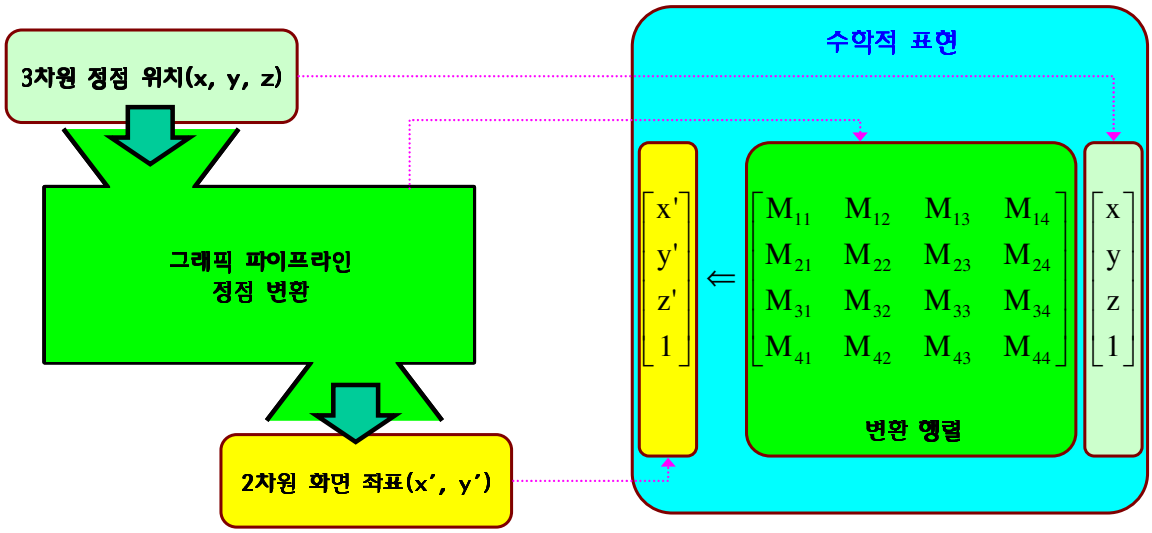
3D Game Programming Basic with Direct3D

2. 3D 기초 수학

그래픽 파이프라인의 모든 과정은 수학적인 모델을 사용합니다. 특히, 정점의 변환 부분은 이와 관련한 수학이 반드시 필요합니다. 만약 여러분이 정점 변환이나 픽셀 처리과정을 프로그램 해야 한다면 수학적인 해결책이 없으면 쉐이더 프로그램을 작성하기가 어렵습니다.
수학이 나오니까 어려워 보이지만 그래픽 파이프라인에 대한 수학적인 지식은 고등학교 정도 수준이면 충분합니다. 여기에 고등학교 교과서에 나오지 않는 몇 가지 개념을 알면 파이프라인의 수학적인 처리과정은 끝냈다고 할 수 있습니다.

3D 렌더링의 기본은 프리미티브(Primitive)이고, 프리미티브는 정점으로 구성된다고 이전 장에서 이야기 했습니다. 정점(Vertex)은 위치(벡터), 색상, 법선(벡터), 텍스처좌표 등으로 구성되어 있습니다. -간혹 Vertex를 순 우리말인 꼭지점으로 번역하기도 하는데 3D의 정점은 위치 그 이상을 포함한 객체이기에 계속 정점이란 단어를 사용하겠습니다. - 이 정점에 반드시 포함시켜야 하는 것이 위치 입니다. 이 위치 값을 이용해서 그래픽 파이프라인의 정점 변환을 진행합니다.
그런데 위치는 3차원에서 표현할 때 보통 x, y, z를 이용합니다. 수학에서 그 표현 대상이 하나이상의 성분으로 구성될 때 이것을 벡터라고 합니다. 즉, 위치는 벡터의 한 종류 이며 위치를 다룬다는 것은 곧 벡터를 이용한다라는 것과 일맥 상통합니다. 따라서 위치를 다루는 3D에서 벡터에 대한 지식은 필 수가 됩니다.
벡터는 행렬을 통해서 변환이 되므로 행렬 또한 잘 알고 있어야 합니다.



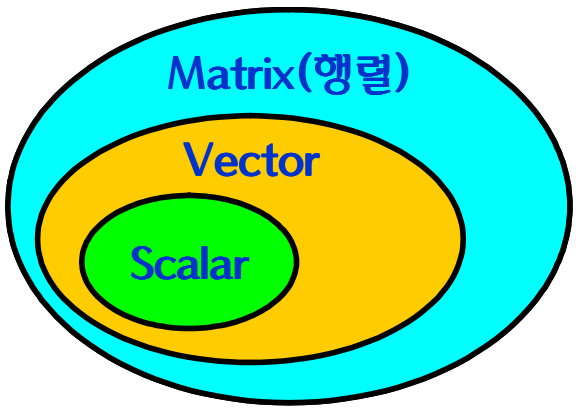
<그래픽 파이프라인의 정점 변환과 수학적 표현>

그리고 파이프라인의 정점 변환은 정점의 위치 값을 가지고 행렬 연산을 합니다. 즉, 3D의 변환이라는 것은 위의 그림처럼 위치를 행렬을 통해서 값을 변화시키는 것이라 할 수 있습니다.
간단히 3D 변환에서 벡터와 행렬의 역할을 살펴 보았고 다음으로 좀 더 수하적인 표현 방법을 공부해 봅시다.

2.1 벡터(Vector)

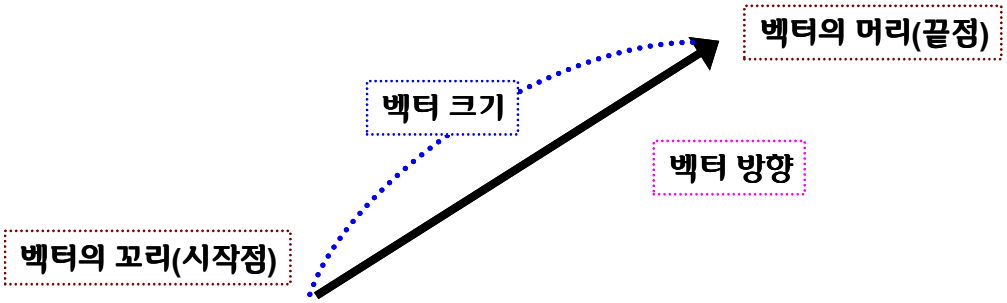
2.1.1 벡터와 벡터의 표현

벡터를 제대로 공부하려면 선형 대수학(Linear Algebra)을 배워야 합니다. 선형 대수학을 거칠게 표현한다면 선형(Linear)적 성질에 대한 숫자를 대신해서 기호로 표현하는 학문이라 할 수 있겠습니다. 이 선형대수학에 벡터공간, 행렬, 스칼라에 대한 자세한 설명 있지만 여기서는 3D에서 사용하는 내용을 경험을 바탕으로 설명하겠습니다.
벡터를 설명할 때 스칼라로 같이 설명하는데 스칼라는 어떤 대상을 수학적으로 표현하기 위해 1개의 요소로 표현할 수 있으면 이것을 스칼라라 합니다. 벡터는 대상을 수학적으로 표현하기 위해 최
소한 1개 이상의 쌍으로 표현을 해야 하는 대상은 벡터가 됩니다.
예를 들어 사람의 키, 몸무게, 온도, 속력, 전력, 저금한 돈, 이런 것들이 스칼라가 됩니다. 벡터의 대표적인 예는 바로 위치 입니다. 또한 위치와 관련 있는 속도, 가속도, 힘 등과 전기장, 자기장 등은 벡터가 됩니다.
만약 성분이 3개가 필요하다면 이 벡터는 3차원 벡터가 됩니다. N개가 필요하면 N차원 벡터가 됩니다. 따라서 스칼라는 1차원 벡터라 할 수 있습니다.



옆의 그림처럼 스칼라는 벡터의 부분집합으로 표현할 수 있습니다. 또한 벡터는 행렬의 부분 집합으로 볼 수 있습니다.
고등 미적분학(Advanced Calculus)에서 이러한 관계를 일반적인 개념으로 확대시킨 텐서(Tensor)를 이용해서 스칼라는 0-Rank 텐서, 벡터는 1-Rank 텐서, 행렬은 2-Rank 텐서로 표현합니다. 3D 게임 프로그램에서는 텐서까지 알 필요가 없기 때문에 이 부분은 그냥 넘어가겠습니다.

벡터를 표현하는 방법은 보통 기하학적, 대수학적, 벡터 대수학적 표현 3가지 방법이 있습니다. 기하학적인 표현은 여러분이 벡터를 교과서에서 처음 보았던 그 모양 그대로 다음 그림과 같이 표현하는 것입니다.



<벡터의 기하학적 표현>

벡터의 크기는 화살표 길이에 방향은 화살표 방향에 대응하는 가장 직관적인 표현법입니다. 사실 이 것은 고등학교에서가 아니라 이미 초등학교 때 “힘의 합력”을 평행사변형을 그려서 구하는 시간에 나왔던 그림입니다.
가장 직관적이지만 약점이 있는데 이렇게 표현한 벡터가 어떤 차원이 있는지 알 수 없고, 크기는 상대적이라 정확하지 않다는 것입니다.

두 번째 표현 방법은 대수학적 방법입니다. 이 방법은 차원에 따라 n개의 원소로 벡터를 표현합니다. 예를 들어 1차원 벡터는 (a) 또는 [a]로 표현하고, 2차원 벡터는 (a, b) 또는 [a, b], 3차원 벡터는 (a, b, c) 또는 [a, b, c] n차원 벡터는 (a, b, …, n) 또는 [a, b,…, n] 등으로 표현합니다. 이 때 벡터를 구성하는 a, b, c 를 벡터의 성분(Element, 원소)이라 합니다.
이 방법은 정확한 계산이 가능한 반면에 직관적이지 않아 머리에서 상상하기가 어렵다는 단점이 있습니다.

벡터를 표현하는 세 번째 방법은 벡터 기호를 이용한 벡터 대수학 입니다. 벡터 기호 표기는 상당히 많습니다. 그 중에서 대표적인 것이 굵은 고딕 문자로 표현하는 것입니다. 이 방법은 인쇄 기술이 발전하기 전에 만든 방법이며 주로 이전 출판물에서 볼 수 있습니다.

EX) **M, V, T**

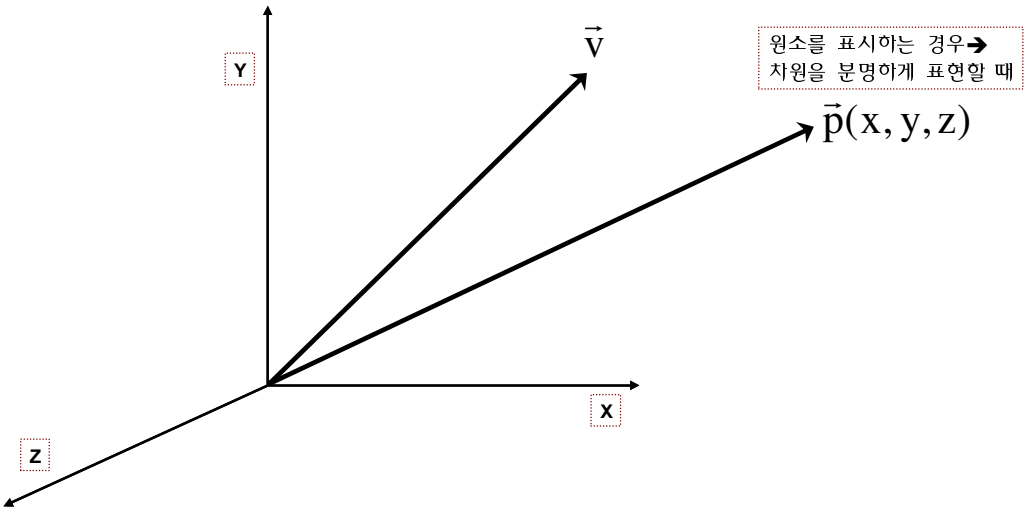
인쇄 기술이 발전하면서 문자 위에 화살표를 넣어서 벡터임을 나타내고, 아직까지 가장 광범위하게 사용하는 방법입니다.

