에 있다면 이 두 벡터는 방향이 반대여서 내적을 구하면 0보다 작은 값이 됩니다. 만약 점이 선의 밖에 있다면 두 벡터는 평행하게 되어 0보다 큰 값을 갖게 됩니다.

```
Float fDot = Dot(V - P1, V - P2)

if(fDot <= 0)
        충돌;
```

DirectX SDK를 이용해서 구현 하면 다음과 같습니다.

```
INT CollisionPointToLine(const D3DXVECTOR3* V
                        , const D3DXVECTOR3* P1
                        , const D3DXVECTOR3* P2
                        , FLOAT fEpsilon=0.0001f    )
{
        INT hr=-1;

        FLOAT fHsq;
        D3DXVECTOR3 vcT1 = *V - *P1;
        D3DXVECTOR3 vcT2 = *V - *P2;
        D3DXVECTOR3 vcL = *P2 - *P1;

        D3DXVECTOR3 vcH;
        D3DXVec3Cross(&vcH, &vcT1, &vcL);
        fHsq = D3DXVec3LengthSq(&vcH);

        if( fHsq > fEpsilon)
                return hr;

        FLOAT fDot =     D3DXVec3Dot(&vcT1, &vcT2);

        if( fDot<=0.f)
```

```
                hr =0;

        return hr;
}
```

2.1.3. 점과 평면의 충돌

점과 면은 법선 벡터와 점의 내적 값을 이용합니다. 평면의 방정식에 점(위치)를 대입하면 0이 되어야 합니다. 이 경우도 그리 어렵지 않습니다. 평면의 법선벡터(N)와 원점에서의 최단거리(D) 가 주어지고 충돌한 점(P) 가 있는 경우 의사 코드로 구현 하면 다음과 같이 간단하게 만들 수 있습니다.

```
Float fEpsilon=0.0001f;
Float d = Dot(N, P)- D;

if(|d|< fEpsilon)
        충돌;
```

때로는 이 d 값을 이용해서 평면의 위(d>0), 또는 아래(d<0)에 있는지 판별하기도 합니다.
DirectX SDK를 이용해서 구현 하면 다음과 같습니다.

```
INT CollisionPointToPlan(const D3DXVECTOR3* P
                        , const D3DXVECTOR3* N
                        , FLOAT D
                        , FLOAT fEpsilon=0.0001f   )
{
        INT hr=-1;

        FLOAT d = D3DXVec3Dot(N, P) - D;

        if(d< fEpsilon)
                hr =0;

        return hr;
}
```

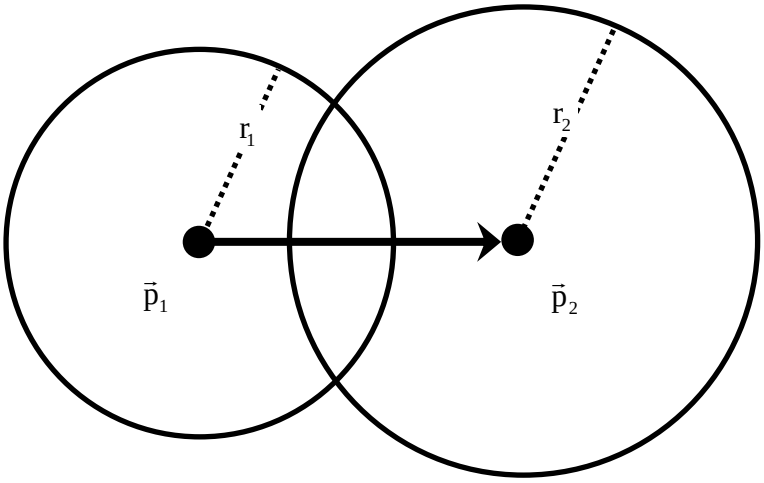
2.2 구 충돌

구와 관련된 충돌은 게임에서 충돌 판정에서 가장 많이 사용되는 방법입니다. 그것은 구의 충돌을 이용하는 것이 구현하기 쉽고, 성능이 우수하기 때문입니다. 세밀하게 충돌을 판정하는 경우에도 충돌 실행의 성능을 위해서 제일 먼저 구 충돌 먼저 시작합니다.

2.2.1 구와 구 충돌

구와 구의 충돌은 점과 점 충돌의 연장입니다. 만약 다음과 같이 중심이 P1, 반경이 r1인 구와 중심이 P2, 반경이 r2인 구가 서로 충돌한다면 다음과 같은 식을 만족합니다.

$$|\vec{p}_1 - \vec{p}_2| \le (r_1 + r_2)$$



이것을 의사 코드로 구현하면 다음과 같습니다.

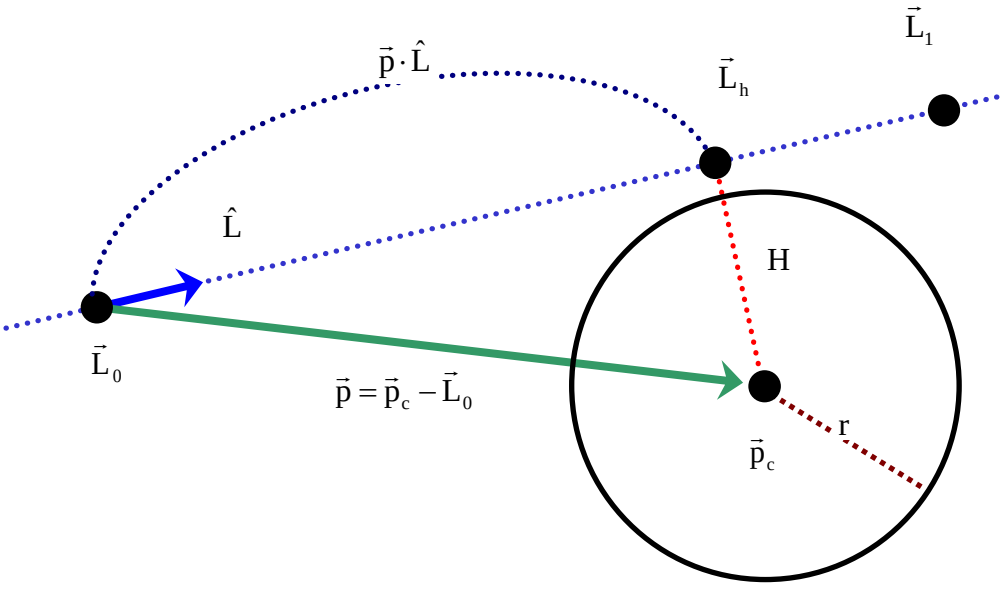
```
Vector3 vcT = P1 - P2;
Float fDistance = Length(vcT);

if( fDistance <= (r1 + r2))
        충돌;
```

DirectX SDK를 이용해서 구현 하면 다음과 같습니다.

```
INT CollisionSphereToSphere(const D3DXVECTOR3* SphereCenter1
                            , FLOAT sphereRadius1
```

```
                        , const D3DXVECTOR3* SphereCenter2
                        , FLOAT sphereRadius2 )
{
        INT hr=-1;

        FLOAT fDistance;
        D3DXVECTOR3 vcTemp = *SphereCenter1 - *SphereCenter2;

        fDistance = D3DXVec3Length(&vcTemp);

        if(fDistance <= (sphereRadius1 + sphereRadius2))
                hr =0;

        return hr;
}
```

2.2.2 구와 직선 충돌



구와 직선의 충돌은 직선이 무한한 경우, 유한한 경우 두 가지로 나누어서 진행합니다. 무한한 경우에는 구의 중심과 직선의 최단 거리와 구의 반경을 비교합니다. 만약 직선이 유한한 경우라면 앞서 구현한 무한 직선처럼 최단 거리와 구의 반경을 먼저 비교한 다음, 직선상에 있는 구와 최단

거리에 있는 위치가 유한 직선의 안 또는 밖에 대한 유무를 내적을 이용해서 구합니다.
무한 직선에 대한 구와 직선의 판정을 의사 코드로 구현 하면 다음과 같습니다.

```
Vector3 L0;     // 직선의 시작점
Vector3 L;      // 직선의 방향 벡터
Vector3 Pc;     // 구의 중심 위치
Float  r;       // 구의 반경

Vector3 p = Pc - L0;    // 구의 위치와 직선의 시작위치에 대한 상대적인 벡터를 구한다.
Float t = Dot(p, L);    // 상대적으로 구한 벡터에 대해서 직선의 방향에 대한 길이를 구한다.
Float H;

H = |p|^2 - t^2;        // 평가를 위한 H 값을 구한다.

//외적을 이용하는 경우 H 값을 다음과 같이 계산 할 수 있다.
{
        Vector3 t = |Cross(p, L)|;
        H = LengthSq(t);
}
// 연산 속도를 위해 제곱근을 구하지 않고, 구 반경을 제곱해서 H 값과 비교한다.
// ^은 승수를 의미. Ex) ^2는 제곱

if( H< r^2)
        충돌;
```

충돌 함수를 구현 하면 다음과 같습니다.

```
INT CollisionSphereToLine(const D3DXVECTOR3* SphereCenter
                        , FLOAT sphereRadius
                        , const D3DXVECTOR3* LineBegin
                        , const D3DXVECTOR3* LineDirection )
{
        INT hr=-1;

        FLOAT fHsq;
        D3DXVECTOR3 vcT = *SphereCenter - *LineBegin;
```

```
        D3DXVECTOR3 vcL = *LineDirection;

//      FLOAT fDot = D3DXVec3Dot(&vcT, &vcL);
//      fDot *= fDot;
//      fHsq =  D3DXVec3LengthSq(&vcT) -fDot;

        D3DXVECTOR3 vcH;
        D3DXVec3Cross(&vcH, &vcT, &vcL);
        fHsq = D3DXVec3LengthSq(&vcH);

        if( fHsq <= sphereRadius * sphereRadius)
                hr =0;

        return hr;
}
```

McCol_LineToSphere.zip

유한 직선의 경우, 직선의 양 끝점 중 하나가 구의 내부에 있는지 검사하는 것은 불필요합니다. 앞서 구한 무한 직선과 동일하게 진행 하되, 최단 거리의 점의 위치 Lh를 구해서 이 점이 유한 직선의 내부에 있는지 밖에 있는지 내적을 이용해서 판별하는 것이 더 빠릅니다.

