UNITY
-CAHPTER5-

SOUL SEEK

목차

기본 이론

1. 벡터의 정의

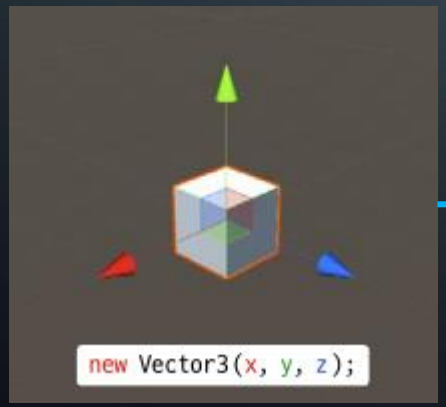
- 물리학자, 공학자, 게임 개발자 : 공간상의 방향으로 사용된다.
  ➔ex)(10, 5, 0)은 오른쪽으로 10단위, 위쪽으로 5단위 만큼 이동하는 방향

- 빅데이터를 관리하는 프로그래머 : 나열된 데이터를 묶는 단위.
  ➔ex(172, 64)은 키 172, 몸무게 64를 나타내는 데이터

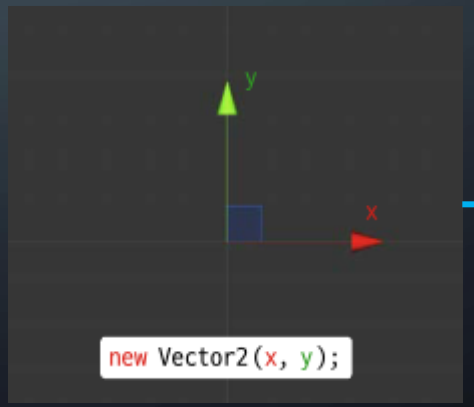
- 수학자 : 벡터 연산을 만족하고 정해진 개수의 원소를 가지면 무엇이든 벡터.

게임 개발자에게는 주로 위치, 방향, 속도를 나타내는 데 사용된다.
➔ 벡터라는 데이터 형식과 특징과 속성을 사용해서 어떻게 이용할 수 있는지는 분야마다 다르다.
➔ 표현법을 활용한 정의 → 수학에서 벡터 연산이 만족하기 위해서는 정해진 개수의 원소를 가져야 한다. 이부분을 활용하여 3D벡터는 평면에 입체의 폭을 더하여 공간을 만들 수 있기 때문에 3개의 원소를 가지는 x, y, z의 벡터를 하나로 하여 3가지의 방향을 표현한다.

유니티에서는 3D벡터를 나타내는 Vector3를 사용해서 3D 공간에서의 x, y, z 좌표를 표현한다.
2D벡터는 Vector2를 이용해서 2D공간에서 필요한 x, y 좌표를 표현한다.