EX) $\vec{M}$, $\bar{V}$, $\underset{\sim}{T}$

일부 물리학자는 강의를 편리하게 하기 위해서 완전한 화살표 대신 반쪽 화살표를 사용하거나 아니면 문자 밑에 ~을 넣기도 합니다.

이 외에 위 첨자 또는 아래 첨자에 차원을 넣어서 V_i, M^i_j 등으로 표기하는 방법도 있습니다.

현재 인쇄물에서 벡터에 대한 여러 표현이 각각 장 단점이 있어 보통 위의 세 가지 방법을 혼합해서 표현하는 경우가 많습니다. 다음 그림은 위치를 위의 3가지 방법을 혼합해서 표현한 것입니다.



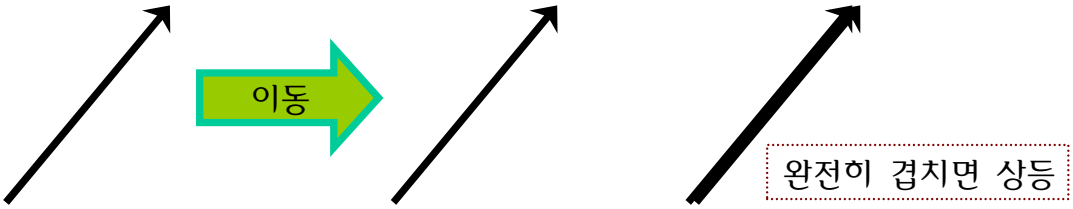
<벡터를 표현하는 방법> -기하학, 대수학, 기호를 가지고 동시에 표현하는 경우가 가장 많다.

EX) 여러 방법을 혼합해서 하는 예 $\vec{P}(x,y,z)$, $\mathbf{M}(x,y,z)$

2.1.2 벡터의 상등

벡터의 상등은 두 벡터가 같음을 의미 합니다.

기하학적 표현: 화살표의 길이와 방향이 같으면 상등인데 상등인지 판별하는 방법은 하나의 벡터를 이동해서 다른 벡터와 완전히 겹치게 되면 상등이 됩니다.



<두 벡터의 상등 테스트>

대수학적 표현에서는 차원이 같고 각각의 차원에 대응되는 원소의 값이 같은 경우에만 상등이 됩니다.

EX) vector a=(a_1, a_2, a_3, a_4), vector b=(b_1, b_2, b_3, b_4)

$a_1 == b_1$, $a_2 == b_2$, $a_3 == b_3$, $a_4 == b_4$ 인 경우에만 상등

기호로 상등을 표현할 때는 그냥 등호('=')를 사용합니다. EX) $\vec{v}_1 = \vec{v}_2$

프로그램에서 벡터의 상등을 판별하기 위해서 다음과 같이 비교하는 코드를 만들 수 있습니다.

```
if(v1.x == v2.x && v1.y == v2.y && v1.x == v2.x)
{
        상등;
}
```

하지만 게임 프로그램에서 벡터는 float형으로 표현되는 경우가 많이 있으므로 float형 오차를 고려해야 할 때도 있습니다. 이럴 때 작은 편차 값 Epsilon을 이용해서 비교하는 코드를 작성합니다.

```
const FLOAT EPSILON = 0.001f;

if(fabsf(v1.x - v2.x)< EPSILON &&
  fabsf(v1.y - v2.y)< EPSILON &&
  fabsf(v1.z - v2.z)< EPSILON)
{
        상등;
}
```

2.1.3 벡터의 크기(Length)

벡터의 크기(Length, 길이)는 기하학적 표현에서는 화살표에 대한 길이 또는 선분의 길이가 됩니다. 대수적인 표현에서 크기는 만약 n차원의 벡터 $\mathbf{v}(v_1, v_2, \cdots, v_n)$ 가 주어지면 이 벡터 $\mathbf{v}$ 의 길이는 다음과 같이 결정합니다.

벡터 $\mathbf{v}$의 크기 = $\sqrt{v_1^2 + v_2^2 + v_3^2 + \ldots + v_n^2} = \left(\sum v_i^2\right)^{\frac{1}{2}}$

벡터의 크기에 대한 기호는 $\|\vec{v}\|$, $|\vec{v}|$ 로 나타내는데 이것을 "Norm" 또는 "Length"로 읽습니다. 현재는 Norm 대신 Length를 많이 사용합니다.

DirectX SDK(이하 DXSDK)는 벡터의 길이를 구해주는 D3DXVec{2|3|4}Length(),
D3DXVec{2|3|4}LengthSq() 함수 2 가지를 제공하고 있습니다.

D3DXVec{2|3|4}Length() 함수는 벡터의 크기를 구해주는 함수이고 D3DXVec{2|3|4}LengthSq() 함수는 벡터 크기의 제곱을 반환합니다. D3DXVec{2|3|4}Length() 함수는 내부에서 제곱근을 구하는 함수 sqrt()를 사용합니다.
이 함수는 단순 곱셈 함수들보다 많은 연산을 하는데 단순하게 두 벡터의 크기 비교만 할 때는 sqrt() 를 사용하지 않는 것이 좋고 이런 경우에 D3DXVec{2|3|4}LengthSq() 함수를 사용합니다.

EX) D3DXVec3Length() 함수 사용 예

```
D3DXVECTOR3 v1(123.F, 456.F, 789.F);
Length = D3DXVec3Length(&v1); // sqrtf( v1.x * v1.x + v1.y * v1.y + v1.z * v1.z)와 동일
printf("%12.6fWn", Length);

FLOAT v2[] = {123.F, 456.F, 789.F};
Length = D3DXVec3Length( (D3DXVECTOR3*)v2);
printf("%12.6fWn", Length);
```

2.1.3 단위 벡터(Unit Vector)와 벡터의 정규화(Normalize, Normalization)

벡터의 정규화 또는 규격화는 벡터의 크기를 1로 만드는 것입니다. 이 정규화의 목적은 벡터의 방향성만 나타내고 싶을 때 사용하는 방법입니다. 이 정규화된 벡터를 단위 벡터(Unit Vector)라 합니다. 또한 차원에 관계 없이 크기가 1인 벡터는 모두 단위 벡터라 할 수 있습니다.

단위 벡터 표현은 벡터 기호를 사용하면 화살표 대신 "^" 을 사용하는 데 Hat(모자)로 읽습니다.
단위 벡터: $\hat{v}$

만약 임의의 벡터를 정규화할 필요가 있을 때 정규화와 단위벡터의 관계를 수식으로 표현하면 다음과 같습니다.

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}, \quad \text{3차원의 경우 } \hat{v} = \left(\frac{v_x}{\|\vec{v}\|}, \frac{v_y}{\|\vec{v}\|}, \frac{v_z}{\|\vec{v}\|}\right)$$

즉, 주어진 벡터를 자신의 크기로 나누면 1이 되므로 위와 같이 표현할 수 있습니다. 정규화 또한 SDK에서 D3DXVec{2|3|4}Normalize(*pOut, *pIn) 함수로 제공하고 있습니다

EX) D3DXVec3Normalize() 함수 사용 예

```
float Length = 0;
D3DXVECTOR3 v1(123.F, 456.F, 789.F);
```

```
D3DXVECTOR3 v3, v4;
D3DXVec3Normalize(&v3, &v1);
printf("%f %f %fWn", v3.x, v3.y, v3.z);
Length = sqrtf( v1.x * v1.x + v1.y * v1.y + v1.z * v1.z);
v4 = v1; v4 /= Length;
printf("%f %f %fWn", v4.x, v4.y, v4.z);
```

n 차원의 단위 벡터 중에서 k 번째 성분만 1이고, 나머지는 전부 0인 벡터를 표준 기저(Base Vector)라 합니다. 예를 들면 3차원 데카르트 좌표계의 경우에 (1,0,0), (0,1,0), (0,0,1) 벡터는 표준 기저 벡터가 됩니다.
이 기저 벡터를 가지고 임의의 벡터를 표현 할 수 있습니다.

3차원 공간에서 (1,0,0), (0,1,0), (0,0,1) 기저 벡터를 $\hat{i}, \hat{j}, \hat{k}$ 라 하고 (2,3,4) 가 주어지면 이 (2, 3, 4) 벡터를 기저 벡터로 표현하면 다음과 같습니다.

(2,3,4) = 2*(1,0,0) + 3*(0,1,0) 4*(0,0,1)

$= 2*\hat{i}+3*\hat{j}+4*\hat{k}$

일반적으로 n 차원의 벡터는 기저 벡터를 이용해서 $\vec{v}(v_1, v_2, \cdots, v_n) \Rightarrow \sum v_i \hat{e}_i$ 으로 표현하며 때로는 시그마(Σ)를 생략하고 $\vec{v}(v_1, v_2, \cdots, v_n) = v_i \hat{e}_i$ 로 표기하기도 합니다.

2.2 벡터 연산

벡터도 스칼라와 마찬가지로 연산이 가능합니다. 벡터는 더하기(합성), Negative, 빼기, 스칼라 배, 내적, 외적 다섯 가지 연산이 있습니다. 이중에서 벡터의 더하기 빼기, 내적, 외적은 벡터 사이의 연산으로 주어진 두 벡터는 반드시 차원이 같아야만 연산이 가능합니다.