이 방법을 의사 코드로 다음과 같이 구현 합니다.

```
Vector3 L0;     // 직선의 시작 점
Vector3 L1;     // 직선의 끝 점
Vector3 Pc;     // 구의 중심 위치
Float  r;       // 구의 반경

// 직선의 방향 벡터를 구한다.
Vector3 L = L1 - L2;
Normalize(L);

// 직선과 구의 최단 거리 위치 Lh를 구한다.
Vector3 p = Pc - L0;    // 구의 위치와 직선의 시작위치에 대한 상대적인 벡터를 구한다.
Float t = Dot(p, L);    // 상대적으로 구한 벡터에 대해서 직선의 방향에 대한 길이를 구한다.
```

```
Lh = t* L + L0;          // 최단 거리 위치 = 직선의 시작 지점 + 직선의 방향 벡터 * t

Float H = |p|^2 - t^2;

if( H< r^2)
{
        Vector3 t0 = Lh - L0;
        Vector3 t1 = Lh - L1;

        Float h = Dot(t0, t1);

        if( h<=0)
                충돌;
}
```

의사 코드를 보면 Lh 벡터에서 직선의 시작과 끝에 대한 상대적인 벡터 t0, t1를 구한 다음, 이 둘의 내적을 이용해서 충돌 판정을 내리는 것을 볼 수 있습니다. 이것은 만약 Lh 벡터가 직선의 내부에 있다면 직선의 양 끝에 대한 상대적인 벡터의 각도는 180도가 되므로 내적은 0보다 작거나 같은 값을 갖게 되기 때문입니다.
구현 코드는 다음과 같습니다.

```
INT CollisionSphereToFiniteLine(const D3DXVECTOR3* SphereCenter
                                , FLOAT sphereRadius
                                , const D3DXVECTOR3* LineBegin
                                , const D3DXVECTOR3* LineEnd )
{
        INT hr=-1;

        D3DXVECTOR3 vcT = *SphereCenter - *LineBegin;
        D3DXVECTOR3 vcL = *LineBegin - *LineEnd;
        D3DXVec3Normalize(&vcL, &vcL);

        FLOAT fDot = D3DXVec3Dot(&vcT, &vcL);
        D3DXVECTOR3 vcH = *LineBegin +  fDot * vcL;
        FLOAT fHsq =  D3DXVec3LengthSq(&vcT) - fDot * fDot;
```

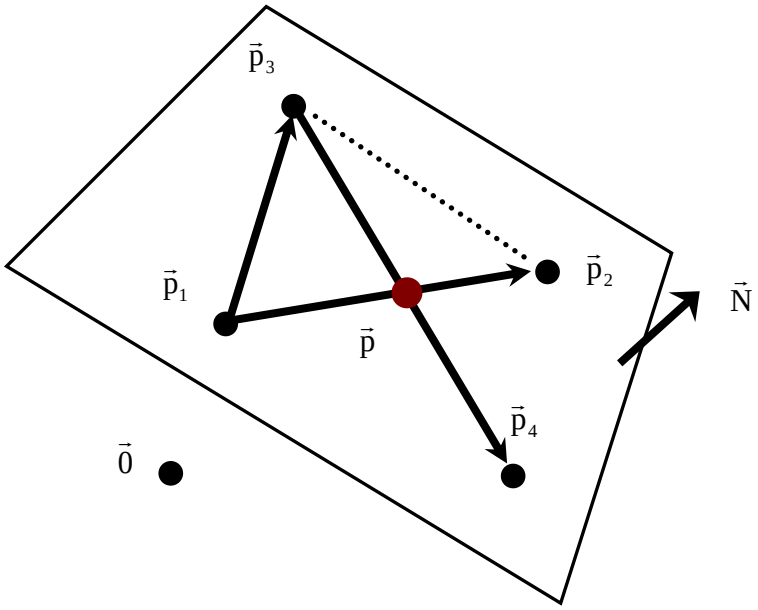
```
        if( fHsq <= sphereRadius * sphereRadius)
        {
                D3DXVECTOR3 t0 = vcH - *LineBegin;
                D3DXVECTOR3 t1 = vcH - *LineEnd;

                FLOAT t = D3DXVec3Dot(&t0, &t1);

                if(t <= 0.0)
                        hr = 0;
        }

        return hr;
}
```

2.3 직선과 충돌

직선 충돌에서 먼저 점과 직선의 충돌을 생각할 수 있는데 이것은 구와 직선의 충돌로 구할 수 있습니다. 점의 위치를 구의 위치로 변경하고, 적절한 Epsilon 값을 구의 반경으로 선택하게 된다면 직선과 점의 충돌이 됩니다. 다음으로 직선과 직선의 충돌입니다.

2.3.1 직선과 직선 충돌

직선과 직선의 충돌을 생각해보면 무한 직선과 무한 직선, 무한 직선과 유한 직선, 유한 직선과 유한 직선의 충돌 세 가지를 생각해 볼 수 있습니다. 그런데 이 충돌을 구현하기에 앞서 먼저 전제되어야 할 것은 충돌 판정을 내릴 두 직선이 모두 하나의 평면 위에 있어야 한다는 것입니다. 만약 같은 평면에 있지 않으면 무조건 충돌이 일어나지 않기 때문입니다.
여기서는 한 평면 위에 있는 두 무한 직선의 충돌을 먼저 구하고 다음으로 충돌 하지 않을 때 가장 가까운 두 지점을 구해 보겠습니다.
먼저 다음 그림과 같이 두 직선이 한 평면 위에 있는 경우 입니다.



이러한 경우 다음의 행렬식을 만족합니다.

$$\vec{L}_1 = \vec{p}_2 - \vec{p}_1,\ \vec{L}_2 = \vec{p}_4 - \vec{p}_3,\ \vec{L}_3 = \vec{p}_3 - \vec{p}_1$$

$$\vec{L}_1 \cdot \vec{L}_2 \times \vec{L}_3 = 0$$

또는

$$\begin{vmatrix} L_{1x} & L_{1y} & L_{1z} \\ L_{2x} & L_{2y} & L_{2z} \\ L_{3x} & L_{3y} & L_{3z} \end{vmatrix} = 0$$

두 직선에 의한 충돌 점 P는 두 직선 모두에 포함 되므로 다음 수식을 적용할 수 있습니다.

$$\vec{p} = \alpha\vec{L}_1 + \vec{p}_1$$
$$\vec{p} = \beta\vec{L}_2 + \vec{p}_3$$

여기서 α, β를 구하고 유한 직선의 경우 $0 \le \alpha \le 1$이고, $0 \le \beta \le 1$ 이면 두 직선은 충돌입니다.
α, β는 외적 또는 내적을 이용해서 구할 수 있는데 먼저 외적을 통해서 구해 보겠습니다.
다음과 같이 식을 하나로 만듭니다.

$$\alpha\vec{L}_1 + \vec{p}_1 = \beta\vec{L}_2 + \vec{p}_3$$

의 양변에 $\vec{L}_2$에 대해서 외적을 취합니다.

$$\vec{L}_2 \times (\alpha\vec{L}_1 + \vec{p}_1) = \vec{L}_2 \times (\beta\vec{L}_2 + \vec{p}_3)$$
$$\alpha\vec{L}_2 \times \vec{L}_1 + \vec{L}_2 \times \vec{p}_1 = \beta\vec{L}_2 \times \vec{L}_2 + \vec{L}_2 \times \vec{p}_3$$
$$\alpha\vec{L}_2 \times \vec{L}_1 + \vec{L}_2 \times \vec{p}_1 = \vec{L}_2 \times \vec{p}_3$$
$$\alpha\vec{L}_2 \times \vec{L}_1 = \vec{L}_2 \times (\vec{p}_3 - \vec{p}_1)$$
$$\alpha\vec{L}_2 \times \vec{L}_1 = \vec{L}_2 \times \vec{L}_3$$
$$\therefore \alpha = \frac{(\vec{L}_2 \times \vec{L}_1) \cdot (\vec{L}_2 \times \vec{L}_3)}{\| \vec{L}_2 \times \vec{L}_1 \|^2}$$

또한 양변에 $\vec{L}_1$에 대해서 외적을 취해서 다음과 같이 β를 구합니다.

$$\vec{L}_1 \times (\beta\vec{L}_2 + \vec{p}_3) = \vec{L}_1 \times (\alpha\vec{L}_1 + \vec{p}_1)$$
$$\beta\vec{L}_1 \times \vec{L}_2 + \vec{L}_1 \times \vec{p}_3 = \vec{L}_1 \times \vec{p}_1$$
$$\beta\vec{L}_1 \times \vec{L}_2 = \vec{L}_1 \times (\vec{p}_1 - \vec{p}_3)$$
$$\beta\vec{L}_1 \times \vec{L}_2 = -\vec{L}_1 \times \vec{L}_3$$
$$\beta\vec{L}_2 \times \vec{L}_1 = \vec{L}_1 \times \vec{L}_3$$
$$\therefore \beta = \frac{(\vec{L}_2 \times \vec{L}_1) \cdot (\vec{L}_1 \times \vec{L}_3)}{\| \vec{L}_2 \times \vec{L}_1 \|^2}$$

α, β를 구했으므로 $0 \le \alpha \le 1$이고, $0 \le \beta \le 1$ 인지 확인합니다.

외적에 대해서 많이 어려움을 느낀다면 내적을 이용할 수도 있습니다.

$$\alpha\vec{L}_1 + \vec{p}_1 = \beta\vec{L}_2 + \vec{p}_3$$

식에 양변에 $\vec{L}_1$ 벡터로 내적을 수행합니다.

$$\vec{L}_1 \cdot (\alpha\vec{L}_1 + \vec{p}_1) = \vec{L}_1 \cdot (\beta\vec{L}_2 + \vec{p}_3)$$
$$\alpha\vec{L}_1 \cdot \vec{L}_1 + \vec{L}_1 \cdot \vec{p}_1 = \beta\vec{L}_1 \cdot \vec{L}_2 + \vec{L}_1 \cdot \vec{p}_3$$
$$\alpha\vec{L}_1 \cdot \vec{L}_1 - \beta\vec{L}_1 \cdot \vec{L}_2 = \vec{L}_1 \cdot \vec{p}_3 - \vec{L}_1 \cdot \vec{p}_1$$
$$\alpha\vec{L}_1 \cdot \vec{L}_1 - \beta\vec{L}_1 \cdot \vec{L}_2 = \vec{L}_1 \cdot (\vec{p}_3 - \vec{p}_1)$$
$$\alpha\vec{L}_1 \cdot \vec{L}_1 - \beta\vec{L}_1 \cdot \vec{L}_2 = \vec{L}_1 \cdot \vec{L}_3$$

또한 $\vec{L}_2$ 벡터로 내적을 수행합니다.

$$\vec{L}_2 \cdot (\alpha\vec{L}_1 + \vec{p}_1) = \vec{L}_2 \cdot (\beta\vec{L}_2 + \vec{p}_3)$$
$$\alpha\vec{L}_2 \cdot \vec{L}_1 + \vec{L}_2 \cdot \vec{p}_1 = \beta\vec{L}_2 \cdot \vec{L}_2 + \vec{L}_2 \cdot \vec{p}_3$$
$$\alpha\vec{L}_1 \cdot \vec{L}_2 - \beta\vec{L}_2 \cdot \vec{L}_2 = \vec{L}_2 \cdot \vec{p}_3 - \vec{L}_2 \cdot \vec{p}_1$$
$$\alpha\vec{L}_1 \cdot \vec{L}_2 - \beta\vec{L}_2 \cdot \vec{L}_2 = \vec{L}_2 \cdot \vec{L}_3$$

식과 미지 항 α, β에 대해서 다항식이 2개 있으므로 연립 방정식을 만들 수 있고 행렬로 변환할 수 있습니다.

$$\alpha\vec{L}_1 \cdot \vec{L}_1 - \beta\vec{L}_1 \cdot \vec{L}_2 = \vec{L}_1 \cdot \vec{L}_3$$

$$\alpha\vec{L}_1 \cdot \vec{L}_2 - \beta\vec{L}_2 \cdot \vec{L}_2 = \vec{L}_2 \cdot \vec{L}_3$$

$$\begin{pmatrix} \vec{L}_1 \cdot \vec{L}_1 & -\vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & -\vec{L}_2 \cdot \vec{L}_2 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \vec{L}_1 \cdot \vec{L}_3 \\ \vec{L}_2 \cdot \vec{L}_3 \end{pmatrix}$$

위의 연립방정식을 크라머 법칙을 이용해서 행렬식으로 α, β를 구하면 다음과 같습니다.

$$\alpha = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_3 & -\vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_2 \cdot \vec{L}_3 & -\vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}}{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & -\vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & -\vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}} = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_2 \cdot \vec{L}_3 & \vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}}{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & \vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & \vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}}$$

$$\beta = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & \vec{L}_1 \cdot \vec{L}_3 \\ \vec{L}_1 \cdot \vec{L}_2 & \vec{L}_2 \cdot \vec{L}_3 \end{vmatrix}}{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & -\vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & -\vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}} = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_1 \\ \vec{L}_2 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_2 \end{vmatrix}}{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & \vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & \vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}}$$