Vector3는
3차원 공간에 대응



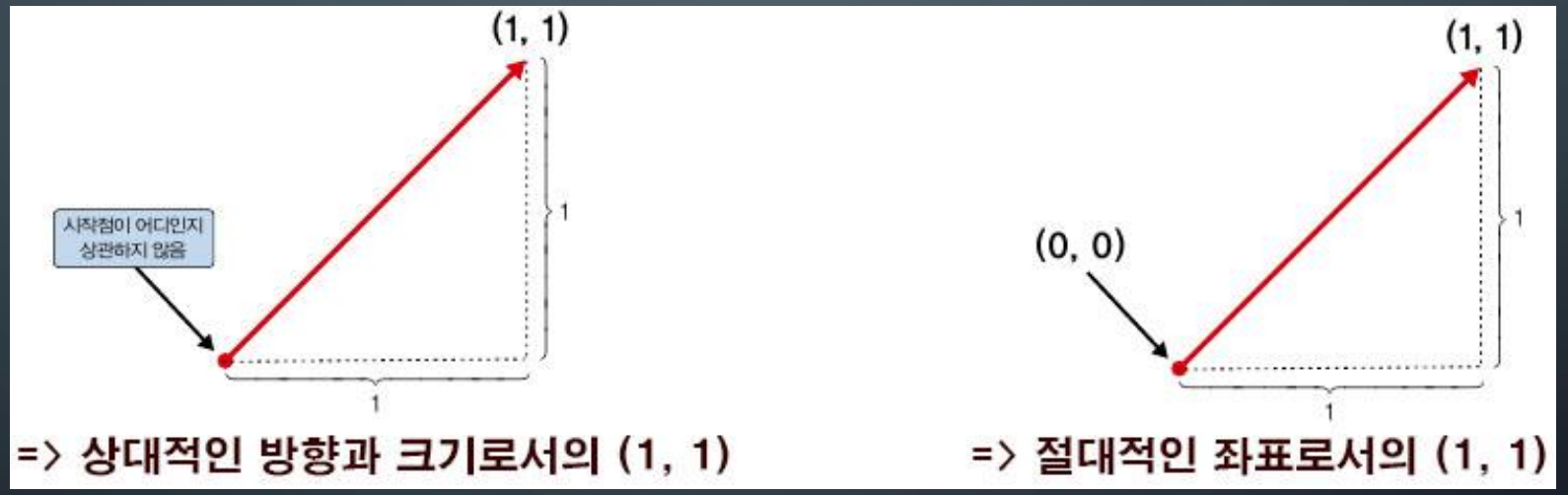
Vector2는
2차원 공간에 대응

2. 절대 위치와 상대 위치

벡터는 방향과 크기를 가지고 있고 화살표의 방향과 길이로 이를 표현할 수 있다.
→절대적인 기준 없이 (현 위치에서) 상대적으로 어느 방향으로 얼마만큼의 크기(길이)로 갈 것인지를 표현한다.

ex) (1 ,1)이란 **Vector2**는 월드에서 두가지 의미를 가질 수 있다.

상대좌표 : 내가 어디 있는지 모르겠지만 현재 좌표에서 **(1, 1)**만큼 더 가려고 한다. → **Local**
절대좌표 : 게임 세상 속에서 나의 좌표가 **(1, 1)**이다. → **World**

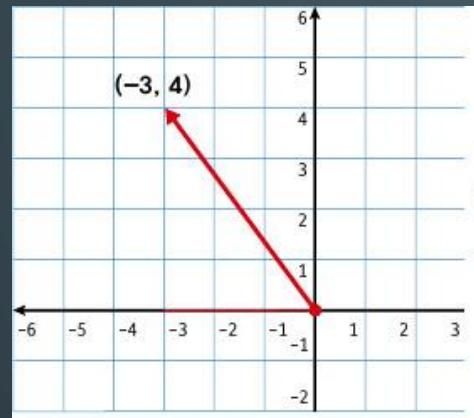


벡터는 두 가지 상태를 나타낸다.

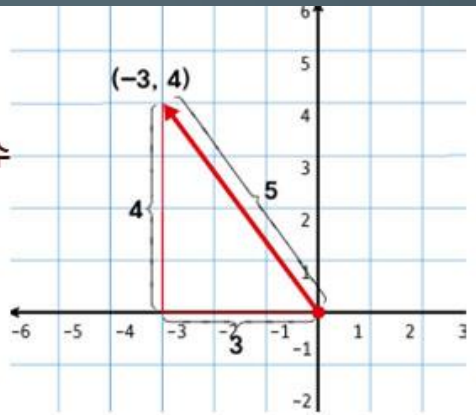
- **절대적 좌표**
- **상대적인 방향과 크기(길이)**

3. 벡터의 크기

2D 벡터 **(-3, 4)**를 표현하면



(−3, 4)를 향하는 화살표로 벡터를 나타낼 수 있다. 방향은 직관적으로 나타낼 수 있지만 크기는 직관적으로 나타나지 않는다.

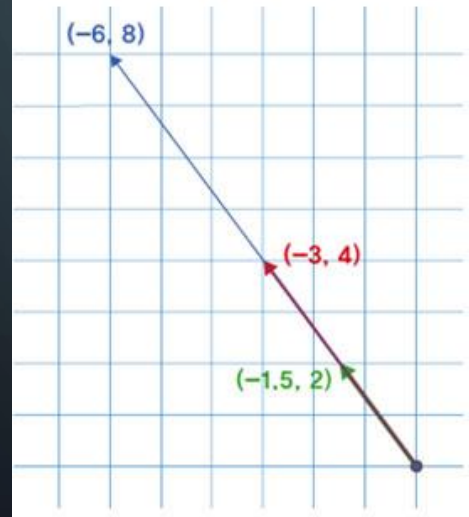


크기는 수학적인 도움을 받아 두 좌표를 밑변과 높이로 사용하는 직각 삼각형을 만들고 피타고라스의 정의를 이용하면 구할 수 있다.