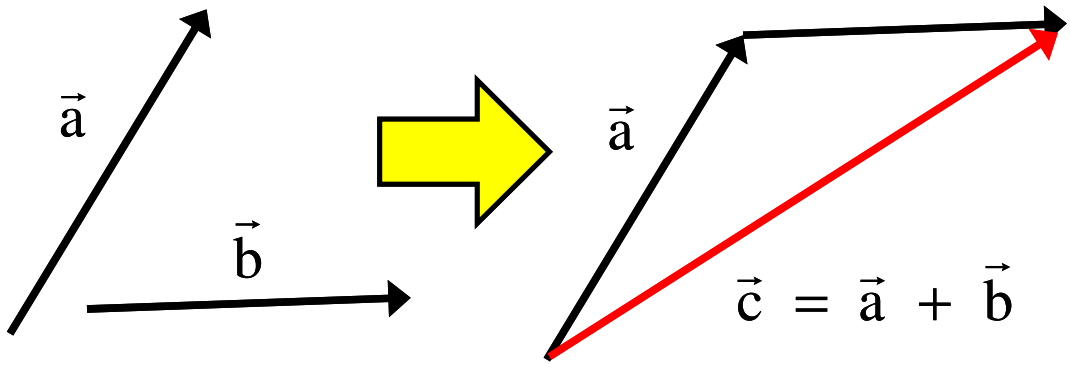
2.2.1 벡터의 덧셈(+: 합성)

벡터의 더하기는 두 벡터의 각각의 성분끼리 더해 새로운 벡터를 만드는 것입니다.
만약 벡터 **a**(a_1, a_2, a_3, …), 벡터 **b**(b_1, b_2, b_3, …) 가 있을 때 이 두 벡터 더해서 새로운 벡터 **c**는 다음과 같이 구합니다.

c = **a** + **b**
= (a_1 + b_1, a_2 + b_2, a_3 + b_3, …)

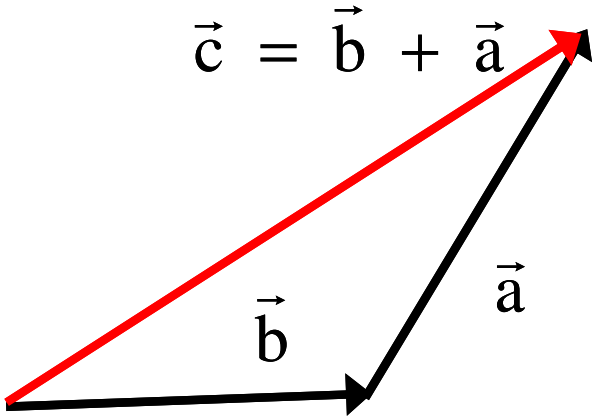
혹은 $\mathbf{c}_i = \mathbf{a}_i + \mathbf{b}_i$ 또는 $\vec{c} = \vec{a} + \vec{b}$

이것을 기하학적으로 삼각형법과 평행사변형법 두 가지 작도법이 있습니다.
삼각형법은 **c** = **a** + **b**
에서 다음 그림과 같이 "+" 다음에 오는 **b** 벡터를 벡터의 화살표 머리에 화살표 꼬리를 옮긴 후 **a**의 시작점과 옮기 **b**의 화살표 머리를 새로운 벡터 **c** 로 작성하는 것입니다.



<벡터의 덧셈1 - 삼각형법>

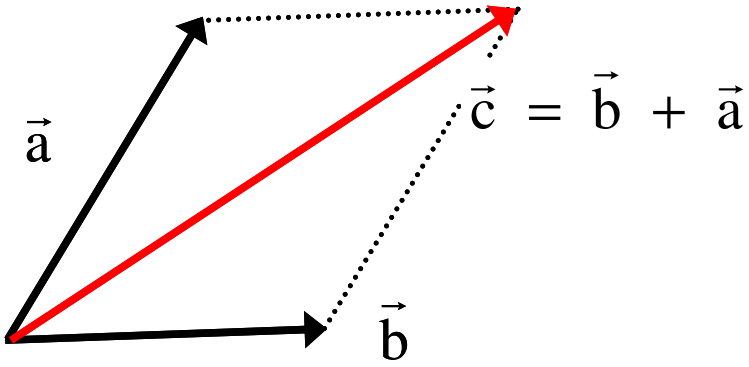
삼각형 법은 반대로 **c** = **b** + **a** 도 구할 수 있습니다.



<벡터의 덧셈2 - 삼각형법>

위의 두 그림에서 보듯이 순서가 바뀌어도 두 벡터의 덧셈 결과는 동일합니다.

평행사변형 법은 여러분이 초등학교 때 배운 힘의 합 구하기에서 했던 방식 그대로 입니다. 이 방법은 **c** = **a** + **b**에서 **a**와 **b**로 평행사변형을 만들어 대각선 방향으로 다음 그림과 같이 최종 벡터를 만드는 것입니다.



<벡터 합성 - 평행사변형법>

합성에 대한 두 가지 방법의 결과는 같지만 논리적으로 벡터의 합성에서는 삼각형법이 좋고, 평행사변형 법은 분해에서 이용하는 것이 좋다고 봅니다.

2.2.2 Negative(부호 반전)

Negative는 벡터 앞에 "-" 부호를 붙이는 것입니다.

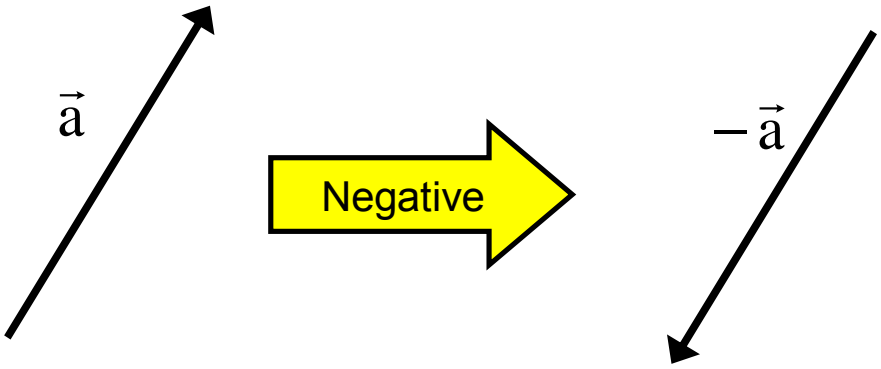
Negative **a** = - **a**

이 것은 벡터 성분의 부호를 반전시키는 것과 동일합니다.

a(a_1, a_2, a_3, …) ➔ - **a** = -(a_1, a_2, a_3, …) ➔ - **a** = (-a_1, -a_2, -a_3, …)

기저 벡터를 이용하면 Negative **a** = - **a** = $-a_i\hat{e}_i$ 으로 표현합니다.

기하학적 표현은 벡터의 크기는 그대로 두고 화살표 방향을 반대로 합니다.



<Negative>

2.2.3 벡터의 뺄셈(-)

두 벡터 사이의 뺄셈은 하나의 벡터를 부호 반전한 다음 더합니다.

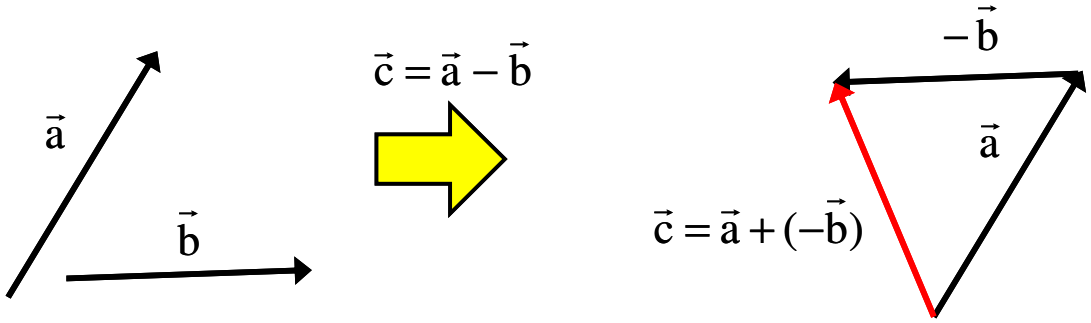
$$\vec{c} = \vec{a} - \vec{b}$$
$$= \vec{a} + (-\vec{b})$$

이것은 다음과 같이 성분끼리 뺄셈한 것과 동일 합니다.

$$\mathbf{c} = \mathbf{a} - \mathbf{b}$$
$$= \mathbf{a} + (-\mathbf{b})$$
$$= (a_1,\ a_2,\ a_3,\ \cdots) + (-b_1,\ -b_2,\ -b_3,\ \cdots)$$
$$= (a_1 - b_1,\ a_2 - b_2,\ a_3 - b_3,\ \cdots)$$

또는 $\mathbf{c}_i = \mathbf{a}_i - \mathbf{b}_i$

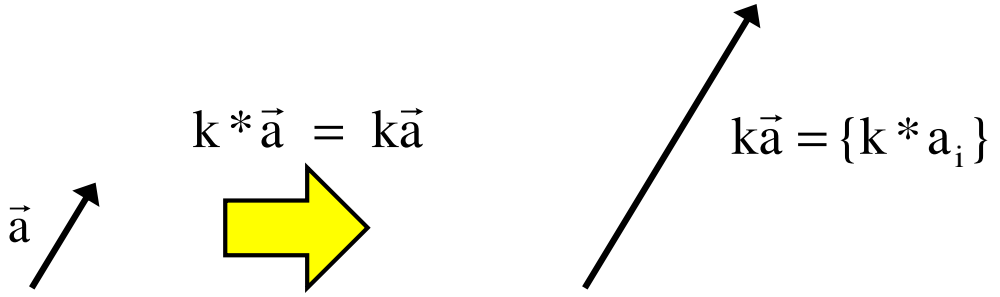
벡터에 대한 뺄셈을 기하학으로 표현하면 "-" 연산자 다음에 오는 벡터의 방향을 반대로 한 다음 덧셈을 진행하면 됩니다.



<두 벡터의 뺄셈>

2.2.4 벡터의 스칼라 배(실수 배)

벡터의 스칼라 배는 벡터에 스칼라를 곱하는 것입니다. 이것은 만약 주어진 스칼라 값이 0보다 크거나 같으면 벡터의 방향은 그대로 두고 크기만 변화 시키는 것과 동일하고 0보다 작으면 벡터의 길이가 음수(-)가 되는데 길이는 0보다 크거나 같으므로 벡터의 크기를 늘린 상태에서 방향을 반대로 합니다. 기하학적인 그림으로 표현하면 다음과 같습니다.



<벡터의 스칼라 배>

대수적으로 방법은 각각의 벡터 성분에 주어진 값을 곱하는 것과 동일합니다.

a(a1, a2, a3, …), k= 실수

k* **a** = k*(a_1, a_2, a_3, …) = (k * a_1, k * a_2, k * a_3, …) = (k * $\mathbf{a}_i$) $\mathbf{e}_i$ = k $\mathbf{a}_i$ $\mathbf{e}_i$

벡터의 스칼라 배로 이전의 Negative를 설명할 수 있습니다. Negative는 하나의 벡터에 -1을 곱한 것과 같습니다.

또한 스칼라 배를 통해서 0-벡터(영 벡터: . $\vec{0}$, **0**)를 정의할 수 있습니다. 벡터에 0을 곱하면 모든 성분이 0이 되므로 크기가 0이 됩니다. 이 영-벡터는 방향이 없어 무 방향 벡터라 부르기도 합니다.

돌발 퀴즈. 여기서 잠깐 단순하게 크기와 방향을 가지는 것이 벡터라 한다면 0-벡터는 벡터라 할 수 있을 까요?

2.3 벡터의 내적

내적의 한자 적(積)은 영어의 product를 번역한 것입니다. 내적은 각각의 성분 사이 곱셈 성질을 반영하는 경우 Inner Product라 합니다. 또한 내적 연산자의 기호가 점(dot, " · ")으로 표기를 해서 Dot Product라 하며 내적의 결과가 스칼라여서 Scalar Product로 부르기도 합니다.

가장 많이 사용되는 용어는 Dot Product 혹은 Scalar Product 입니다.

내적의 정의는 두 벡터의 크기를 곱하고 여기에 다시 두 벡터가 이루는 각도에 대한 cosine 값을 곱한 것이 내적입니다.

벡터 기호를 사용하면 다음과 같습니다.