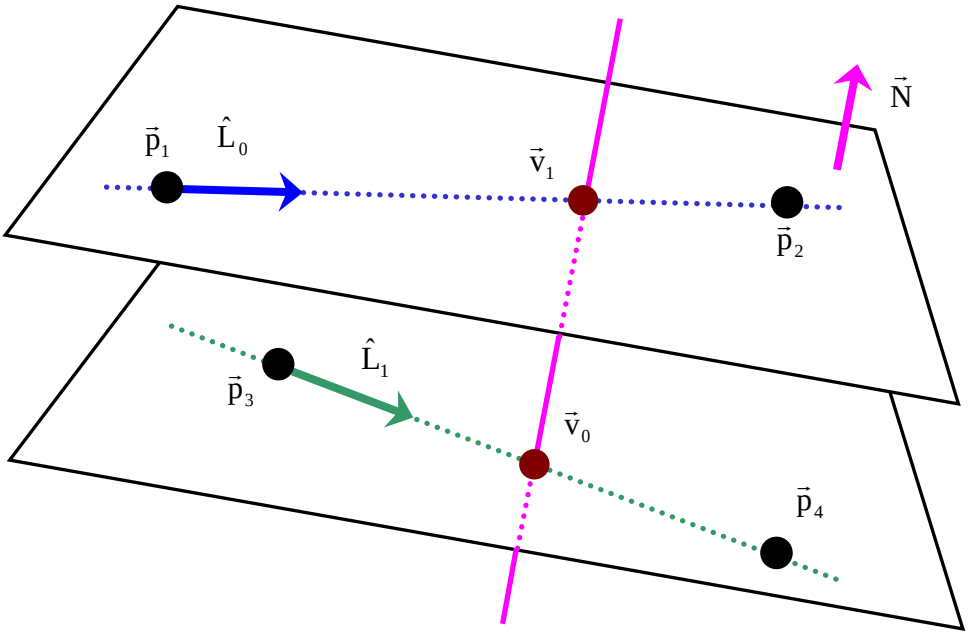
외적과 마찬가지로 $0 \le \alpha \le 1$이고, $0 \le \beta \le 1$ 이면 충돌입니다.

만약 두 직선이 무한 직선이고 $\vec{L}_1$, $\vec{L}_2$ 벡터가 단위벡터로 직선의 방향벡터라면 α, β중에 하나만 구해도 됩니다. 또한 내적을 통해서 구하는 방법이 더욱 간단해지는데 이 때 β는 다음과 같아집니다.

$$\beta = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_1 \\ \vec{L}_2 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_2 \end{vmatrix}}{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_1 & \vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & \vec{L}_2 \cdot \vec{L}_2 \end{vmatrix}} = \frac{\begin{vmatrix} \vec{L}_1 \cdot \vec{L}_3 & 1 \\ \vec{L}_2 \cdot \vec{L}_3 & \vec{L}_1 \cdot \vec{L}_2 \end{vmatrix}}{\begin{vmatrix} 1 & \vec{L}_1 \cdot \vec{L}_2 \\ \vec{L}_1 \cdot \vec{L}_2 & 1 \end{vmatrix}} = \frac{(\vec{L}_1 \cdot \vec{L}_3)(\vec{L}_1 \cdot \vec{L}_2) - \vec{L}_2 \cdot \vec{L}_3}{1 - (\vec{L}_1 \cdot \vec{L}_2)^2}$$

만약 두 직선이 충돌하지 않는 다면 두 직선의 최단 거리의 두 지점을 구해야 합니다. 여기서는 다음 그림과 같이 두 무한 직선이 충돌하지 않는 경우에 최단 거리와 최 근접 점을 구해보겠습니

다.



두 무한 직선이 충돌하지 않는 경우는 평면의 법선 벡터가 같은 평행한 평면 위에 각각 두 직선이 놓여 있는 것과 동일합니다. 그러므로 두 직선의 최단 거리는 곧, 두 평면의 거리와 같습니다. 따라서 간단하게 다음과 같은 공식으로 풀립니다.

$D_1 = \vec{N} \cdot \vec{p}_1$, $D_2 = \vec{N} \cdot \vec{p}_3$, $D = D_1 - D_2$ 또는 $D = \vec{N} \cdot (\vec{p}_1 - \vec{p}_3)$

최단 거리 = $|D|$

평면이 법선 벡터는 $\vec{L}_1$와 $\vec{L}_2$이 외적으로 구합니다.

$$\vec{N} = \frac{\hat{L}_0 \times \hat{L}_1}{|\hat{L}_0 \times \hat{L}_1|}$$

최 근접 점은 두 직선 중 하나를 법선 벡터를 따라 이동 시켜 한 평면 위에 있는 경우와 같은 방법으로 충돌 점 하나를 구한 다음, 다시 이 점에 법선 벡터와 D 값을 곱해서 다른 위치를 구합니다.

$$\vec{p}_1' = \vec{p}_1 - D * \vec{N}$$

$$\vec{L}_3' = \vec{p}_3 - \vec{p}_1'$$

$$\beta = \frac{(\vec{L}_1 \cdot \vec{L}_3')(\vec{L}_1 \cdot \vec{L}_2) - \vec{L}_2 \cdot \vec{L}_3'}{1 - (\vec{L}_1 \cdot \vec{L}_2)^2}$$

$$\vec{v}_0 = \beta\vec{L}_2 + \vec{p}_3$$

$$\vec{v}_1 = \vec{v}_0 + D*\vec{N}$$

내용은 수식 전개가 대부분이라 상당히 길었는데 정리를 하면 다음과 같습니다.

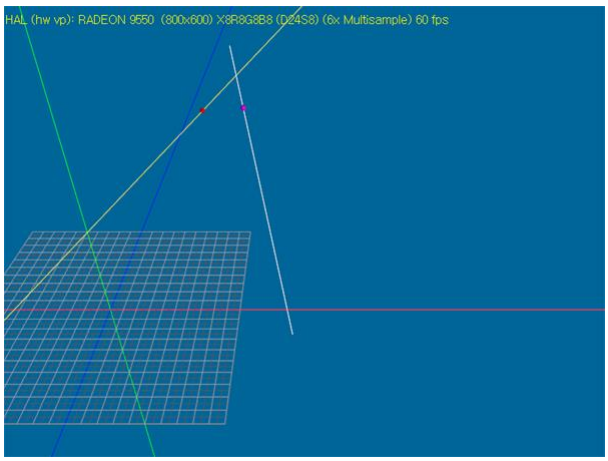
1. 두 직선이 이루는 평면의 법선 벡터를 구한다.
2. 법선 벡터를 이용해서 D를 구한다.
3. 두 직선 중 하나를 선택해서 직선의 시작 지점을 법선 벡터의 반대 방향으로 D를 곱한 만큼 이동한다.
4. β를 구하고, 충돌 지점 하나를 먼저 구한다.
5. 4에서 구한 충돌 점으로부터 평면의 법선벡터 방향으로 D만큼 이동한 또 하나의 점을 구해 두 직선의 최 근접 점을 구한다.

두 직선의 법선 벡터와 최단 거리를 구하는 함수, 그리고 충돌 자체를 판정하는 함수로 나누는 것이 구현하기 쉽습니다. 다음은 무한 직선에 대한 충돌을 구현한 코드 입니다.

```
FLOAT   LcMath_LineToLineDistance(D3DXVECTOR3* pOut      // Normal vector
                                , const LcLine* pV1      // Input Line1
                                , const LcLine* pV2      // Input Line1
                                )
{
        D3DXVECTOR3 vcN;
        D3DXVECTOR3 L1 = pV1->t;
        D3DXVECTOR3 L2 = pV2->t;
        D3DXVECTOR3 L3 = pV1->p - pV2->p;

        FLOAT    fD = 0.f;

        D3DXVec3Cross(&vcN, &L1, &L2);
        D3DXVec3Normalize(&vcN, &vcN);

        fD = D3DXVec3Dot(&vcN, &L3);

        *pOut = vcN;

        return fD;
```

```
}

INT LcMath_LineToLineIntersection(D3DXVECTOR3* pOut      // Intersection point
                                  , const LcLine* pV1    // Input Line1
                                  , const LcLine* pV2    // Input Line1
                                  )
{
        D3DXVECTOR3 vcN;
        D3DXVECTOR3 L1 = pV1->t;
        D3DXVECTOR3 L2 = pV2->t;
        D3DXVECTOR3 P1 = pV1->p;
        D3DXVECTOR3 P3 = pV2->p;
        D3DXVECTOR3 L3 = P3 - P1;

        FLOAT   fD = LcMath_LineToLineDistance(&vcN, pV1, pV2);

        FLOAT   DotL12 = D3DXVec3Dot(&L1, &L2);
        FLOAT   DotL13 = D3DXVec3Dot(&L1, &L3);
        FLOAT   DotL23 = D3DXVec3Dot(&L2, &L3);

        FLOAT   fDet1 = -1 + DotL12 * DotL12;
        FLOAT   fDet2 = DotL23 - DotL12 * DotL13;
        FLOAT   fBeta = 0.f;

        // 충돌
        if(fabsf(fD)<0.0001f)
        {
                // 부정(정할 수 없다.)
                if(fabsf(fDet2)<0.0001f)
                        return -1;

                fBeta = fDet2/ fDet1;
                pOut[0] = P3 + fBeta * L2;
                pOut[1] = pOut[0];

                // 충돌한 점의 개수를 돌려준다.
                return 1;
```

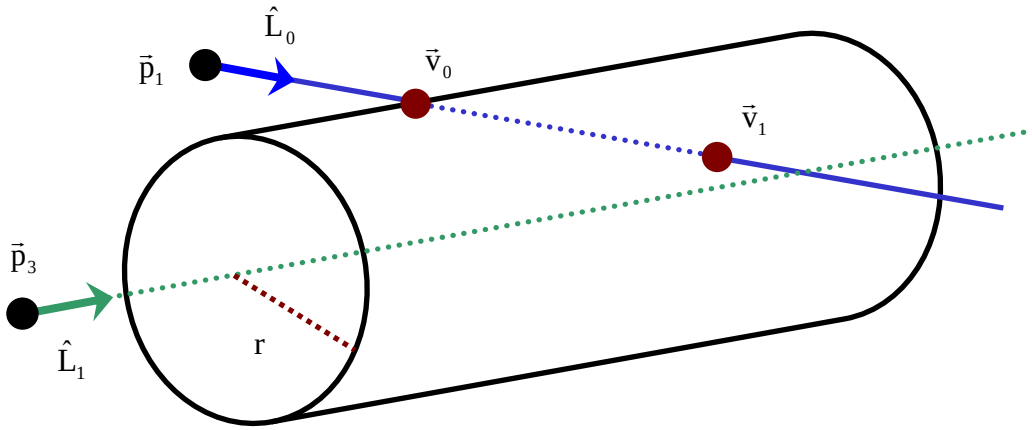
```
        }

        // 충돌이 아닌데 평행인 경우
        // 불능(구할 수 없다.)
        if( fabsf(DotL12)>0.9999f)
                return -2;

        P1 = P1 - fD * vcN;
        L3 = P3 - P1;

        DotL13 = D3DXVec3Dot(&L1, &L3);
        DotL23 = D3DXVec3Dot(&L2, &L3);

        fDet2 = DotL23 - DotL12 * DotL13;

        fBeta = fDet2/ fDet1;
        pOut[0] = P3 +  fBeta * L2;
        pOut[1] = pOut[0] +  fD * vcN;

        // 충돌한 점의 개수를 돌려준다.
        return 2;
}
```