$$(-3,4)\text{의 크기} = \sqrt{(-3)^2 + 4^2} = 5$$

벡터의 크기는 화살표의 길이에 대응하므로 음수가 될 수 없다.

$$\text{벡터의 크기} = \sqrt{x^2 + y^2 + z^2}$$



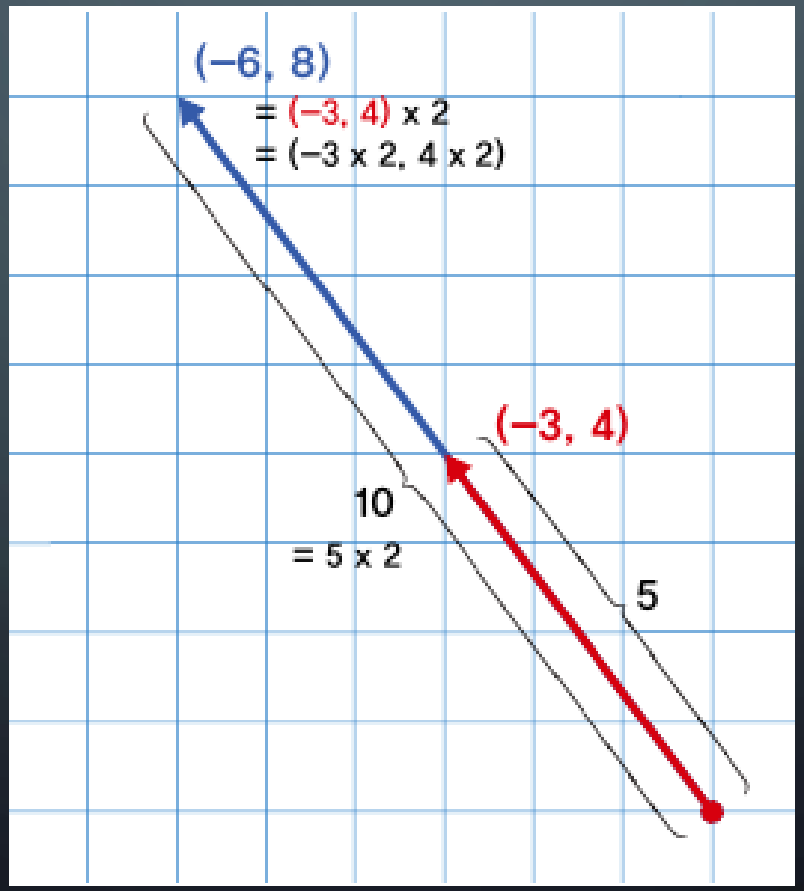
벡터의 방향과 크기는 별개의 값이라는 것으로 방향은 같지만 크기만 다른 벡터가 얼마든지 존재할 수 있다는 의미가 된다. 즉, 방향은 같지만 크기만 크기가 다른 벡터를 셀 수 없이 많이 만들 수 있다.

4. 벡터의 스칼라 곱

(-6, 8)은 각 원소들이 **(-3, 4)**의 **2**배 값을 가지고 있다.

- 벡터의 원소들에게 **동일한 배수를 계산**에서 크기 값의 배율을 조정할 수 있다.
- 보통 이렇게 각 원소에 배율을 조정하면 결국 벡터의 크기 값의 배율도 함께 변한다.
  ➔그렇기 때문에 **방향은 변하지 않는다.**

- 전체적인 원소의 변화로 크기만 변하기 때문에 **벡터의 스칼라 곱**이라고 부른다.
  ➔이런 속성은 **Scale** 값을 조정할 때 사용된다.



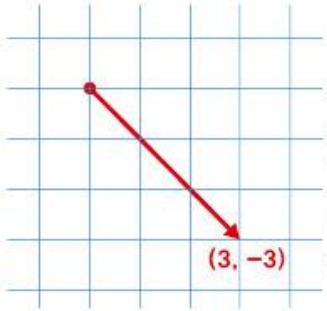
5. 방향벡터(NORMALIZED VECTOR)

벡터는 방향과 크기를 가지고 있기 때문에 여러가지로 사용할 수 있다.