두 벡터 $\vec{a}, \vec{b}$에 대한 내적: $\vec{a} \bullet \vec{b} = |\vec{a}||\vec{b}|\cos(\vec{a}, \vec{b})$

$\vec{a} \bullet \vec{b}$를 읽는 방법은 vector a dot vector b 혹은 a dot b(에이 도트 비)라 읽습니다.

이 정의를 cosine 제 2 법칙을 적용해서 풀면 다음과 같이 벡터의 성분끼리 각각 곱하고 이 값들을 더한 결과가 됩니다. (이 부분은 각자 증명해 보기 바랍니다.)

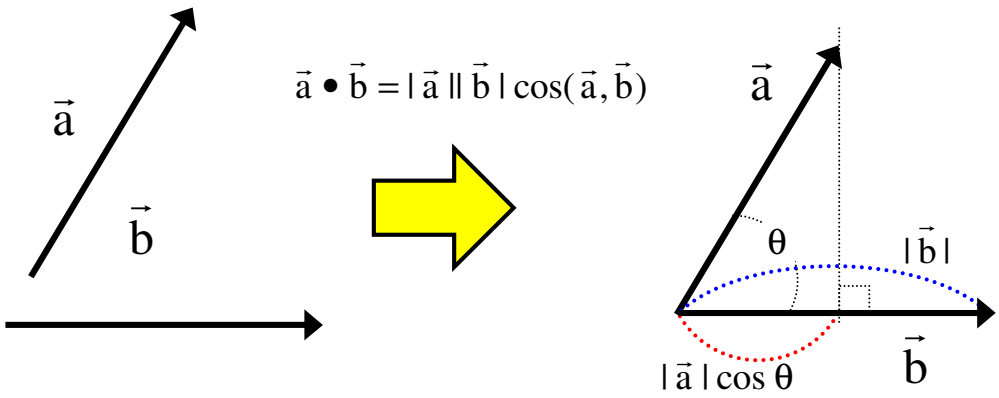
$\vec{a} \bullet \vec{b} = \sum a_i * b_i$, 혹은 $\vec{a} \bullet \vec{b} = a_i * b_i$

게임 프로그램에서는 내적을 구할 때 $\vec{a} \bullet \vec{b} = \sum a_i * b_i$ 식을 사용합니다. 내적에 대한 정의 대로 풀려면 두 벡터 사이의 각도를 구하고 다시 벡터의 크기 등을 계산해야 하는데 이 식은 단순한 곱셈 연산만으로 정확한 값을 구할 수 있습니다.

DXSDK의 D3DXVec{2|3|4}Dot() 함수는 내적을 구하는 함수 입니다. 이 함수의 사용법은 다음과 같습니다.

```
float    fDot;
D3DXVECTOR3 v5(1,2,3);
D3DXVECTOR3 v6(7,8,9);
fDot = D3DXVec3Dot(&v5, &v6);
```

내적을 그림으로 표현하면 다음과 같습니다.



<벡터의 내적>

위의 그림에서 만약 벡터 $\vec{b}$가 크기가 1인 단위 벡터라면

$$\vec{a} \bullet \vec{b} = |\vec{a}||\vec{b}|\cos(\vec{a},\vec{b})$$
$$= |\vec{a}|\cos(\vec{a},\vec{b})$$
$$= |\vec{a}|\cos\theta$$

이 됩니다. 이 값은 $\vec{a}$ 가 $\vec{b}$ 에 사상(Projection)하는 것과 같아서 "내적의 의미는 Projection 이다" 라고 생각해도 됩니다.

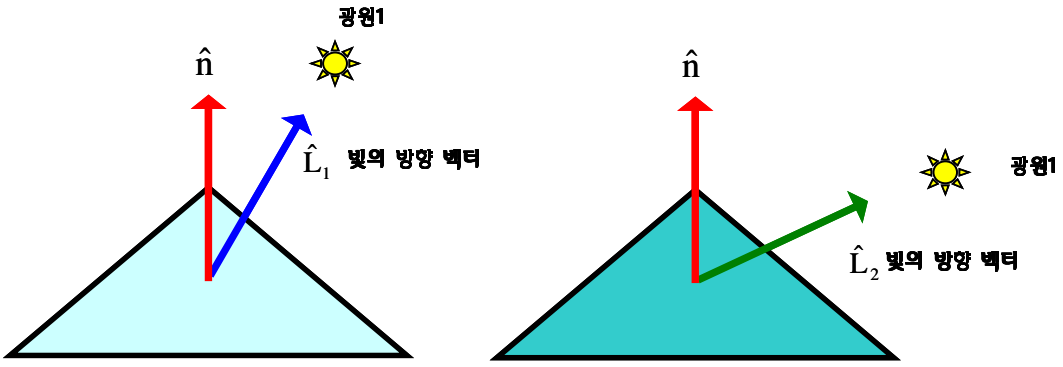
만약 두 벡터가 같은 벡터라면 내적은 벡터의 크기에 대한 제곱이 됩니다. 즉 내적을 이용해서 벡터의 길이를 구할 수 있습니다.

$$\vec{a} \bullet \vec{a} = |\vec{a}||\vec{a}|\cos(\vec{a},\vec{a})$$
$$= |\vec{a}|^2 \cos(0)$$
$$= |\vec{a}|^2 * 1$$
$$= |\vec{a}|^2$$
$$\therefore |\vec{a}| = \sqrt{(\vec{a} \bullet \vec{a})} = (\vec{a} \bullet \vec{a})^{\frac{1}{2}}$$

그래픽 파이프라인에서는 내적을 은면 제거와 조명 반사에 대한 밝기(Intensity)를 정할 때 사용합니다. 조명의 밝기 부분은 쉐이더 부분에서 자세히 다루게 되므로 일단 여기서는 간단하게만 소개하는 정도로 하겠습니다.

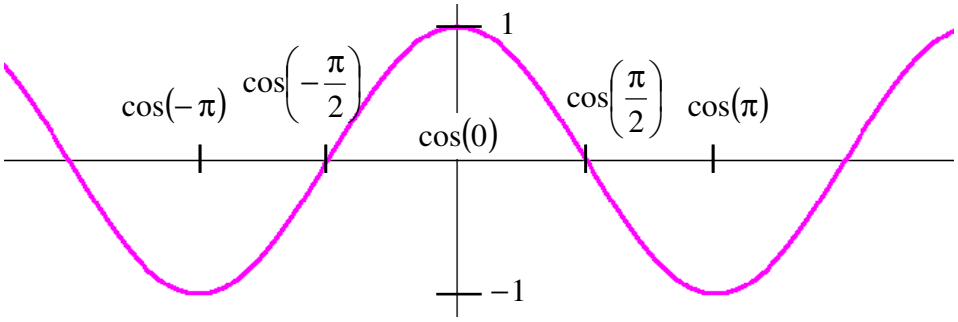
2.4 내적을 이용한 반사 밝기 설정 (Back Face Culling)

은면 제거는 다음 그림과 같이 렌더링에서 가상 카메라의 시선 z 축 벡터와 오브젝트의 삼각형의 법선 벡터의 내적 결과를 이용하는 것입니다.



<그림 반사 세기(Intensity)>

위의 그림에 만약 광원1과 광원2가 같은 밝기의 조명이라면 삼각형에서 반사되는 빛은 광원1일 때가 더 밝습니다. 즉, 빛의 방향 벡터와 삼각형의 법선 벡터가 평행일 때가 가장 밝고 그 이외는 점차 어두워져야 합니다. 그런데 cosine 그래프를 다음 그림에서 보면 각도의 부호에 상관 없이 0도에서는 최대값이 되고-π에서는 최소가 됩니다.



<cosine 그래프>

3D의 조명은 빛의 방향 벡터를 [-π/2, π/2] 범위로 제한해서 이 범위의 값을 cosine에 적용해서 사용합니다.

즉, 위의 광원 L1, L2에 대해서 반사에 대한 빛의 세기를 내적을 통해서 얻습니다.

$$\begin{aligned}\hat{n} \bullet \hat{L}_1 &= |\hat{n}||\hat{L}_1|\cos(\hat{n},\hat{L}_1) \\ &= 1*1\cos(\hat{n},\hat{L}_1) \\ &= \cos(\hat{n},\hat{L}_1)\end{aligned}$$

이므로 **L1**에 대한 반사 세기 = $\mathbf{n_x} * \mathbf{L1_x} + \mathbf{n_y} * \mathbf{L1_y} + \mathbf{n_z} * \mathbf{L1_z}$

$$\begin{aligned}\hat{n} \bullet \hat{L}_2 &= |\hat{n}||\hat{L}_2|\cos(\hat{n},\hat{L}_2) \\ &= 1*1\cos(\hat{n},\hat{L}_2) \\ &= \cos(\hat{n},\hat{L}_2)\end{aligned}$$

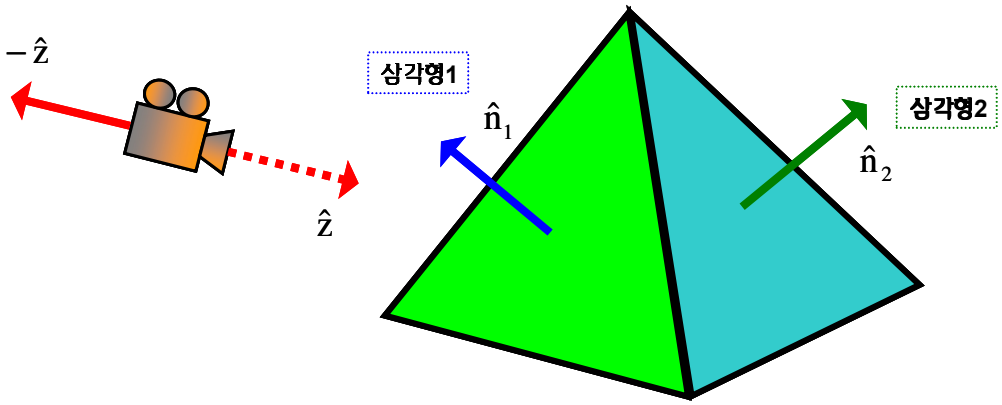
이므로 **L2**에 대한 반사 세기 = $\mathbf{n_x} * \mathbf{L2_x} + \mathbf{n_y} * \mathbf{L2_y} + \mathbf{n_z} * \mathbf{L2_z}$

이후 쉐이더에서 조명을 직접 프로그램할 때 위와 같은 방법으로 내적을 이용해서 밝기를 설정하니 잘 기억하기 바랍니다.

2.5 내적을 이용한 은면 제거(Back Face Culling)

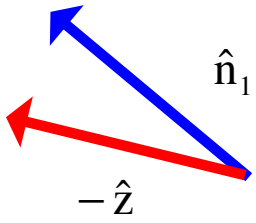
은면 제거는 다음 그림과 같이 렌더링에서 가상 카메라의 시선 z 축 벡터와 오브젝트의 삼각형의

법선 벡터의 내적 결과를 이용하는 것입니다.

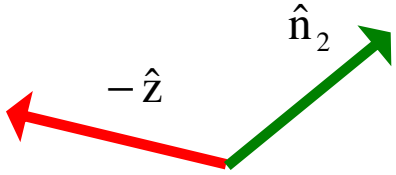


<가상 카메라와 오브젝트의 내적 비교>

그림에서 보듯 렌더링이 필요 없는 삼각형은 "삼각형2"가 됩니다. 이 판정은 어떻게 하는 것일까요? 그것은 삼각형의 법선 벡터를 이용하면 됩니다.
삼각형1과 삼각형2의 법선 벡터와 카메라의 -z축 벡터를 떼어 내어 보면 다음 그림처럼 됩니다.