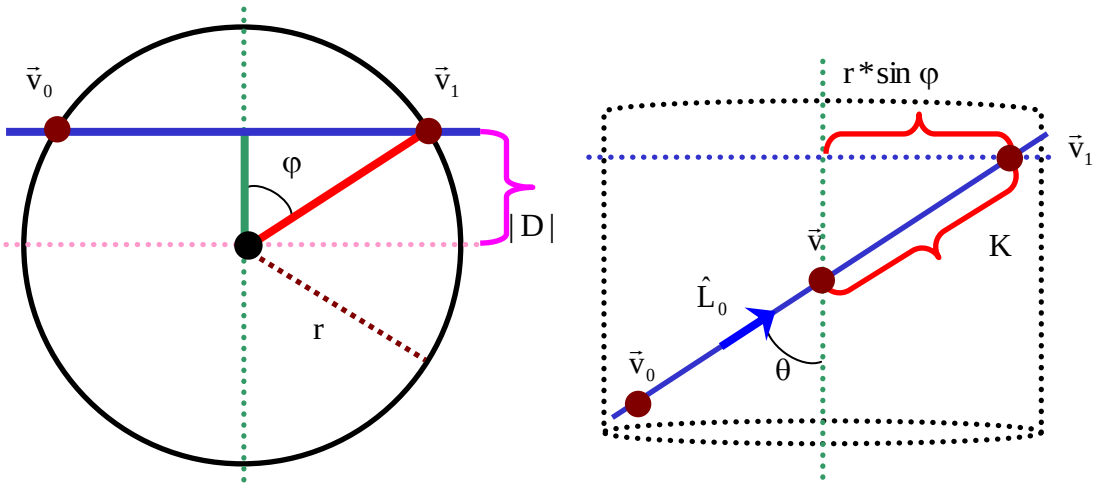
McCol_LineToLine.zip

2.3.2 직선과 원통 충돌

직선과 직선의 충돌 중에 직선과 원통 충돌이 있습니다. 잘 사용하지 않지만 간혹 사용할 때가 있으므로 이를 구현해 보겠습니다. 직선과 원통 충돌은 무한 직선과 무한 직선의 충돌에서 시작을 합니다. 간단히 생각하면 두 직선의 최단 거리가 원통의 반지름 보다 작거나 같으면 충돌입니다. 다음 그림과 같은 경우에 충돌을 구해 봅시다.



이것을 원통의 앞, 위에서 보면 다음과 같이 됩니다.



순서대로 평면의 법선 벡터, 최 근접 점, 그리고 K값 구해 최종 충돌 두 지점을 결정하는데 법선 벡터를 만들 때 sinΘ 값을 얻어 낼 수 있습니다.

$$\sin\theta = |\hat{L}_0 \times \hat{L}_1|, \quad \vec{N} = \frac{\hat{L}_0 \times \hat{L}_1}{|\hat{L}_0 \times \hat{L}_1|}$$

K 값은 φ를 이용해서 다음과 같이 구할 수 있습니다.

$$\varphi = \cos^{-1}\frac{D}{r}$$

$$\sin\theta = \frac{r\sin\varphi}{K}$$

$$\therefore K = r * \left|\frac{\sin\varphi}{\sin\theta}\right|$$

각도를 반환하는 acos() 사용을 피하고 연산을 줄이기 위해 위의 식을 좀 더 정리해서 K 값을 다음과 같이 구하는 것이 좋습니다.

$$K = \frac{r * \sqrt{1-(\frac{D}{r})^2}}{|\sin\theta|} \quad \therefore K = \frac{\sqrt{r^2 - D^2}}{|\sin\theta|}$$

두 지점 $\vec{v}_0$, $\vec{v}_1$을 구하기 위해서 직선에 있는 원통의 최 근접 점 $\vec{v}$을 구합니다.

$$D = \vec{N} \cdot (\vec{p}_1 - \vec{p}_3)$$

$$\vec{p}_1' = \vec{p}_1 - D * \vec{N}$$

$$\vec{L}_3' = \vec{p}_3 - \vec{p}_1'$$

$$\beta = \frac{(\vec{L}_1 \cdot \vec{L}_3')(\vec{L}_1 \cdot \vec{L}_2) - \vec{L}_2 \cdot \vec{L}_3'}{1-(\vec{L}_1 \cdot \vec{L}_2)^2}$$

$$\vec{v} = \beta\vec{L}_2 + \vec{p}_3 + D * \vec{N}$$

$\vec{v}$ 을 구했다면 직선의 방향 벡터를 이용해서 다음과 같이 원통과 직선에 충돌하는 두 지점 $\vec{v}_0$, $\vec{v}_1$을 결정합니다.

$$\vec{v}_0 = \vec{v} + K * \hat{L}_0$$

$$\vec{v}_1 = \vec{v} - K * \hat{L}_0$$

두 직선의 각도는 외적을 구할 때 얻을 수 있으므로 앞서 구현한 무한 직선 충돌 함수 중에서 최단 거리를 구하는 함수는 다음과 같이 법선 벡터와 각도에 대한 Sin(Θ) 값을 얻도록 함수의 인수와 내용을 수정합니다.

```
FLOAT LcMath_LineToLineDistance(FLOAT* pSinTheta// Sin(Θ)
                                , D3DXVECTOR3* pOut      // Normal vector
                                , const LcLine* pV1      // Input Line1
                                , const LcLine* pV2      // Input Line2
                                )
{
        ...
        FLOAT    fD = 0.f;
        FLOAT    fL = 0.f;

        D3DXVec3Cross(&vcN, &L1, &L2);

        fL = D3DXVec3Length(&vcN);

        if(pSinTheta)
                *pSinTheta = fL;

        fL = 1.f/fL;

        vcN.x *= fL;
        vcN.y *= fL;
        vcN.z *= fL;

        fD = D3DXVec3Dot(&vcN, &L3);
        ...
}
```

최 근접 점을 구하는 함수도 역시 Sin(Θ) 에 대해서 수정해야 하는데 Sin(Θ) 처리는 법선 벡터를 구하는 함수가 처리하므로 인수 값만 전달하도록 합니다.

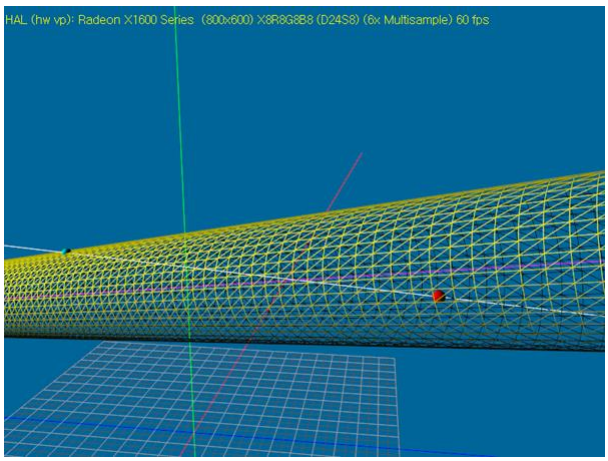
```
INT LcMath_LineToLineIntersection(FLOAT* pSinTheta       // Sin(Θ)
                                , FLOAT* pDst            // Distance
```

```
                                , D3DXVECTOR3* pOut       // Intersection point
                                , const LcLine* pV1       // Input Line1
                                , const LcLine* pV2       // Input Line2
                                  )
{
        ...
        fD = LcMath_LineToLineDistance(pSinTheta, &vcN, pV1, pV2);
        ...
}
```

원통과 충돌하는 함수는 최 근접 점을 구하는 함수도 역시 sinΘ 에 대해서 수정해야 하는데 sinΘ 처리는 법선 벡터를 구하는 함수가 처리하므로 인수 값만 전달하도록 합니다.

```
INT LcMath_LineToCylinderIntersection(D3DXVECTOR3* pOut          // Intersection point
                                    , const LcLine* pV1          // Input Line
                                    , const LnCylinder* pV2      // Input Cylinder
                                   )
{
        INT iInsc=0;

        D3DXVECTOR3 vcInsc[2];                  // 실린더 중심선과 직선이 교차하는 지점
        D3DXVECTOR3     L1 = pV1->t;
        D3DXVECTOR3     L2 = pV2->t;

        FLOAT   fR = pV2->r;                    // 실린더의 반지름
        FLOAT   fD = 0.f;                       // 실린더와 직선의 거리
        FLOAT   fK = 0.f;
        FLOAT   fSinTheta = 0.f;

        iInsc = LcMath_LineToLineIntersection(&fSinTheta, &fD, vcInsc, pV1, (LcLine*)pV2);

        // 부정이나 불능 두 가지 중 하나
        if(FAILED(iInsc))
        {
                // 원통의 반경과 비교
                // 직선이 원통 안에 완전히 존재
                if(fD <= pV2->r)
```

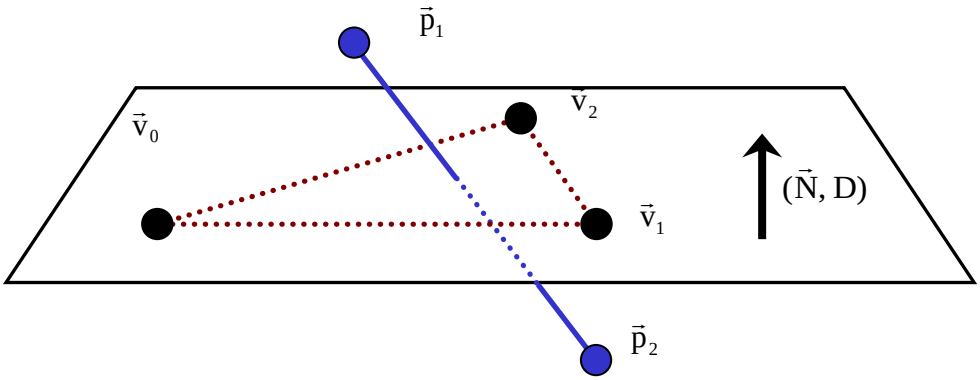
```
                return 0;       // 충돌 점을 정할 수가 없음

        // 직선이 원통 밖에 존재
        else
                return -1;
    }

    // 직선이 원통 밖에 존재
    if(fabsf(fD)>=fR)
        return -1;

    // 충돌한 두 점의 위치를 구하기 위해 K 값을 결정
    fK = fabsf(fSinTheta);
    fK = sqrtf(fR*fR - fD*fD)/fK;

    // vcInsc[0]은 실린더 쪽 최 근접 점
    // vcInsc[1]은 직선 쪽  최 근접 점
    pOut[0] = vcInsc[1] - fK * L1;
    pOut[1] = vcInsc[1] + fK * L1;

    return 2;
}
```



McCol_LineToCylinder.zip

2.3.3 유한 직선과 평면의 충돌

다음 그림과 같이 유한 직선과 평면의 충돌에서 직선의 양 끝점의 위치가 동시에 평면 위에 있거나 동시에 밑에 있으면 평면의 수직벡터와의 내적 결과를 곱한 값은 항상 0보다 큰 값을 갖게 되므로 0보다 작거나 같은 경우에 유한 직선과 평면은 충돌 하게 됩니다.