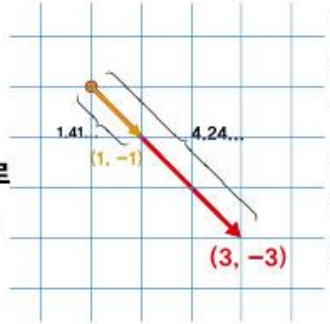
➔속도로 사용하면 벡터의 화살표 방향은 이동하려는 방향이 되고 화살표 길이는 속력(이동거리)이 된다. **(방향) x (속력 또는 이동거리)**

➔속도를 표현한다면 일반적인 벡터 **x** 스칼라 값이 아닌 일반 벡터를 정규화**(Normalized)**하여 사용해야 올바르게 표현된다.

➔**속도 = 방향벡터 x 스칼라 곱**



(3, -3)이라는 2D 벡터를 생각해보자.
스칼라 곱을 이용하면 (3, -3)은 (1, -1)의 3배에 해당한다는 것을 알고 있다.
그렇기 때문에 (3, -3)은 (1, -1)의 방향으로 (1, -1)의 3배의 속력으로 가고 있다고 표현할 수 있다. (3, -3) = (1, -1) x 3



(3, -3) = (1, -1) x 3은 (3, -3)의 속력이 (1, -1)의 3배라는 뜻이 된다. 하지만 (3, -3)의 속력 자체는 표현하지 못하고 있다.
길이를 속력으로 사용하고 있는데 실제 (3, -3)의 속력은 3이 아니다.
물론 (1, -1)도 속력은 1이 아니다.
그래서 우리가 표현하고 있는 일반적인 (3, -3), (1, -1)은 방향벡터로 그대로 사용할 수 없다.

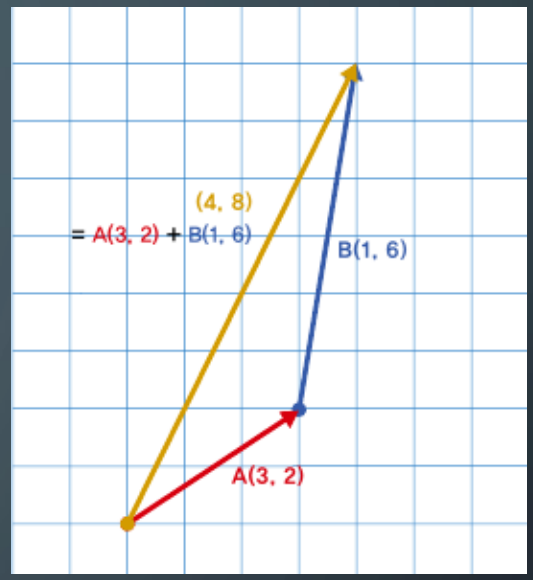
'A의 방향으로 B만큼의 속력' 을 '벡터 A x 스칼라 B' 로 표현했을 때 벡터 A의 크기가 1이 아니면 '벡터A x 스칼라B' 로 표현된 속도의실제속력은 크거나 작은 문제가 나타난다. 그렇기 때문에 크기가 1인 벡터로 정규화해서 배수로 사용할 수 있는 형태로 변형을 하게 되는데 이것을 방향벡터 또는 노말벡터 라고 부른다.
크기가 1이므로 방향은 같지만 크기가 서로 다른 벡터를 비교하는 기분으로 삼을 수 있다.
만약, (1, -1)의 방향벡터를 구하고 싶다면 벡터의 시작점에서 반지름이 1인 원을 그려서 (1, -1)의 원 바깥부분을 잘라낸다. 이말의 의미는 원은 중심점과 동일한 거리(크기)에 있는 점들의 모임이라는 특징을 그대로 가지고 있다는 것이다.

벡터의 정규화 = 벡터의 크기를 1로 만드는 것

6. 벡터의 덧셈과 뺄셈

벡터의 덧셈

- 두 벡터의 같은 성분끼리 합하면 된다.
- 덧셈 첫 번째 항이 시작 위치가 출발, 두 번째 항이 이동 위치, 결과가 도착 위치



Ex) A(3, 2)와 **B(1, 6)**의 합.