<삼각형1, 삼각형2, 카메라 -z축>



<삼각형1, 삼각형2, 카메라 -z축>

카메라 -z축 벡터와 삼각형1의 법선 벡터는 예각입니다. 삼각형2의 법선 벡터와 카메라의 -z축 벡터는 둔각입니다.
내적에는 cosine이 있습니다. cosΘ는 [-π/2, π/2]에서는 "+"이고 나머지는 전부 "-"입니다. 이 cosine 성질을 이용해서 삼각형1과 삼각형2에서의 내적을 구하면 삼각형1의 결과는 "+"가 되고 삼각형2의 결과는 "-"가 됩니다.

$$
\begin{aligned}
-\hat{z} \bullet \hat{n}_1 &= -|\hat{z}||\hat{n}_1|\cos(\hat{z},\hat{n}_1) \\
&= -1*1\cos(\hat{z},\hat{n}_1) \\
&= -\cos(\hat{z},\hat{n}_1) >= 0
\end{aligned}
$$

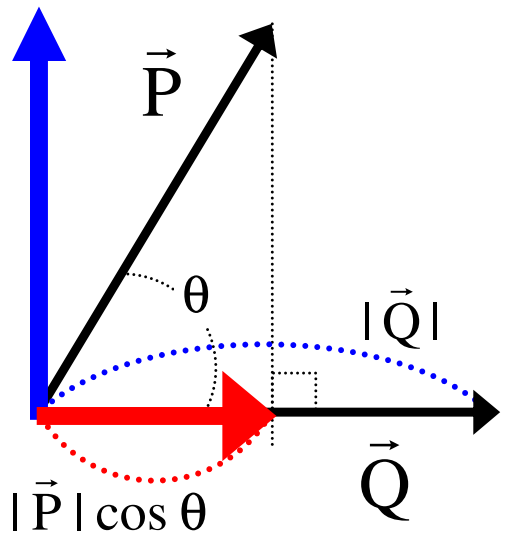
$$
\begin{aligned}
-\hat{z} \bullet \hat{n}_2 &= -|\hat{z}||\hat{n}_2|\cos(\hat{z},\hat{n}_2) \\
&= -1*1\cos(\hat{z},\hat{n}_2) \\
&= -\cos(\hat{z},\hat{n}_2) <= 0
\end{aligned}
$$

그래픽 파이프라인에서 가장 많은 부하가 드는 과정이 픽셀을 만드는 래스터 과정입니다. 내적은 단순 곱셈 과정이어서 수치 계산이 빠릅니다. 래스터 과정 전에 은면 제거를 하고, 은면 제거 방법을 내적을 이용하게 되면 렌더링 속도는 빠르게 될 것입니다.

2.6 벡터 투영

벡터와 행렬에 익숙하지 않은 분들은 이 장은 넘어가도 됩니다.
3D 프로그램 에서는 하나의 벡터를 다른 벡터의 평행 성분과 수직 성분으로 분해할 일이 발생합니다. 이 때 내적을 이용합니다.

만약 다음 그림처럼 벡터 $\vec{P}$를 $\vec{Q}$의 수평 성분과 수직성분으로 분해하는 경우, 그림으로 한다면 평행사변형을 이용합니다.



<한 벡터를 다른 벡터의 수평, 수직 성분으로 분해>

붉은 색 벡터는 $\vec{P}$를 $\vec{Q}$의 수평 성분으로 분해한 벡터이고 파란색 벡터는 $\vec{Q}$의 수직 성분으로 분해한 벡터입니다.

수평 성분은 다음과 같이 구합니다.

$$\vec{P}_{proj\vec{Q}} = |\vec{P}|\cos\theta * \hat{Q} = |\vec{P}|\left(\frac{\vec{P}\bullet\vec{Q}}{|\vec{P}||\vec{Q}|}\right) * \hat{Q}$$

$$= \left(\frac{\vec{P}\bullet\vec{Q}}{|\vec{Q}|}\right) * \hat{Q} = \left(\frac{\vec{P}\bullet\vec{Q}}{|\vec{Q}|}\right) * \frac{\vec{Q}}{|\vec{Q}|}$$

$$= \frac{(\vec{P}\bullet\vec{Q})}{|\vec{Q}|^2}\vec{Q}$$

$$\therefore \vec{P}_{proj\vec{Q}} = \vec{P}\bullet\frac{\vec{Q}\vec{Q}}{|\vec{Q}|^2}$$

$$\therefore \vec{\vec{M}}_{\vec{P}proj\vec{Q}} = \frac{1}{|\vec{Q}|^2}\begin{bmatrix} Q_x{}^2 & Q_xQ_y & Q_xQ_z \\ Q_yQ_x & Q_y{}^2 & Q_yQ_z \\ Q_zQ_x & Q_zQ_y & Q_z{}^2 \end{bmatrix}$$

여기서 $\vec{Q}\vec{Q} = \begin{bmatrix} Q_xQ_x & Q_xQ_y & Q_xQ_z \\ Q_yQ_x & Q_xQ_x & Q_yQ_z \\ Q_zQ_x & Q_zQ_y & Q_zQ_z \end{bmatrix}$ 입니다.

수직성분은 다음과 같이 구합니다.

$$\vec{P}'_{proj\vec{Q}} = \vec{P} - \vec{P}_{proj\vec{Q}}$$

$$= \vec{P} - \frac{(\vec{P}\bullet\vec{Q})}{|\vec{Q}|^2}\vec{Q}$$

$$= \vec{P} - \vec{P}\bullet\frac{\vec{Q}\vec{Q}}{|\vec{Q}|^2}$$

$$= \vec{P}\bullet\left(\vec{\vec{I}} - \frac{\vec{Q}\vec{Q}}{|\vec{Q}|^2}\right)$$

$$\therefore \vec{\vec{M}}'_{\vec{P}proj\vec{Q}} = \vec{\vec{I}} - \frac{\vec{Q}\vec{Q}}{|\vec{Q}|^2}$$

$$= \frac{1}{|\vec{Q}|^2}\begin{bmatrix} 0 & -Q_xQ_y & -Q_xQ_z \\ -Q_yQ_x & 0 & -Q_yQ_z \\ -Q_zQ_x & -Q_zQ_y & 0 \end{bmatrix}$$

위에서 벡터 $\vec{P}$를 $\vec{Q}$의 성분으로 수평 성분, 수직 성분을 구하기 위한 행렬로 바꾸어 보았습니다. 앞서 말했듯이 행렬이 익숙하지 않은 분들은 이 장은 넘어가기 바랍니다.

2.7 벡터의 외적

벡터의 연산 마지막으로 외적에 대한 설명을 시작하겠습니다. 내적까지는 고교 수학에 소개되어 있지만 외적은 아마 나오지 않는 것으로 알고 있고 수학 선생님이 따로 가르쳐 주셨던 기억이 있습니다. 지금도 교과서에 빠져 있다면 저로서는 무척 아쉽습니다.
외적이라는 표현은 Outer Product를 그대로 번역해 놓은 것이며 외적이란 표현 보다 Cross Product 또는 Vector Product 라는 표현을 자주 사용합니다. Cross Product는 외적의 연산 기호를 ("X", Cross) 사용하고 있어서 이 기호의 모양대로 부르는 것이며 Vector Product는 외적의 연산 결과가 벡터이기 때문입니다.
외적의 정의는 벡터 기호를 사용하면 다음과 같습니다.

$$\vec{N} = \vec{a} \times \vec{b} = |\vec{a}||\vec{b}|\sin(\vec{a},\vec{b})\,\hat{n}, \quad \hat{n} : \vec{a} \perp \vec{b} \text{ unit vector}$$

$\vec{a} \times \vec{b}$를 읽는 방법은 vector a cross vector b 혹은 a cross b(에이 크로스 비)라 읽습니다. 그리고 두 벡터의 외적으로 만든 이 수직 벡터를 법선 벡터(Normal Vector)로 부르기도 합니다. N은 Normal의 첫 글자 입니다.
외적을 대수적인 방법으로 표현하면 다음과 같이 됩니다.

$$\vec{C} = \vec{A} \times \vec{B}$$
$$C_k = \sum_i \sum_j a_i * b_j * \varepsilon_{ijk}$$

$$\varepsilon_{ijk} : (\text{Levi-Civita symbol})$$
$$i = j \text{ or } j = k \text{ or } k = i : \varepsilon_{ijk} = 0$$
$$i \to j \to k,\ j \to k \to i,\ k \to i \to j : \varepsilon_{ijk} = 1$$
$$i \to k \to j,\ j \to i \to k,\ k \to j \to i : \varepsilon_{ijk} = -1$$

레비-시비타 기호(Levi-Civita symbol)도 처음 보는 것이니 이렇게 하면 좀 어렵죠? 이렇게 한 것은 외적의 정의를 보여 드리기 위해서였습니다.
위의 표현을 기억하지 않고 위의 결과로 만든 다음과 같은 식을 프로그래머는 이용합니다.

a(a_1, a_2, a_3), **b**(b_1, b_2, b_3)
a X b = ($a_2 * b_3 - a_3 * b_2$, $a_3 * b_1 - a_1 * b_3$, $a_1 * b_2 - a_2 * b_1$)

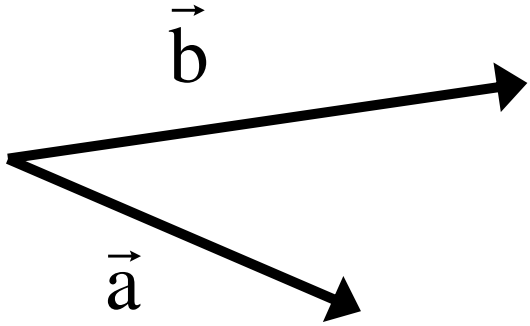
첨자를 1,2,3을 사용했는데 이것을 x, y, z 좌표계에서는 다음과 같이 외적을 구합니다.

a(a_x, a_y, a_z), **b**(b_x, b_y, b_z)

a X b = ($a_y * b_z - a_z * b_y$
, $a_z * b_x - a_x * b_z$
, $a_x * b_y - a_y * b_x$)

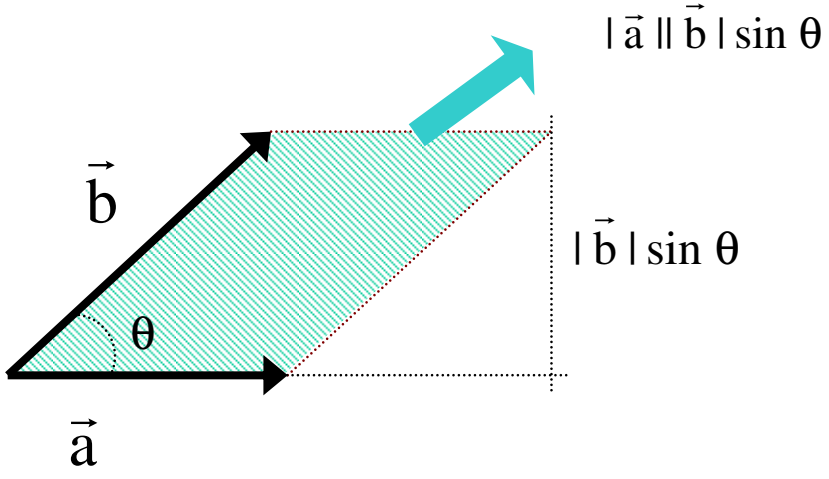
이 값은 $\vec{N} = \vec{a} \times \vec{b} = |\vec{a}||\vec{b}|\sin(\theta)\,\hat{n}$ 을 cosine 제 2법칙을 이용해서 구할 수 있습니다. 증명은 생략하겠습니다. 외적의 이 수식은 이해를 못해도 꼭 기억하기 바랍니다.