만약 평면의 방정식이 주어진다면 $(\vec{N}\cdot\vec{p}_1+D)*(\vec{N}\cdot\vec{p}_2+D)\le 0$ 이면 충돌이 되어 다음과 같이 간단하게 구현할 수 있습니다.

```
INT LcMath_CollisionLineToPlane(const D3DXPLANE* pPlane, const D3DXVECTOR3* pLine)
{
	FLOAT D1 = D3DXPlaneDotCoord(pPlane, &pLine[0]);
	FLOAT D2 = D3DXPlaneDotCoord(pPlane, &pLine[1]);

	// Collision
	if(D1 * D2<= 0)
		return 0;

	return -1;
}
```

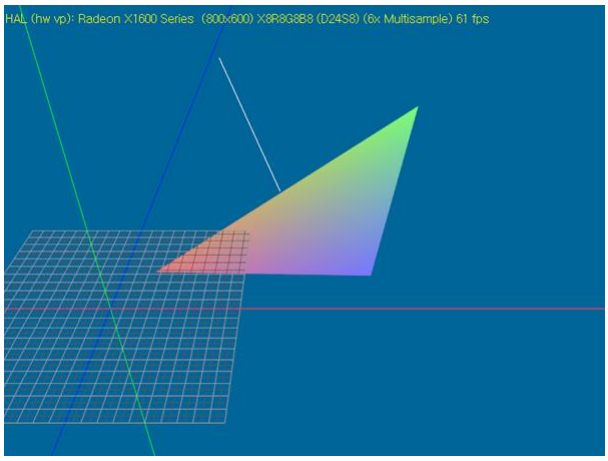
D3DXPlaneDotCoord() 함수는 내적 $\vec{N}\cdot\vec{p}+D$ 를 구하는 DirectX SDK 함수 입니다

평면의 방정식이 주어지지 않고 세 점이 주어진다면 평면의 방정식을 만들어야 합니다. 다음과 같이 법선 벡터와 D 값을 계산합니다.

$$\vec{N} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0), \quad D = \vec{N} \cdot \vec{v}_0$$

그리고 앞의 방법을 그대로 적용해서 다음과 같이 구현합니다.

```
INT LcMath_CollisionLineToPlane(const D3DXVECTOR3* pTri, const D3DXVECTOR3* pLine)
{
	D3DXVECTOR3 N;
	FLOAT		D;

	D3DXVECTOR3 vcA = pTri[1] - pTri[0];
	D3DXVECTOR3 vcB = pTri[2] - pTri[0];

	D3DXVec3Cross(&N, &vcA, &vcB);
	D = - D3DXVec3Dot(&N, &pTri[0]);

	FLOAT D1 = D3DXVec3Dot(&N, &pLine[0]) + D;
	FLOAT D2 = D3DXVec3Dot(&N, &pLine[1]) + D;

	// Collision
	if(D1 * D2<= 0)
		return 0;

	return -1;
}
```

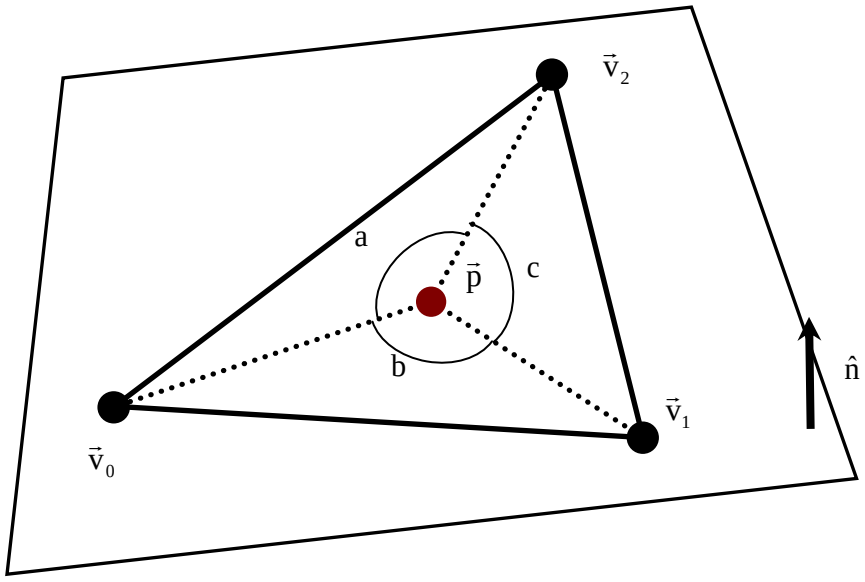


McCol_LineToPlane1.zip

2.4 삼각형 충돌

2.4.1 점과 삼각형의 충돌

한 평면 위에 있는 점과 삼각형의 충돌은 여러 가지 방법을 동원해서 가장 재미있게 풀 수 있는 문제 입니다. 가장 쉬운 방법은 각도를 이용하는 방법입니다.



먼저 점 P가 평면에 있는지 평면의 법선 벡터와 내적을 이용합니다.

$$\vec{N} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$$

$$\hat{n} = \frac{\vec{N}}{\|\vec{N}\|}$$

만약 $|\vec{N} \cdot (\vec{p} - \vec{v}_0)| < \text{Epsilon}$ 이면 같은 평면 안에 있지 않기 때문에 충돌 하지 않는 것으로 처리 합니다. 평면과 충돌 한다면 각도 a, b, c를 다음과 같이 구합니다.

$$a = \cos^{-1}\left(\frac{(\vec{p}-\vec{v}_0)\cdot(\vec{p}-\vec{v}_2)}{\|\vec{p}-\vec{v}_0\|\|\vec{p}-\vec{v}_2\|}\right)$$

$$b = \cos^{-1}\left(\frac{(\vec{p}-\vec{v}_1)\cdot(\vec{p}-\vec{v}_0)}{\|\vec{p}-\vec{v}_1\|\|\vec{p}-\vec{v}_0\|}\right)$$

$$c = \cos^{-1}\left(\frac{(\vec{p}-\vec{v}_2)\cdot(\vec{p}-\vec{v}_1)}{\|\vec{p}-\vec{v}_2\|\|\vec{p}-\vec{v}_1\|}\right)$$

$a+b+c=2\pi$ 이면 점은 삼각형 내부에 있으므로 충돌 입니다.

만약 평면과 삼각형이 충돌한다고 가정하고 코드로 구현하면 다음과 같습니다.

```
FLOAT LcMath_DotAngle(const D3DXVECTOR3* p0, const D3DXVECTOR3* p1, const D3DXVECTOR3* p2)
{
	D3DXVECTOR3 vcA = *p1 - *p0;
	D3DXVECTOR3 vcB = *p2 - *p0;

	D3DXVec3Normalize(&vcA,&vcA);
	D3DXVec3Normalize(&vcB,&vcB);

	FLOAT fDot = D3DXVec3Dot(&vcA, &vcB);

	if(fDot>0.9999f)
		return 0.f;
	else if(fDot<-0.9999f)
		return D3DX_PI;

	fDot = acosf(fDot);

	return fDot;
}

BOOL LcMath_CollisionPointToTriangle(D3DXVECTOR3* vcOut
				, const D3DXVECTOR3* pTri
				, const D3DXVECTOR3* pPoint)
{
	FLOAT	fA;
```

```
	FLOAT	fB;
	FLOAT	fC;

	fA = LcMath_DotAngle(pPoint, &pTri[0], &pTri[1]);
	fB = LcMath_DotAngle(pPoint, &pTri[1], &pTri[2]);
	fC = LcMath_DotAngle(pPoint, &pTri[2], &pTri[0]);

	if(vcOut)
		*vcOut = *pPoint;

	if(fA+ fB+ fC>=(D3DX_PI*2-0.001f))
		return 0;

	return -1;
}
```

acos() 함수는 -1보다 작거나 1 보다 클 경우 문제가 발생 합니다. 입력 값이 -0.9999 정도 보다 적은 값이 입력되면 각도를 π로 정하고 0.9999 보다 크면 0도로 처리하도록 합니다.

McCol_PointToTri1.zip

각도를 이용하는 방법 이외에 면적을 이용하는 것도 고려할 수 있습니다. 만약 점이 삼각형의 밖에 있다면 삼각형의 정점과 점이 이루는 또 다른 삼각형의 면적은 항상 커야 합니다. 그런데 여기에 면적을 처리하는 방법을 구현해야 하는데 이 것은 외적을 이용합니다. 외적은 기하학적으로 두 벡터가 이루는 평행사변형의 면적을 나타냅니다.
각각의 면적은 다음과 같이 구할 수 있습니다.

$$A = |(\vec{p} - \vec{v}_2) \times (\vec{p} - \vec{v}_0)|$$
$$B = |(\vec{p} - \vec{v}_0) \times (\vec{p} - \vec{v}_1)|$$
$$C = |(\vec{p} - \vec{v}_1) \times (\vec{p} - \vec{v}_2)|$$
$$T = |(\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)|$$

만약 |A - S| < Epsilon이면 삼각형 안에 점이 존재 하게 됩니다.

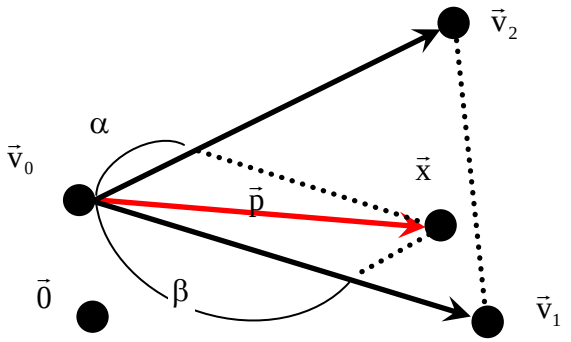
```
INT LcMath_CollisionPointToTriangle(D3DXVECTOR3* pTri, D3DXVECTOR3* pPoint)
```

```
{
	FLOAT	fT;			// Triangle Dimension
	FLOAT	fA;			// A Dimension
	FLOAT	fB;			// B Dimension
	FLOAT	fC;			// C Dimension

	D3DXVECTOR3	vcA = pTri[1] - pTri[0];
	D3DXVECTOR3	vcB = pTri[2] - pTri[0];

	D3DXVECTOR3	vcT;

	D3DXVec3Cross(&vcT, &vcA, &vcB);
	fT = D3DXVec3Length(&vcT);

	vcA = *pPoint - pTri[0];
	vcB = *pPoint - pTri[1];
	D3DXVec3Cross(&vcT, &vcA, &vcB);
	fA = D3DXVec3Length(&vcT);

	vcA = *pPoint - pTri[1];
	vcB = *pPoint - pTri[2];
	D3DXVec3Cross(&vcT, &vcA, &vcB);
	fB = D3DXVec3Length(&vcT);

	vcA = *pPoint - pTri[2];
	vcB = *pPoint - pTri[0];
	D3DXVec3Cross(&vcT, &vcA, &vcB);
	fC = D3DXVec3Length(&vcT);

	if( absf(fA+fB+fC-fT) < 0.0001f)
		return 0;

	return -1;
}
```

McCol_PointToTri2.zip

점과 삼각형의 충돌은 삼각형과 어떤 한 점이 같은 평면 위에 있을 때 이 점이 삼각형의 내부에 있는지 또는 외부에 있는지 판정하는 문제입니다. 그런데 위의 두 방법은 acos() 함수 또는 sqrt() 함수를 호출을 피할 수는 없습니다.
다음으로 보여줄 방법은 삼각형의 두 변을 벡터로 보고 충돌 시험을 할 점을 이 벡터의 크기로 분해 하는 방법입니다. 이 방법은 내부의 구현이 모두 곱셈으로 구성되어 있고, 대부분의 게임 프로그램에서 가장 많이 사용되는 방법입니다. 이 방법을 사용하면 부가적인 정보를 얻을 수 있는 장점이 있어서 DirectX SDK 에서도 D3DXIntersectTri() 함수로 구현되어 있습니다.

다음 그림과 같이 충돌 시험할 점을 삼각형을 구성하는 세 점으로 구성된 두 벡터로 분해해 보도록 합시다.
삼각형의 두 벡터로 분해된 크기를 α, β 라 하면 α≥0 && β≥0 && (α+β) ≤ 1 의 조건인 경우에 점은 삼각형에 충돌하게 됩니다.