다음 그림과 같이 두 벡터가 있을 경우 이 두 벡터의 외적을 구 기하학적으로 구해 봅시다.



<두 벡터의 외적 구하기>

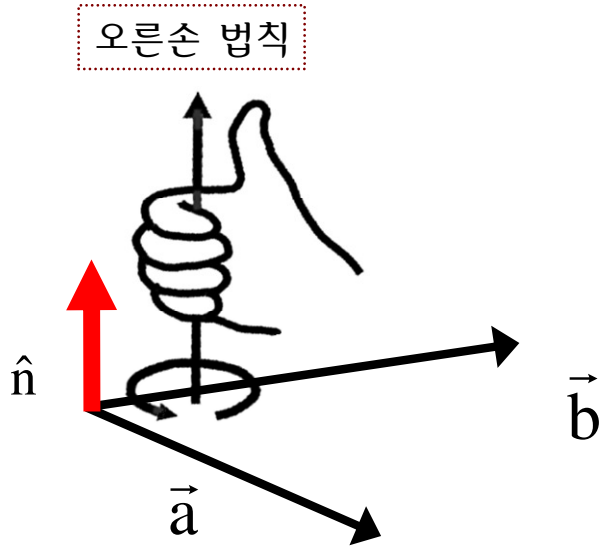
먼저 이 두 벡터가 이루는 평행사변형의 면적을 구합니다.



<두 벡터가 만드는 평행사변형의 넓이 >

다음으로 두 벡터의 수직 벡터에 대한 방향을 구해야 합니다. 두 벡터가 만드는 수직 벡터의 방향

은 2 종류가 수학에서는 벡터 $\vec{a}$를 시작으로 벡터 $\vec{b}$를 오른손으로 감아 쥐고, 오른손 엄지를 곧게 펴서 이 것을 두 벡터의 수직 벡터로 정합니다.

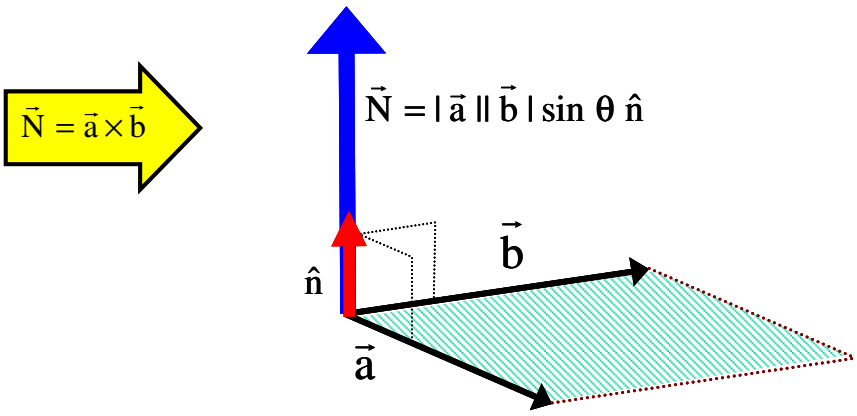


<두 벡터의 수직 벡터 방향 구하기>

오른손 법칙에 의해서 수직방향이 결정되므로 외적을 적용하는 순서가 반대가 되면 벡터의 벡터이 방향이 뒤집히는 Negative가 됩니다. 이 것은 매우 중요한 내용이므로 기억하기 바랍니다.

$$\vec{A}\times\vec{B}=-\vec{B}\times\vec{A}$$

이 방향 벡터에 앞서 구한 평행사변형의 면적을 곱하면 두 벡터의 외적을 구하게 됩니다.



<두 벡터의 외적 구하기- 최종>

외적의 크기는 평행사변형과 같아 만약 같은 두 벡터를 외적하면 0-벡터가 됩니다.

$$\vec{A}\times\vec{A}=\vec{0}\ (\leftarrow \sin\theta=0)$$

앞서 내적은 Projection의 의미가 있다고 했습니다. 외적은 위의 그림처럼 두 벡터의 수직 벡터를 나타냅니다. 또한 외적으로 구한 벡터의 크기는 두 벡터가 만드는 평행사변형 넓이와 일치해서 세 점으로 이루어진 삼각형의 면적을 구할 때도 외적을 이용 합니다.

외적은 3D에서 3개의 점을 가지고 평면의 방정식을 만들 때 평면의 법선 벡터를 구하는데 사용됩니다. 이 법선 벡터는 그래픽 파이프라인에서 은면 제거, 반사 밝기를 정하는 내적으로 사용됩니다.
또한 각종 충돌 알고리듬에서 외적이 자주 나오니 꼭 기억하기 바랍니다.

2.8 외적에 대한 행렬, 행렬식 표현

외적의 행렬과 행렬식 표현은 다음 장의 행렬을 먼저 공부한 다음 다시 보기 바랍니다.
앞서 내적을 이용해서 수평, 수직 성분을 구하는 행렬을 만들어보았습니다. 외적도 마찬가지로 두 벡터 중에 하나의 벡터를 행렬로 만들어 표현할 수 있습니다.

다음 수식은 두 벡터 $\vec{P}$ $\vec{Q}$중 $\vec{Q}$를 행렬로 표현한 것입니다.

$$\vec{C}=\vec{P}\times\vec{Q}$$
$$=(P_yQ_z-P_zQ_y,\ P_zQ_x-P_xQ_z,\ P_xQ_y-P_yQ_x)$$

$$=(P_x,P_y,P_z)\begin{pmatrix}0 & -Q_z & Q_y\\ Q_z & 0 & -Q_x\\ -Q_y & Q_x & 0\end{pmatrix}$$

$$\therefore M_{\vec{P}\times\vec{Q}}=\begin{pmatrix}0 & -Q_z & Q_y\\ Q_z & 0 & -Q_x\\ -Q_y & Q_x & 0\end{pmatrix}$$

이렇게 행렬을 굳이 사용하는 이유는 3D의 그래픽 파이프라인의 정점 변환이 행렬을 이용하고 있

기 때문입니다.

외적의 결과를 항상 기억하기 어려울 때가 있습니다. 행렬식을 알고 있으면 외적의 공식을 좀 더 머리 속에 간직할 수 있습니다. 다음은 행렬식을 통해서 외적의 공식을 유도하는 과정입니다.

$$\vec{C} = \vec{P} \times \vec{Q} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ P_x & P_y & P_z \\ Q_x & Q_y & Q_z \end{vmatrix}$$

$$\begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ P_x & P_y & P_z \\ Q_x & Q_y & Q_z \end{vmatrix}$$

$$= \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} & \hat{x} & \hat{y} \\ P_x & P_y & P_z & P_x & P_y \\ Q_x & Q_y & Q_z & Q_x & Q_y \end{vmatrix}$$

$$= \hat{x}(P_y Q_z - P_z Q_y) + \hat{y}(P_z Q_x - P_x Q_z) + \hat{z}(P_x Q_y - P_y Q_x)$$

과제) 다음과 같은 3차원 벡터 클래스가 주어질 때 벡터의 상등을 비교 연산자 "==" 와 "!="로 구하기 위한 코드를 완성하시오.

```
#include <stdio.h>
#include <math.h>
#include <windows.h>

struct LCXVECTOR3
{
        union { struct { float x; float y; float z; }; float m[3]; };

        // 생성자
        LCXVECTOR3():x(0),y(0),z(0){}
```

```
        LCXVECTOR3(float _x, float _y, float _z):x(_x),y(_y),z(_z){}
        LCXVECTOR3(const LCXVECTOR3& r) {       x =r.x; y =r.y; z = r.z ;       }
        LCXVECTOR3(const LCXVECTOR3* r) {       x =r->x; y =r->y; z = r->z;     }

        // 벡터의 상등을 위한 Operator Overloading
        BOOL operator ==(const LCXVECTOR3& r)
        {
                // 이 부분을 완성하시오.
        }

        BOOL operator !=(const LCXVECTOR3& r)
        {
                // 이 부분을 완성하시오.
        }

        // + Operator
        LCXVECTOR3 operator+() const {  return *this; }
};
```

과제) 위의 클래스에 Operator Overloading을 추가해서 벡터의 덧셈은 "+", Negative 는 "-", 뺄셈은 "-", 스칼라 배는 "*" 연산자로 만들어보시오.

과제) 두 벡터의 내적은 "*", 외적은 "^" 연산자로 구현하시오.

과제) 벡터의 길이를 구하는 함수, 정규화 함수 등도 멤버 함수로 구현하시오.

2.2 행렬(Matrix)

행렬은 어떤 대상을 수학적으로 표현할 때 행과 열의 2차원으로 표현한 것입니다. M x N 행렬은 행이 M, 열이 N인 행렬을 의미합니다.

EX) 행렬의 예

2X3 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 3X2 $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ 3X3 $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$

행렬을 기호로 나타내면 굵은 고딕 대문자를 주로 사용합니다.
EX) 행렬 **M**, **S**, **R**, **T**

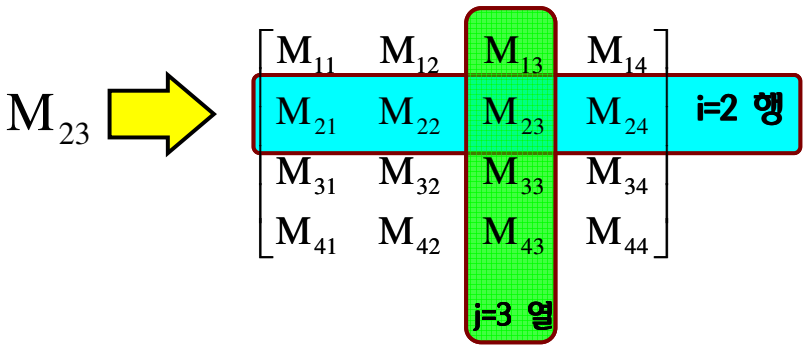
공학, 물리학에서는 벡터와 비슷하게 양방향 화살표를 문자 위에 붙입니다.

EX) $\vec{\mathbf{M}}$, $\vec{\mathbf{S}}$, $\vec{\mathbf{R}}$, $\vec{\mathbf{T}}$

선형대수에서는 위의 행렬의 예처럼 대괄호[], 또는 소괄호() 안에 원소 표시를 합니다. 때로는 중괄호{} 안에 원소에 아래 첨자, 또는 위 첨자를 붙여서 행렬을 나타냅니다.

EX) $\{M_{ij}\}$, S_{ij}, R^{ij}, T^{i}_{j}

4X4 행렬 **M**이 다음과 같이 주어질 때 행렬의 원소는 i는 행을, j는 열인 $\mathbf{M}_{ij}$로 나타냅니다.