다음과 같이 임시 벡터를 만들고 점 p를 α, β 로 표현 한다면 다음과 같은 수식을 만들 수 있습니다.

$$\vec{A} = (\vec{v}_1 - \vec{v}_0),\ \vec{B} = (\vec{v}_2 - \vec{v}_0),\ \vec{p} = \vec{X} - \vec{v}_0$$

$$\vec{p} = \alpha\vec{A} + \beta\vec{B}$$

α, β를 구하기 위해 점 p와 우변에 A, B 벡터의 내적 연산을 하고 이를 연립방정식으로 다음과 같이 바꾸어 놓는다.

$$\vec{A}\cdot\vec{p} = \alpha\vec{A}\cdot\vec{A} + \beta\vec{A}\cdot\vec{B}$$

$$\vec{B}\cdot\vec{p} = \alpha\vec{A}\cdot\vec{B} + \beta\vec{B}\cdot\vec{B}$$

$$\begin{pmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{B} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{B} \end{pmatrix}\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \vec{A}\cdot\vec{p} \\ \vec{B}\cdot\vec{p} \end{pmatrix}$$

충돌 조건은 앞서 이야기 한 바와 같이 $0 \le \alpha,\ 0 \le \beta,\ \alpha+\beta \le 1$ 이며 크라머 법칙(Cramer's Rule)을 이용하면 답을 구할 수 있는데 나눗셈을 피하도록 수식을 구성합니다.

$$\alpha = \frac{\begin{vmatrix} \vec{A}\cdot\vec{p} & \vec{A}\cdot\vec{B} \\ \vec{B}\cdot\vec{p} & \vec{B}\cdot\vec{B} \end{vmatrix}}{\begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{B} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{B} \end{vmatrix}} \ge 0 \ \Rightarrow\ \begin{vmatrix} \vec{A}\cdot\vec{p} & \vec{A}\cdot\vec{B} \\ \vec{B}\cdot\vec{p} & \vec{B}\cdot\vec{B} \end{vmatrix} \ge 0 \qquad (2\text{-}1)$$

$$\beta = \frac{\begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{p} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{p} \end{vmatrix}}{\begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{B} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{B} \end{vmatrix}} \ge 0 \ \Rightarrow\ \begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{p} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{p} \end{vmatrix} \ge 0 \qquad (2\text{-}2)$$

$$\alpha+\beta \le 1 \Rightarrow \begin{vmatrix} \vec{A}\cdot\vec{p} & \vec{A}\cdot\vec{B} \\ \vec{B}\cdot\vec{p} & \vec{B}\cdot\vec{B} \end{vmatrix} + \begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{p} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{p} \end{vmatrix} \le \begin{vmatrix} \vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{B} \\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{B} \end{vmatrix} \qquad (2\text{-}3)$$

충돌을 검사하기 위해 총 5번의 내적 $\vec{A}\cdot\vec{A}, \vec{B}\cdot\vec{B}, \vec{A}\cdot\vec{B}, \vec{A}\cdot\vec{p}, \vec{B}\cdot\vec{p}$ 과 3번의 2X2 행렬식 계산이 필요한데 행렬식 계산도 곱셈이므로 이 방법은 전부가 곱셈으로 구성되어 있다고 볼 수 있습니다. 따라서 앞서 보여준 각도나 면적을 이용하는 방법보다 성능이 좋고, 또한 α, β를 얻을 수 있고 이것으로 점 p를 표현할 수 있습니다. 또한 α, β는 텍스처의 u, v 좌표 등으로 활용이 가능합니다.
앞의 두 방법에 대한 코드를 다음과 같이 변경할 수 있습니다.

```
INT LcMath_CollisionPointToTriangle(const D3DXVECTOR3* pTri, const D3DXVECTOR3* pPoint)
{
        D3DXVECTOR3     vcA = pTri[1] - pTri[0];
        D3DXVECTOR3     vcB = pTri[2] - pTri[0];
        D3DXVECTOR3     vcP = *pPoint - pTri[0];

        FLOAT   AA = D3DXVec3Dot(&vcA, &vcA);
        FLOAT   AB = D3DXVec3Dot(&vcA, &vcB);
        FLOAT   BB = D3DXVec3Dot(&vcB, &vcB);
        FLOAT   AP = D3DXVec3Dot(&vcA, &vcP);
        FLOAT   BP = D3DXVec3Dot(&vcB, &vcP);

        FLOAT   fA = AP * BB - AB * BP;
        FLOAT   fB = AA * BP - AB * AP;
```

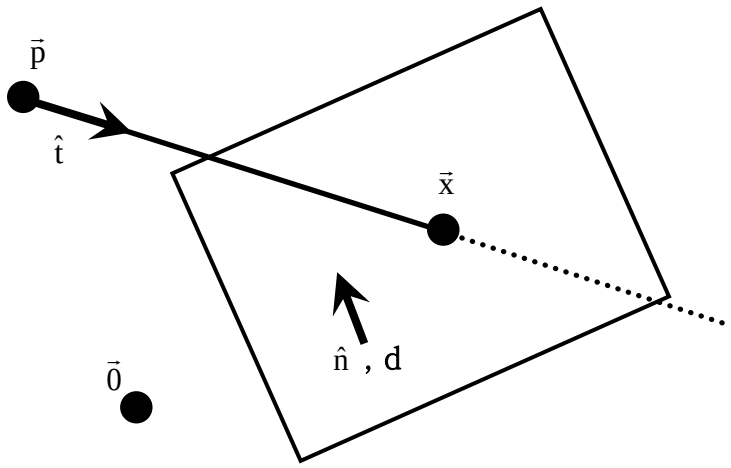
```
        FLOAT    fD = AA * BB - AB * AB;

        if( fA>=0 && fB>=0 && (fA+ fB)<=fD)
                return 0;

        return -1;
}
```

McCol_PointToTri3.zip

지금까지 삼각형과 같은 평면에 있는 점에 대한 충돌을 구현했는데 직선과 삼각형의 충돌에서 유용하게 사용이 되므로 꼭 구현해 보기 바랍니다.

2.4.3 무한 직선과 세 점으로 구성된 평면의 충돌

다음 그림과 같이 시작점이 p, 방향이 t 인 직선과 (n, d)로 구성되어 있는 평면이 만나는 점 X를 구해봅시다.



미지의 점 x 는 다음과 같이 임의의 상수 k를 사용해서 다음과 같이 나타낼 수 있습니다.

$$\vec{x} = \vec{p} + k\hat{t} \qquad (2\text{-}4)$$

이 점 x 는 또한 평면 위의 점이므로 평면의 방정식을 만족합니다.

$$\hat{n} \cdot \vec{x} = d \qquad (2\text{-}5)$$

(2-4)의 식을 (2-5)에 적용하기 위해 (2-4)의 양 변에 평면의 법선 벡터 n을 내적 시킵니다.

$$\hat{n} \cdot \vec{x} = \hat{n} \cdot (\vec{p} + k\hat{t})$$

좌변은 (2-5)에서 d가 되므로 이를 정리해서 k 값을 다음과 같이 구할 수 있습니다.

$$d = \hat{n} \cdot \vec{p} + k\,\hat{n} \cdot \hat{t}$$
$$k = \frac{d - \hat{n} \cdot \vec{p}}{\hat{n} \cdot \hat{t}}$$

K를 구했으므로 이를 다시 (2-4)에 적용해서 다음과 같은 결론을 얻을 수 있습니다.

$$\vec{x} = \vec{p} + \left(\frac{d - \hat{n} \cdot \vec{p}}{\hat{n} \cdot \hat{t}} \right) \hat{t}$$
$$\therefore \vec{x} = \vec{p} + \frac{d}{\hat{n} \cdot \hat{t}}\,\hat{t} - \frac{\hat{n} \cdot \vec{p}}{\hat{n} \cdot \hat{t}}\,\hat{t} \qquad (2\text{-}6)$$

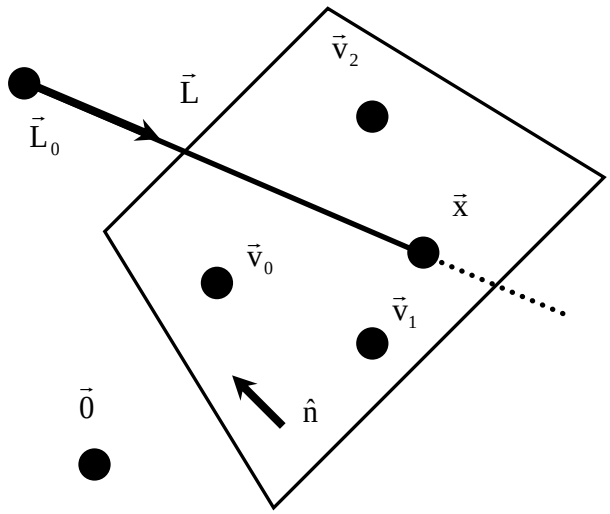
내용이 많지만 결과 (2-6) 식을 유도하기 위한 것이므로 코드는 간단합니다.

```
INT LcMath_CollisionLineToPlane(D3DXVECTOR3* pOut                 // Output Collision
                                , const D3DXPLANE* pPlane         // Input Plane
                                , const LcLine* pLine             // Input Line
{
        D3DXVECTOR3 N = D3DXVECTOR3(pPlane->a, pPlane->b, pPlane->c);
        FLOAT       D = -pPlane->d;

        D3DXVECTOR3 P = pLine->p;
        D3DXVECTOR3 T = pLine->t;

        if(pOut)
        {
                float NT = D3DXVec3Dot(&N, &T);
                float NP = D3DXVec3Dot(&N, &P);

                *pOut = P + (D-NP)/(NT) * T;
        }
```

```
        return 0;
}
```

문제 1-6) 식 (2-3)을 $\vec{x} = \vec{p}M_{d,\vec{n},\vec{t}}$ 식으로 동차좌표계에서 표현 되는 행렬 $M_{d,\vec{n},\vec{t}}$ 을 구하시오.

다음 그림과 같이 시작점이 $\vec{L}_0$, 방향이 $\vec{L}$ 인 직선과 세 점 $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$ 으로 구성된 평면의 충돌 점을 구해 봅시다.



이 문제는 평 면을 세 점으로 구성하도록 하기만 하면 식(2-6)을 이용할 수 있습니다. 먼저 평면의 정의를 하기 위해서 법선 벡터와 원점에서의 최단 거리를 다음과 같이 구할 수 있습니다.

평면의 수직 벡터 $\vec{N} = (\vec{v}_1 - \vec{v}_0) \times (\vec{v}_2 - \vec{v}_0)$,

평면의 법선 벡터 $\hat{n} = \dfrac{\vec{N}}{\|\vec{N}\|}$

평면에서의 최단거리는 다음과 같이 구할 수 있습니다.

$d = \hat{n} \cdot \vec{v}_0$

이렇게 법선 벡터와 최단거리를 구했으므로 이것을 식 (2-6)에 대입하면 다음과 같이 구해집니다.

$$\vec{x} = \vec{L}_0 + \left(\frac{\hat{n} \cdot \vec{v}_0 - \hat{n} \cdot \vec{L}_0}{\hat{n} \cdot \vec{L}_0} \right) \vec{L} \qquad (2\text{-}7)$$

수학 이론 이라면 여기서 끝내도 상관 없지만 프로그램으로 이 식을 작성한다면 법선 벡터를 만들기 위해서 수직 벡터를 정규화 하는 과정이 생기게 됩니다. 이것은 (2-7) 식을 다음과 같이 변경하면 정규화 과정을 제거 할 수 있습니다.
먼저 위의 식을 다음과 같이 정리하고 분수 식으로 되어 있는 부분에 평면의 수직 벡터의 크기를 곱합니다.

$$\vec{x} = \vec{L}_0 + \frac{\hat{n}\cdot(\vec{v}_0 - \vec{L}_0)}{\hat{n}\cdot\vec{L}}\vec{L} \quad , \quad \vec{x} = \vec{L}_0 + \frac{\|\vec{N}\|\hat{n}\cdot(\vec{v}_0 - \vec{L}_0)}{\|\vec{N}\|\hat{n}\cdot\vec{L}}\vec{L}$$

$\vec{N} = \|\vec{N}\|\hat{n}$ 이므로 다음과 같이 정리됩니다.

$$\vec{x} = \vec{L}_0 + \frac{\vec{N}\cdot(\vec{v}_0 - \vec{L}_0)}{\vec{N}\cdot\vec{L}}\vec{L} \quad , \quad \vec{x} = \vec{L}_0 + \frac{\vec{N}\cdot\vec{v}_0 - \vec{N}\cdot\vec{L}_0}{\vec{N}\cdot\vec{L}}\vec{L}$$

$D = \vec{N}\cdot\vec{v}_0$, $s = \vec{N}\cdot\vec{L}_0$, $s = \vec{N}\cdot\vec{L}_0$, $m = \vec{N}\cdot\vec{L}$ 이라 하면

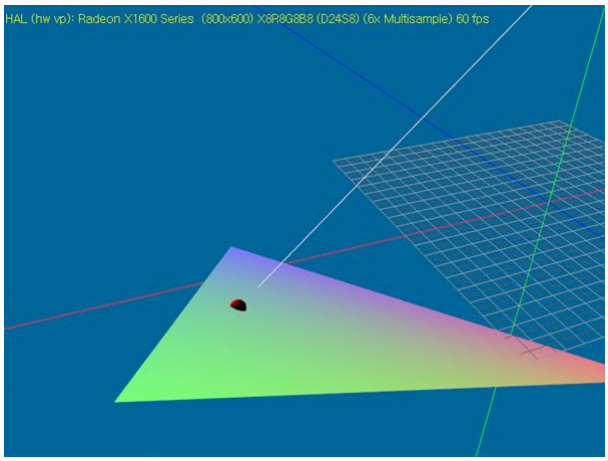
$$\therefore \vec{x} = \vec{L}_0 + \frac{D}{m}\vec{L} - \frac{s}{m}\vec{L} \qquad (2\text{-}8)$$

(단, $D = \vec{N}\cdot\vec{v}_0$, $s = \vec{N}\cdot\vec{L}_0$, $m = \vec{N}\cdot\vec{L}$)

DirectX SDK를 이용해서 구현하면 다음과 같습니다.

```
INT LcMath_CollisionLineToPlane(D3DXVECTOR3* pOut                  // Output Collision
                                , const D3DXVECTOR3* pTri// Input Triangle
                                , const LcLine* pLine              // Input Line
                                )
{
        D3DXVECTOR3 N;
        FLOAT       D;

        D3DXVECTOR3 P = pLine->p;
        D3DXVECTOR3 T = pLine->t;

        D3DXVECTOR3 vcA = pTri[1] - pTri[0];
        D3DXVECTOR3 vcB = pTri[2] - pTri[0];

        D3DXVec3Cross(&N, &vcA, &vcB);
        D = D3DXVec3Dot(&N, &pTri[0]);
```

```
        if(pOut)
        {
                float NT = D3DXVec3Dot(&N, &T);
                float NP = D3DXVec3Dot(&N, &P);

                *pOut = P + (D-NP)/(NT) * T;
        }

        return 0;
}
```



https://github.com/3dapi/ef04_math/raw/master/Col_Tri_Line1.zip

2.4.4 삼각형과 직선의 충돌 구하기

삼각형과 직선의 충돌은 식 (2-8)로 먼저 삼각형이 이루는 평면과 직선의 교차점을 구합니다. 다음으로 (2-1), (2-2), (2-3)을 구해서 조건에 맞는지 검사 하면 무한 직선과 평면의 충돌 판정을 내릴 수 있습니다.
중요한 변수와 식을 적으면 다음과 같이 됩니다.

입력 변수: 직선의 시작 위치 $\vec{L}_0$, 직선의 방향 $\vec{L}$ 충돌 검사할 삼각형 세 점 $\vec{v}_0$, $\vec{v}_1$, $\vec{v}_2$

알고리듬 구현은 다음과 같습니다.

(1) $\vec{A}=(\vec{v}_1-\vec{v}_0)$, $\vec{B}=(\vec{v}_2-\vec{v}_0)$, $\vec{N}=(\vec{v}_1-\vec{v}_0)\times(\vec{v}_2-\vec{v}_0)$을 구한다.

(2) $D=\vec{N}\cdot\vec{v}_0$, $s=\vec{N}\cdot\vec{L}_0$, $m=\vec{N}\cdot\vec{L}$ 을 구한다.

(3) 교차점 $\vec{x}=\vec{L}_0+\frac{D}{m}\vec{L}-\frac{s}{m}\vec{L}$ 을 구한다.

(4) 이 점을 삼각형 시작 위치의 상대 좌표 $\vec{x}=\vec{x}-\vec{v}_0$ 으로 만든다.

(5) $\vec{A}\cdot\vec{x}=\alpha\vec{A}\cdot\vec{A}+\beta\vec{A}\cdot\vec{B}$, $\vec{B}\cdot\vec{x}=\alpha\vec{A}\cdot\vec{B}+\beta\vec{B}\cdot\vec{B}$ 식에 대응하는

$\vec{A}\cdot\vec{x}$, $\vec{B}\cdot\vec{x}$, $\vec{A}\cdot\vec{A}$, $\vec{B}\cdot\vec{B}$, $\vec{A}\cdot\vec{B}$ 를 구한다.

(6) 행렬식 $\begin{vmatrix}\vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{B}\\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{B}\end{vmatrix}$, $\begin{vmatrix}\vec{A}\cdot\vec{x} & \vec{A}\cdot\vec{B}\\ \vec{B}\cdot\vec{x} & \vec{B}\cdot\vec{B}\end{vmatrix}$, $\begin{vmatrix}\vec{A}\cdot\vec{A} & \vec{A}\cdot\vec{x}\\ \vec{A}\cdot\vec{B} & \vec{B}\cdot\vec{x}\end{vmatrix}$ 를 계산한다.

(7) α, β 를 구하고 $0\le\alpha,\ 0\le\beta,\ \alpha+\beta\le1$ 를 검사해 본다. 이 조건에 맞으면 충돌이다.

수학적 알고리듬이 만들어지면 이것은 프로그램으로 작성이 가능하고 위의 경우는 거의 1: 1 코드로 바꾸어 작성할 수 있습니다.
충돌 프로그램 함수를 LcxIntersectTri() 함수라 한다면 위의 내용을 다음과 같이 의사 코드로 작성할 수 있습니다.

```
// 충돌: 0, 실패: -1
int LcxIntersectTri(    Vector3 V0      // 삼각형 꼭지점
                      , Vector3 V1      // 삼각형 꼭지점
                      , Vector3 V2      // 삼각형 꼭지점
                      , Vector3 L0      // 직선의 시작 위치
                      , Vector3 L       // 직선의 방향 벡터
                      , Float* u        // α 출력(선택)
                      , Float* v        // β 출력(선택)
)
{
        int hr = -1;    // 충돌을 실패로 설정

        // 임시 벡터들을 선언한다.
        Vector3 A;
        Vector3 B;
        Vector3 N;
        Vector3 X;
```

```
        // 임시 변수들을 선언한다.
        Float D, s, m;
        Float aa, bb, ab, ax, bx, m0, m1, m2;

        // 위의 알고리듬 1번부터 구한다.
        A = V1 - V0;
        B = V2 - V0;
        N = Cross(A, B);

        D = Dot(N, V0);
        s = Dot(N, L0);
        m = Dot(N, L);

        // 충돌 점을 구한다.
        X = L0 + (D-s)/m * L;
        X -=V0;
        // 행렬식에 필요한 변수들을 계산한다.
        aa = Dot(A, A);
        bb = Dot(B, B);
        ab = Dot(A, B);
        ax = Dot(A, X);
        bx = Dot(B, X);

        // 행렬식을 계산한다.
        m0 = aa * bb - ab * ab;
        m1 = ax * bb - ab * bx;
        m2 = aa * bx - ab * ax;

        if( m1>= 0 && m2 >= 0 && m0>=(m1 + m2))
                hr = 0; // 충돌

        if(u)   *u = m1/m0;
        if(v)   *v = m2/m0;

        return hr;
}
```

DirectX SDK 함수를 이용해서 다음과 같이 작성합니다.

```
INT LcMath_IntersectTri(D3DXVECTOR3* pOut                  // 충돌 위치
                        , const D3DXVECTOR3* V0            // 삼각형 꼭지점
                        , const D3DXVECTOR3* V1            // 삼각형 꼭지점
                        , const D3DXVECTOR3* V2            // 삼각형 꼭지점
                        , const D3DXVECTOR3* L0            // 직선의 시작 위치
                        , const D3DXVECTOR3* L             // 직선의 방향 벡터
                        , FLOAT* u = NULL          // α 출력(선택)
                        , FLOAT* v = NULL          // β 출력(선택)
                        )
{
        int hr = -1;     // 충돌을 실패로 설정

        // 임시 벡터들을 선언한다.
        D3DXVECTOR3      A;
        D3DXVECTOR3      B;
        D3DXVECTOR3      N;
        D3DXVECTOR3      X;

        // 임시 변수들을 선언한다.
        FLOAT D, s, m;
        FLOAT aa, bb, ab, ax, bx, m0, m1, m2;

        // 위의 알고리듬 1번부터 구한다.
        A = *V1 - *V0;
        B = *V2 - *V0;
        D3DXVec3Cross(&N, &A, &B);

        D = D3DXVec3Dot(&N, V0);
        s = D3DXVec3Dot(&N, L0);
        m = D3DXVec3Dot(&N, L);

        // 충돌 점을 구한다.
        X = *L0 + (D-s)/m * (*L);
```

```
        if(pOut)*pOut = X;

        X -= *V0;
        // 행렬식에 필요한 변수들을 계산한다.
        aa = D3DXVec3Dot(&A, &A);
        bb = D3DXVec3Dot(&B, &B);
        ab = D3DXVec3Dot(&A, &B);
        ax = D3DXVec3Dot(&A, &X);
        bx = D3DXVec3Dot(&B, &X);

        // 행렬식을 계산한다.
        m0 = aa * bb - ab * ab;
        m1 = ax * bb - ab * bx;
        m2 = aa * bx - ab * ax;

        if( m1>= 0 && m2 >= 0 && m0>=(m1 + m2))
                hr = 0;  // 충돌

        if(u)    *u = m1/m0;
        if(v)    *v = m2/m0;

        return hr;
}
```

https://github.com/3dapi/ef04_math/raw/master/Col_Tri_Line2.zip

위의 프로그램을 진행 한 후에 충돌 점을 반환 값 u, v로도 다음과 같이 계산이 됩니다.

충돌 점 = V0 + u * (V1-V0) + v * (V2 - V0)

앞의 방식이 교차점을 구한 다음 U, V를 구했습니다. 하지만 U, V를 먼저 구해서 계산량을 더 줄일 수 있는 방법이 있습니다.

먼저 다음과 같이 삼각형으로 구성한 평면에 충돌하는 점 $\vec{x}$에 대한 다음의 공식을 정리합니다.

직선의 방정식을 $\vec{x} = \vec{P} + k\,\vec{D}$ 으로 하고 $\vec{x} = \vec{v}_0 + u(\vec{v}_1 - \vec{v}_0) + v(\vec{v}_2 - \vec{v}_0)$ 에서 $\vec{A} = (\vec{v}_1 - \vec{v}_0)$,

$\vec{B} = (\vec{v}_2 - \vec{v}_0)$, $\vec{T} = \vec{P} - \vec{v}_0$로 하면 $\vec{x} = \vec{P} - \vec{T} + u\vec{A} + v\vec{B}$ 가 됩니다. 이렇게 정리하는 이유는 나중에 프로그램을 좀 더 편하게 작성하기 위해서 입니다.