<행렬의 행 i=2, 열 j=3 인 원소>

행렬의 여러 표현 중에서 굵은 대문자 또는 양방향 화살표, 소괄호 안에 행렬의 원소를 직접 나타내는 방법을 가장 많이 사용합니다.

2.3 행 벡터(Low Vector), 열 벡터(Column Vector)

행 벡터는 행렬이 하나의 행으로만 구성된 행렬입니다. 2차원, 3차원, 4차원 행 벡터의 예는 다음과 같습니다.

[x, y], (x, y, z), (x, y, z, w)

이 형태는 벡터와 같습니다. 그래서 행 벡터라는 이름이 붙습니다. 이것으로 벡터가 행렬의 특별한 형태임을 간접적으로 알 수 있습니다.

열 벡터는 행 벡터와 반대로 행렬의 구성이 하나의 열로만 구성된 행렬입니다. 행 벡터 예를 열 벡터로 다음과 같이 나타낼 수 있습니다.

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix}, \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}, \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

벡터를 표기할 때 원소를 가로 또는 세로로 써도 상관 없으나 행렬과 연산을 할 때는 가로, 세로의 연산이 달라져 분명하게 표시해야 합니다. 이 부분은 행렬의 연산에서 다시 보겠습니다.

스칼라, 벡터의 정의를 정리하면 스칼라는 1차원 벡터라 할 수 있고, 또한 1행, 1열인 1x1 행렬이 됩니다. 벡터는 행 벡터 또는 열 벡터라 할 수 있고, 이것은 1xN 행렬, 또는 Nx1 행렬 이라 할 수 있겠습니다.

- Bra-ket Notation

행 벡터, 열 벡터를 분명히 구분하기 위해서 지금까지 변수 위에 화살표를 사용한 $\vec{V}$ 과 같은 기호 대신 "<"브라(Bra)와 ">"켓(Ket)을 이용한 브라킷 표기(Bra-ket Notation)을 사용할 때도 있습니다.

이 기호는 물리학자들이 자주 사용하는 기호이며 직관적으로 벡터와 행렬을 이해하는 데 많은 도움이 됩니다. 또한 인쇄물을 작성할 때 수식 기호 도구를 이용하지 않아도 벡터를 편하게 표기할 수 있습니다.

다음은 행 벡터, 열 벡터, 행렬을 이 Bra-ket 기호로 표시한 예제 들입니다.

브라 벡터(행 벡터):

2차원: $\langle V(x,y)| = [V_x \; V_y]$, 3차원: $\langle X(r,\theta,\varphi)| = [X_r \; X_\theta \; X_\varphi]$, n차원: $\langle X| = [X_1 \; X_2 \; ... \; X_n]$

켓 벡터(열 벡터):

2차원: $|V(x,y)\rangle = \begin{bmatrix} V_x \\ V_y \end{bmatrix}$, 3차원: $|X(r,\theta,\varphi)\rangle = \begin{bmatrix} X_r \\ X_\theta \\ X_\varphi \end{bmatrix}$, n차원: $|X\rangle = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_n \end{bmatrix}$

Outer Product: $|a><b| = \begin{bmatrix} a_1*b_1 & a_1*b_2 & \cdots & a_1*b_n \\ a_2*b_1 & a_2*b_2 & \cdots & a_2*b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m*b_1 & a_m*b_2 & \cdots & a_m*b_n \end{bmatrix}$

전치: $|b>^T = <b|, \; <b|^T = |b>$

내적: 내적은 행 벡터 * 열 벡터이므로 $<a|*|b> = <a|b> = \sum a_i b_i$

행 벡터 * 행렬 곱셈 연산:

$<X'| = <X|*|M><N|$

$= \begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} \begin{bmatrix} M_1N_1 & M_1N_2 & M_1N_3 \\ M_2N_1 & M_2N_2 & M_2N_3 \\ M_3N_1 & M_3N_2 & M_3N_3 \end{bmatrix}$

$= \begin{bmatrix} X_1M_1N_1 + X_2M_2N_1 + X_3M_3N_1 & X_1M_1N_2 + X_2M_2N_2 + X_3M_3N_2 & X_1M_1N_3 + X_2M_2N_3 + X_3M_3N_3 \end{bmatrix}$

$= \begin{bmatrix} (X_1M_1 + X_2M_2 + X_3M_3)N_1 & (X_1M_1 + X_2M_2 + X_3M_3)N_2 & (X_1M_1 + X_2M_2 + X_3M_3)N_3 \end{bmatrix}$

$= (X_1M_1 + X_2M_2 + X_3M_3) \begin{bmatrix} N_1 & N_2 & N_3 \end{bmatrix}$

$= (<X|M>)<N|$

행렬 * 열 벡터 연산:

$|X'> = |M><N|*|X>$

$= \begin{bmatrix} M_1N_1 & M_1N_2 & M_1N_3 \\ M_2N_1 & M_2N_2 & M_2N_3 \\ M_3N_1 & M_3N_2 & M_3N_3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}$

$= \begin{bmatrix} M_1N_1X_1 + M_1N_2X_2 + M_1N_3X_3 \\ M_2N_1X_1 + M_2N_2X_2 + M_2N_3X_3 \\ M_3N_1X_1 + M_3N_2X_2 + M_3N_3X_3 \end{bmatrix}$

$$= \begin{bmatrix} (N_1X_1 + N_2X_2 + N_3X_3)M_1 \\ (N_1X_1 + N_2X_2 + N_3X_3)M_2 \\ (N_1X_1 + N_2X_2 + N_3X_3)M_3 \end{bmatrix}$$

$$= \begin{bmatrix} M_1 \\ M_2 \\ M_3 \end{bmatrix} (N_1X_1 + N_2X_2 + N_3X_3)$$

$$= |M><N|X>$$
$$= (<N|X>)|M>$$

행렬 * 행렬 연산: (이 부분의 증명은 생략하겠습니다.)

$$|a><b|*|c><d| \quad = t\,|a><c|\ ,\ (\text{단}, t = <b|c>)$$

2.4 행렬의 연산

행렬의 연산은 벡터보다 상당히 적습니다. 행렬의 연산은 상등, 스칼라 배, 덧셈, 뺄셈, 곱셈 5종류로 이들 연산은 이해가 매우 쉽습니다. 이중에서 행렬의 곱셈만 벡터의 내적, 외적과 더불어 꼭 기억해야 할 내용입니다.

- 행렬의 상등

행렬의 상등은 차원이 같은 두 행렬의 i, j 원소 모두 같을 경우입니다.

$$\forall_{ij}(a_{ij} == b_{ij} \rightarrow \text{true})$$

- 스칼라 배

행렬에 대한 Scalar 배는 행렬의 모든 원소에 Scalar를 곱한 것입니다.

$$k\overleftrightarrow{M} = \{k * M_{ij}\} \text{ or } kM_{ij}$$

$$\overleftrightarrow{M} = \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ M_{21} & M_{22} & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ M_{m1} & \cdots & \cdots & M_{mn} \end{bmatrix} \text{ 에서 } \quad k\overleftrightarrow{M} = \begin{bmatrix} k*M_{11} & k*M_{12} & \cdots & k*M_{1n} \\ k*M_{21} & k*M_{22} & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ k*M_{m1} & \cdots & \cdots & k*M_{mn} \end{bmatrix}$$

- 행렬의 덧셈, 뺄셈

행렬의 덧셈과 뺄셈은 차원이 같은 두 행렬의 각각 대응되는 원소들의 덧셈, 뺄셈을 수행하여 새로운 행렬을 만드는 것입니다. 이것을 대수적으로 표현하면 다음과 같습니다.

$$\overleftrightarrow{M} = \overleftrightarrow{R} \pm \overleftrightarrow{S} \Rightarrow M_{ij} = R_{ij} \pm S_{ij}$$

각각의 모든 원소로 표현하면 다음과 같습니다.

$$\overleftrightarrow{R} = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1n} \\ R_{21} & R_{22} & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ R_{m1} & \cdots & \cdots & R_{mn} \end{bmatrix}, \quad \overleftrightarrow{S} = \begin{bmatrix} S_{11} & S_{12} & \cdots & S_{1n} \\ S_{21} & S_{22} & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ S_{m1} & \cdots & \cdots & S_{mn} \end{bmatrix}$$

$$\overleftrightarrow{M} = \overleftrightarrow{R} \pm \overleftrightarrow{S} = \begin{bmatrix} R_{11} \pm S_{11} & R_{12} \pm S_{12} & \cdots & R_{1n} \pm S_{1n} \\ R_{21} \pm S_{21} & R_{22} \pm S_{22} & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ R_{m1} \pm S_{m1} & \cdots & \cdots & R_{mn} \pm S_{mn} \end{bmatrix}$$

- 행렬의 곱셈

행렬의 곱셈은 행렬의 연산에서 가장 신경을 써야 하는 부분입니다. 만약 행렬 M과 행렬 S의 곱이 성립하려면 M의 행렬의 열과 S의 행이 같은 차원일 때만 성립합니다. 이것은 행렬의 곱셈이 다음과 같은 공식의 룰(rule)을 따르기 때문입니다.

$$\overleftrightarrow{T} = \overleftrightarrow{M} * \overleftrightarrow{S} \quad \Leftrightarrow \quad T_{ij} = \sum_{k} M_{ik} S_{kj}$$

즉, 행렬 **M**이 "any 행 x n 열" 행렬이면 이 행렬과 곱셈이 가능한 행렬 **S**는 "n 행 x any 열" 이어야 합니다. 곱셈 후 최종 행렬의 차원은 "**M**의 행" x "**S**의 열" 이 됩니다.

다음은 위의 식에 대한 2x3, 3x2 행렬 곱셈에 대한 예입니다.

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} g & h \\ i & j \\ k & l \end{bmatrix} = \begin{bmatrix} a*g+b*i+c*k & a*h+b*j+c*i \\ d*g+e*i+f*k & d*h+e*j+f*i \end{bmatrix}$$

2x3, 3x2 행렬의 곱셈 후에 행렬의 차원은 2x2 가 됐습니다.

또 다른 예로 1x3, 3x1 행렬의 곱셈을 봅시다.

$$\begin{bmatrix}4 & 5 & 6\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix}=\begin{bmatrix}4*1+5*2+6*3\end{bmatrix}=\begin{bmatrix}32\end{bmatrix}$$

결과는 1x1 행렬이 되었습니다. 위의 예를 3차원으로 일반화 시키면 다음과 같이 표현할 수 있습니다.

$$\begin{bmatrix}P_x & P_y & P_z\end{bmatrix}\begin{bmatrix}Q_x\\Q_y\\Q_z\end{bmatrix}=\begin{bmatrix}P_x*Q_x+P_y*Q_y+P_z*Q_z\end{bmatrix}=\vec{P}\bullet\vec{Q}$$

결국 벡터의 내적은 행렬에서는 행 벡터, 열 벡터의 곱과 동일합니다.

이렇게 차원이 줄어드는 곱셈도 있지만 다음 예제처럼 반대로 늘어나는 곱셈도 있습니다.

$$\begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}4 & 5 & 6\end{bmatrix}=\begin{bmatrix}1*4 & 1*5 & 1*6\\2*4 & 2*5 & 2*6\\3*4 & 3*5 & 3*6\end{bmatrix}$$

3x1, 1x3 행렬의 곱셈 후 3x3 행렬이 됐습니다. 이런 예는 고등 수학 텐서에서 많이 나오는 형태입니다.
단순하게 행 벡터 곱하기 열 벡터, 또는 열 벡터 곱하기 행 벡터를 통해서 우리는 행렬의 연산 중에 곱셈이 교환 법칙이 성립하지 않음을 직관적으로 알 수 있습니다. 이것은 또한 Bra-ket을 이용하면 바로 증명할 수 있습니다.

$\overleftrightarrow{M}=|a><b|$, $\overleftrightarrow{S}=|c><d|$라 할 때 $\overleftrightarrow{M}*\overleftrightarrow{S}$과 $\overleftrightarrow{S}*\overleftrightarrow{M}$비교해 보면 다음과 같습니다.

$$\begin{aligned}&\overleftrightarrow{M}*\overleftrightarrow{S}=|a><b|*|c><d| && \overleftrightarrow{S}*\overleftrightarrow{M}=|c><d|*|a><b|\\&=|a><b|c><d| && =|c><d|a><b|\\&=|a>(<b|c>)<d| && =|c>(<d|a>)<b|\\&=(<b|c>)|a><d| && =(<d|a>)|c><b|\end{aligned}$$

$$\therefore \overleftrightarrow{M}*\overleftrightarrow{S}\neq\overleftrightarrow{S}*\overleftrightarrow{M}$$

- 항등 행렬(Identity Matrix: 단위(Unit) 행렬

가로 세로의 크기가 같은 행렬을 정방 행렬이라 합니다. 이 정방 행렬 중에서 대각선 모두가 1이고, 나머지는 0인 행렬을 항등 행렬이라 합니다. 이것은 행렬의 곱셈에 대한 항등원과 같아 어떤 행렬에 항등 행렬을 곱해도 항상 자기 자신이 됩니다. 기호는 영문자 **I**를 사용하거나 $\vec{\vec{I}}$ 으로 나타냅니다.

DXSDK에서 D3DXMatrixIdentity() 함수를 이용해서 4x4 행렬을 항등 행렬로 바꿀 수 있습니다.

- 전치 행렬 (Transpose)

행렬의 열과 행을 교환해서 새로운 행렬을 만드는 것입니다. 기호는 T 또는 t를 위 첨자로 사용해서 다음과 같이 표현합니다.

$$M_{ij}{}^{T} = M_{ji}$$

행렬의 곱셈에서는 다음과 같이 곱셈의 순서가 반대로 됩니다.

$$\left(\vec{\vec{M}}\vec{\vec{R}}\vec{\vec{S}}\right)^{T} = \vec{\vec{S}}^{T}\ \vec{\vec{R}}^{T}\ \vec{\vec{M}}^{T}$$

DXSDK의 D3DXMatrixTranspose()함수가 전치 행렬을 만듭니다.

- 행렬식(Determinant)

행렬에서는 행렬식의 특성을 반영하는 "행렬식"이라는 행렬에서 유도되는 Scalar 값이 있습니다. 기호 표현은 **det(M)**, 또는 **|M|** 으로 표현 합니다.

이 것을 수식으로 표현하면 다음과 같습니다.

$\det(\vec{\vec{M}}) = \sum M_{ij} * C_{ij}(\vec{\vec{M}})$, $C_{ij}(\vec{\vec{M}})$: Cofactor - 여인수

$C_{ij} = (-1)^{i+j} * \det(\vec{\vec{M}}_{ij})$, $\det(\vec{\vec{M}}_{ij})$: i행, j열을 제외한 행렬의 행렬식

위의 식이 처음 보는 분들에게 어려움이 있을 수 있습니다. 그런데 우리는 일반적인 프로그램이 아닌 3D 게임 프로그램을 만드는 사람들입니다. 3D 게임 프로그램에서는 4x4 행렬까지만 사용합니다.따라서 2x2, 3x3, 4x4 행렬식만 구하는 방법만 알고 있으면 됩니다.

2x2, 3x3 행렬식을 Sarrus 방법으로 빠르게 구할 수 있습니다. 의 행렬식은 다음과 같이 대각선 방향의 곱셈으로 간단히 행렬식을 만듭니다.

2x2

$$|\overleftrightarrow{M}| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = a*d - b*c$$

<2x2 Sarrus 행렬식>

위의 공식은 고등학교 수학 책에 행렬의 판별 값으로 나와 있습니다.

3x3 행렬식은 다음 그림과 같이 1,2 열을 추가해서 오른쪽으로 내려가는 계산은 "+", 왼쪽으로 내려가는 계산은 "-"을 붙여 행렬식을 계산합니다.

3x3

$$|\overleftrightarrow{M}| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = \begin{vmatrix} a & b & c & a & b \\ d & e & f & d & e \\ g & h & i & g & h \end{vmatrix}$$

$$= aei + bfg + cdh - ceg - afh - bdi$$

$$= a(ei - fh) + b(fg - di) + c(eg - dh)$$

<3x3 Sarrus 행렬식>

행렬식을 공식으로 외우면 잘 안 되는데 Sarrus 방식을 알고 있으면 잊혀지지 않습니다. 이 행렬식을 이용해서 외적도 구할 수 있습니다.

$$\vec{A} \times \vec{B} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} = \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} & \hat{x} & \hat{y} \\ A_x & A_y & A_z & A_x & A_y \\ B_x & B_y & B_z & B_x & B_y \end{vmatrix}$$

$$= \hat{x}A_yB_z + \hat{y}A_zB_x + \hat{z}A_xB_y - \hat{z}A_yB_x - \hat{x}A_zB_y - \hat{y}A_xB_z$$

$$= \hat{x}(A_yB_z - A_zB_y) + \hat{y}(A_zB_x - A_xB_z) + \hat{z}(A_yB_x - A_xB_y)$$

$$= (A_yB_z - A_zB_y,\ A_zB_x - A_xB_z,\ A_yB_x - A_xB_y)$$

행렬식을 알고 있으면 외적을 쉽게 구할 수 있으므로 꼭 기억하기바랍니다.
행렬식을 구하는 DXSDK 함수는 D3DXMatrixDeterminant() 입니다.

- 역 행렬(Inverse Matrix)

정방행렬 어떤 행렬에 행렬을 곱하면 그 결과가 항등 행렬인 행렬을 역 행렬이라 합니다. 역 행렬은 행렬에 위 첨자 -1를 붙여 표시 합니다.

$\overleftrightarrow{M}$ 의 역 행렬 = $\overleftrightarrow{M}^{-1}$

$$\overleftrightarrow{M} * \overleftrightarrow{M}^{-1} = \overleftrightarrow{M}^{-1} * \overleftrightarrow{M} = \overleftrightarrow{I}$$

모든 정방 행렬이 역 행렬을 갖는 것은 아닙니다. 임의의 정방 행렬에서 역 행렬을 구하는 것은 상당히 어려운 부분입니다. 그런데 3D에서 가장 많이 사용하는 행렬은 이동, 회전, 크기 변환 세 종류 입니다. 이 행렬들의 역 행렬은 간단하게 구할 수 있습니다.

먼저 역 행렬의 의미는 앞서 항등 행렬을 만드는 행렬이라고 했습니다. 이것은 행렬의 변환 전으로 돌아 오는 것과 같습니다. 만약 이동을 행렬로 표현한다면 이의 역 행렬은 원래대로 돌아오는 행렬이 될 것이며 크기 변환 행렬은 원래 크기로 되돌리는 행렬이 될 것입니다.
회전 행렬 또한 회전의 위치로 돌아와야 합니다.
그렇다면 이동에 대한 역 행렬은 이동의 방향을 반대로 하면 간단하게 구해질 수 있습니다. 즉 방향을 반대로 적용한 이동 행렬이 이동에 대한 역 행렬이 됩니다.
크기 변환 행렬의 역 행렬은 크기 변환 값 Scale 을 1/Scale로 적용하면 이 행렬이 역 행렬이 됩니다.
회전 행렬의 역 행렬은 회전 각도를 반대로 적용한 행렬이 회전 행렬의 역 행렬이 됩니다.

이렇게 모든 일이 다 쉬우면 좋겠지만 위의 경우는 특수한 경우 이므로 일반적으로 구할 때는 가우스 소거법이나 수반 행렬을 이용해서 구합니다.
행렬의 차원이 높으면 가우스 소거법으로 역 행렬을 구하는 것이 유리하지만 3D프로그램은 4x4 이하 행렬을 사용하기 때문에 다음과 같은 수반 행렬을 이용한 역 행렬을 사용합니다.

$$\overleftrightarrow{A}^{-1} = \frac{1}{|\overleftrightarrow{A}|}\,\mathrm{adj}(\overleftrightarrow{A})$$

$$\mathrm{adj}(\overleftrightarrow{A}) = \begin{bmatrix} |A_{11}| & |A_{12}| & \cdots & |A_{1n}| \\ |A_{21}| & |A_{22}| & \cdots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ |A_{m1}| & \cdots & \cdots & |A_{mn}| \end{bmatrix}^T$$

$$|A_{ij}| = (-1)^{i+j} * M_{ij} (M_{ij} : \overleftrightarrow{A}\text{의 } i,j\text{를 제외한 소행렬식})$$

욕심 같아서는 위의 수식을 설명하고 싶지만 다른 선형 대수학 책에 잘 설명이 되어 있으므로 넘어 가도록 하고 DXSDK의 역 행렬 구하는 함수를 소개하겠습니다.
D3DXMatrixInverse() 함수는 역 행렬을 구하는 함수로 4x4 행렬을 필요로 합니다. 이 함수를 통해서 행렬식(Determinant)도 얻을 수 있는데 잘 사용을 안 하므로 NULL 인수를 전달합니다.

EX)D3DXMatrixInverse() 함수 사용법

```
D3DXMATRIX      mtR;
...
D3DXMATRIX      mtRI;
D3DXMatrixInverse(&mtRI, NULL, &mtR);
```

행렬의 곱에 대해서 역 행렬을 적용하면 곱의 순서가 다음과 같이 바뀝니다.

$$\left(\overleftrightarrow{M}\overleftrightarrow{R}\overleftrightarrow{S}\overleftrightarrow{T}\right)^{-1} = \overleftrightarrow{T}^{-1}\overleftrightarrow{S}^{-1}\overleftrightarrow{R}^{-1}\overleftrightarrow{M}^{-1}$$

이 성질은 실제 컴퓨터에서 각각의 역 행렬 $\overleftrightarrow{T}^{-1}$, $\overleftrightarrow{S}^{-1}$, $\overleftrightarrow{R}^{-1}$, $\overleftrightarrow{M}^{-1}$을 구해서 전체 행렬의 역 행렬을 구하는 것 보다 모든 행렬을 곱한 후에 이의 역 행렬을 구하는 방법-$\left(\overleftrightarrow{M}\overleftrightarrow{R}\overleftrightarrow{S}\overleftrightarrow{T}\right)^{-1}$-이 훨씬 빠름을 암시하고 있습니다.

과제) float형을 멤버로 하는 4x4 행렬 클래스를 작성하시오.
과제) 앞의 행렬 클래스에 연산자 다중 정의(Overloading)을 적용해서 항등, 덧셈, 뺄셈, 곱셈을 구현하시오.