이것을 $\vec{x} = \vec{P} + k\,\vec{D}$와 결합하면 $\vec{P} + k\,\vec{D} = \vec{P} - \vec{T} + u\vec{A} + v\vec{B}$ 에서 $k\,\vec{D} = -\vec{T} + u\vec{A} + v\vec{B}$으로 됩니다. 여기서 k, u, v를 외적을 이용하면 빠르게 구할 수 있습니다.

$\vec{N}_{(d,a)} = \vec{D} \times \vec{A}$, $\vec{N}_{(d,b)} = \vec{D} \times \vec{A}$, $\vec{N}_{(a,b)} = \vec{A} \times \vec{B}$ 로 하고 $k\,\vec{D} = -\vec{T} + u\vec{A} + v\vec{B}$ 식의 양변에 이 법선 벡터들을 하나씩 내적을 적용합니다. 이것은 $\vec{A}$ 와 $\vec{A} \times \vec{B}$는 수직이기 때문에 $\vec{A} \bullet \vec{A} \times \vec{B} = 0$ 의 성질을 이용해서 식을 줄이기 위함입니다.

$(k\,\vec{D}) \bullet \vec{N}_{(d,a)} = (-\vec{T} + u\vec{A} + v\vec{B}) \bullet \vec{N}_{(d,a)}$, $0 = -\vec{T} \bullet \vec{N}_{(d,a)} + u\vec{A} \bullet \vec{N}_{(d,a)} + v\vec{B} \bullet \vec{N}_{(d,a)}$

$0 = -\vec{T} \bullet \vec{N}_{(d,a)} + 0 + v\vec{B} \bullet \vec{N}_{(d,a)}$, $v\vec{B} \bullet \vec{N}_{(d,a)} = \vec{T} \bullet \vec{N}_{(d,a)}$ 에서

$$v = \frac{\vec{T} \bullet \vec{N}_{(d,a)}}{\vec{B} \bullet \vec{N}_{(d,a)}} \quad \text{또는} \quad v = \frac{\vec{T} \bullet \vec{D} \times \vec{A}}{\vec{B} \bullet \vec{D} \times \vec{A}}$$

이렇게 v를 구하고 u를 구하기 위해 또한 $\vec{N}_{(d,b)}$ 벡터로 내적을 가하면

$(k\,\vec{D}) \bullet \vec{N}_{(d,b)} = (-\vec{T} + u\vec{A} + v\vec{B}) \bullet \vec{N}_{(d,b)}$, $0 = -\vec{T} \bullet \vec{N}_{(d,b)} + u\vec{A} \bullet \vec{N}_{(d,b)} + v\vec{B} \bullet \vec{N}_{(d,b)}$

$0 = -\vec{T} \bullet \vec{N}_{(d,b)} + u\vec{A} \bullet \vec{N}_{(d,b)} + 0$

$$u = \frac{\vec{T} \bullet \vec{N}_{(d,b)}}{\vec{A} \bullet \vec{N}_{(d,b)}} \quad \text{또는} \quad u = \frac{\vec{T} \bullet \vec{D} \times \vec{B}}{\vec{A} \bullet \vec{D} \times \vec{B}}$$

으로 u를 구합니다.

마찬가지로 $\vec{N}_{(a,b)}$ 벡터로 내적을 가하면 k를 구할 수 있습니다.

$$k = -\frac{\vec{T} \bullet \vec{N}_{(a,b)}}{\vec{D} \bullet \vec{N}_{(a,b)}} \quad \text{또는} \quad k = -\frac{\vec{T} \bullet \vec{A} \times \vec{B}}{\vec{D} \bullet \vec{A} \times \vec{B}}$$

$\vec{A}$, $\vec{B}$, $\vec{C}$가 주어질 경우 내적연산에 대해서 벡터에서는 다음과 같은 성질이 있습니다.

$$\vec{A} \bullet \vec{B} = -\vec{B} \times \vec{A}, \quad \vec{A} \bullet \vec{B} \times \vec{C} = \vec{C} \bullet \vec{A} \times \vec{B} = \vec{B} \bullet \vec{C} \times \vec{A}$$

이것을 $\vec{D}$, $\vec{A}$, $\vec{B}$로 바꾸면 $\vec{A} \bullet \vec{D} \times \vec{B} = \vec{B} \bullet \vec{A} \times \vec{D} = -\vec{B} \bullet \vec{D} \times \vec{A} = -\vec{D} \bullet \vec{A} \times \vec{B}$ 이 되고 u와 k에 대해서 다음과 같이 정리됩니다.

$$u = \frac{\vec{T} \bullet \vec{N}_{(d,b)}}{\vec{A} \bullet \vec{N}_{(d,b)}}, \quad v = -\frac{\vec{T} \bullet \vec{N}_{(d,a)}}{\vec{A} \bullet \vec{N}_{(d,b)}}, \quad k = \frac{\vec{T} \bullet \vec{N}_{(a,b)}}{\vec{A} \bullet \vec{N}_{(d,b)}}$$

외적 3번 내적 4번으로 u, v, k를 아주 쉽게 구할 수 있습니다.
이것을 가지고 삼각형의 충돌과 이 때 u, v, k를 프로그램으로 작성할 때는 다음과 같은 순서로 작성하면 됩니다.

1. $\vec{A} = (\vec{v}_1 - \vec{v}_0)$, $\vec{B} = (\vec{v}_2 - \vec{v}_0)$, $\vec{T} = \vec{P} - \vec{v}_0$

2. u를 구하기 위해 먼저 $\vec{N}_{(d,b)} = \vec{D} \times \vec{B}$를 구한다.

3. 판별값(Determinant) det $= \vec{A} \bullet \vec{N}_{(d,b)}$를 구한다. det 값의 크기가 0근처이면 이후 나눗셈이 안되므로 충돌이 아니다. 또한 det가 음수이면 양수로 바꾼다. 이 때 모든 계산식이 동일하게 동작할 수 있도록 벡터 T의 부호도 바꾼다.($\vec{T} = -\vec{T}$)

4. u를 구한다. $u = \vec{T} \bullet \vec{N}_{(d,b)}$. 원래는 $u = \frac{\vec{T} \bullet \vec{N}_{(d,b)}}{\det}$에서 u가 (0,1) 범위 인지 확인해야 하나 나눗셈을 피하기 위해서 $u = \vec{T} \bullet \vec{N}_{(d,b)}$로 하고 (0, det) 범위 밖이면 충돌이 아니다.

5. $\vec{N}_{(d,a)} = \vec{D} \times \vec{A}$, $v = -\vec{T} \bullet \vec{N}_{(d,a)}$ 를 구하고 v<0 이거나 u+v>det 이면 충돌이 아니다.

6. u와 v를 dat로 나눈다. u /= det, v /= det

7. 거리 k도 필요하면 $\vec{N}_{(a,b)} = \vec{A} \times \vec{B}$, $k = \vec{T} \bullet \vec{N}_{(a,b)}$ 를 구하고 k도 det로 나눈다. k /= det.

내적, 외적이 많이 나와 조금 힘들겠지만 이전의 방식보다 계산량이 적으니 코드로 구현해 봅시다.

```
INT LcMath_IntersectTri2(D3DXVECTOR3* pOut                  // 충돌 위치
                        , const D3DXVECTOR3* V0             // 삼각형 꼭지점
                        , const D3DXVECTOR3* V1             // 삼각형 꼭지점
                        , const D3DXVECTOR3* V2             // 삼각형 꼭지점
                        , const D3DXVECTOR3* L0             // 직선의 시작 위치
                        , const D3DXVECTOR3* L              // 직선의 방향 벡터
                        , FLOAT* pU=NULL                    // Barycentric Hit Coordinates
                        , FLOAT* pV=NULL                    // Barycentric Hit Coordinates
                        , FLOAT* pDist=NULL)                // 충돌점까지 거리
{
        // 1. A, B, D, T를 정한다.
        D3DXVECTOR3 A = *V1 - *V0;
        D3DXVECTOR3 B = *V2 - *V0;
        D3DXVECTOR3 D = *L;
        D3DXVECTOR3 T = *L0 - *V0;
        D3DXVECTOR3 Nab;                    // Nab = A X B
        D3DXVECTOR3 Nda;                    // Nda = D X A
        D3DXVECTOR3 Ndb;                    // Ndb = D X B
        FLOAT    k, u, v, det;
        D3DXVECTOR3 P;

        // 2. D, B 두 벡터에 수직인 벡터를 구한다. (Ndb = D X B)
        D3DXVec3Cross(&Ndb, &D, &B);

        // 3. Determinant를 구한다. det가 음수면 양수로 바꾼다. 또한 T의 방향도 바꾼다.
        det= D3DXVec3Dot(&A, &Ndb);     // det = A·(D X B)
        if(det<0)
        {
                det = -det;
```

```
        T = -T;
}

// Determinant 0 근처면 나눗셈이 안되므로 충돌이 안된 것으로 판정하고 나간다.
if(det < 0.00001f)
        return -1;

// 4. u를 계산한다.
u = D3DXVec3Dot(&T, &Ndb);

// u의 범위는 (0, 1) 이다. 나눗셈을 하지 않았으므로 det와 비교한다.
if(u<0 || u>det)
        return -1;

// 5. Nda, v를 계산한다.
D3DXVec3Cross(&Nda, &D, &A);
v = -D3DXVec3Dot(&T, &Nda);

// u+v의 범위는 (0, 1) 이다. v도 나눗셈을 하지 않았으므로 det와 비교한다.
if(v<0 || (u+ v)>det)
        return -1;

// 6. 충돌 결과를 계산하기 위해 det를 역수 취하고 나머지 숫자에 곱한다.
det = 1/det;
u*= det;
v*= det;

if(pU)  *pU = u;
if(pV)  *pV = v;

// 7. 거리를 계산한다.
if(pDist)
{
        D3DXVec3Cross(&Nab, &A, &B);
        k = D3DXVec3Dot(&T, &Nab);
        k *=det;
        *pDist = k;
```

```
        }

        // 8. 충돌 지점을 구한다.
        if(pOut)
        {
                P = u * A +  v * B;      P +=*V0;
                *pOut = P;
        }

        return 0;
}
```

전체 코드와 테스트는 https://github.com/3dapi/ef04_math/raw/master/Col_Tri_Line2_1.zip 를 참고 하기 바랍니다.

만약 DirectX SDK에서 제공하는 함수 D3DXIntersectTri()을 사용하려고 한다면 다음과 같이 코드를 작성해야 합니다. D3DXIntersectTri() 함수는 충돌이 발생할 때 TRUE 값을 반환하고 이 때 u, v 값을 이용해서 충돌 지점을 (2-9) 식을 이용해서 다음과 같이 구해야 합니다.

```
D3DXVECTOR3 V0, V1, V2;
V0 = m_pTri[0].p;
V1 = m_pTri[1].p;
V2 = m_pTri[2].p;

INT hr;
D3DXVECTOR3 LineBegin      = m_pLine[0].p;
D3DXVECTOR3 LineDirection = m_pLine[1].p - m_pLine[0].p;
FLOAT              u, v, d;

hr = D3DXIntersectTri(&V0, &V1, &V2, &LineBegin, &LineDirection, &u, &v, &d);

// 충돌 위치 구하기
if(TRUE == hr)
        m_vcPick = V0 +  u * (V1 - V0) +  v * (V2 - V0);
```

전체 코드는 다음 예제를 이용하기 바랍니다.

https://github.com/3dapi/ef04_math/raw/master/Col_Tri_Line3.zip

문제 1-7) 위의 의사 코드를 DirectX SDK를 사용해서 함수로 만들어 보시오. 입력 변수 값은 포인터를 이용하시오.

그리고 임의로 입력 값을 주었을 때 D3DXIntersectTri ()함수와 같은 결과가 나오는지 확인하시오.