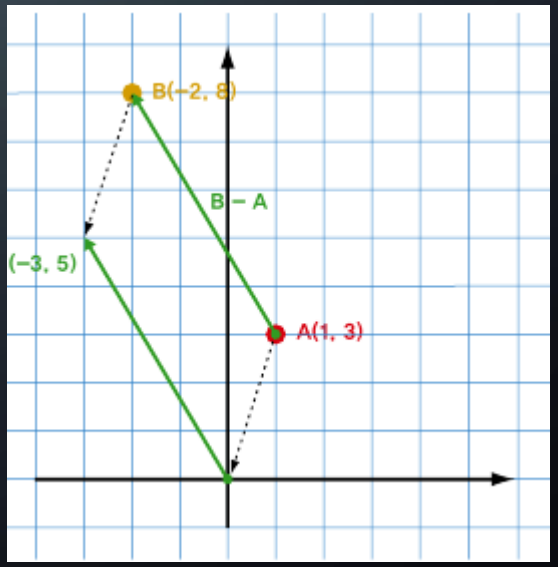
→**A + B = (3, 2) + (1, 6) = (3+1, 2+6) = (4, 8)**

→두 벡터의 합은 공강상에서 보면 **A**만큼 이동한 상태에서 **B**만큼 더 이동한다는 의미이다.

→현재위치 **(3, 2)**에서 **(1, 6)**만큼 이동해서 도착한 위치가 **(4, 8)**이다.

벡터의 뺄셈

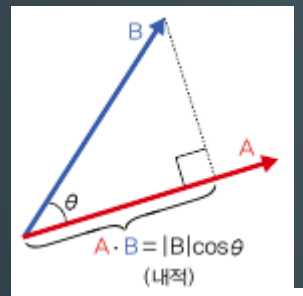
두 벡터의 같은 성분끼리 빼면 된다.
두 벡터의 간격을 말한다.
두 번째 항에서 첫 번째 항으로 가기 위한 방향과 거리



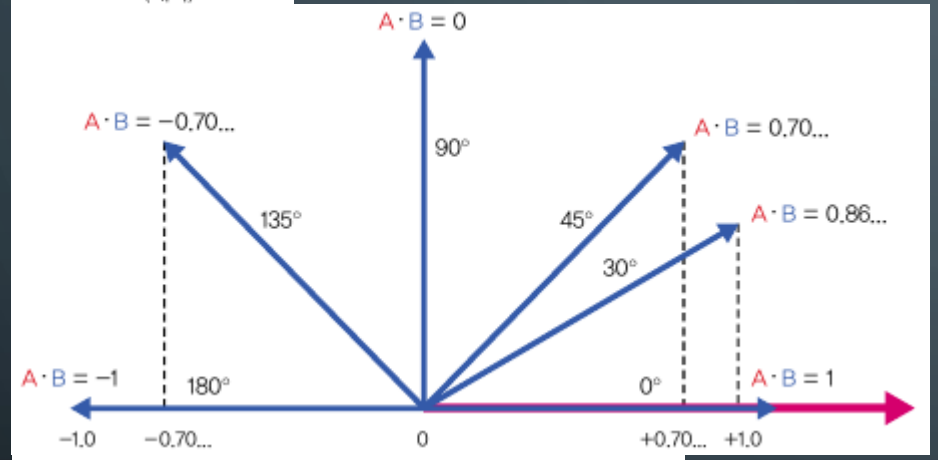
B – A = A에서 **B**까지의 간격 **= A**에서 **B**까지의 방향과 거리
(목적지**) – (**현재 위치**) =** 현재 위치에서 목적지까지의 방향과 거리

6. 벡터의 내적

- 어떤 벡터를 다른 벡터로 '투영'하는 연산 → 점 연산**(dot product)**라고 부른다.
- 두 벡터 **A**와 **B**사이의 내적은 **A • B**로 표현한다.



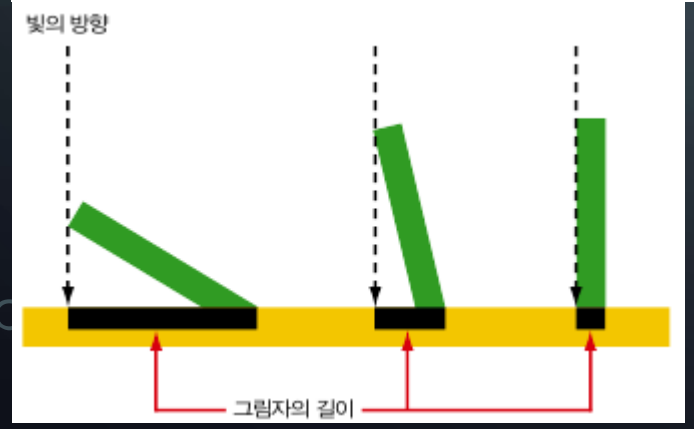
- 내적 공식이나 각도에 따른 결과는 무조건 외울 필요는 없다.
- 다만 내적으로 할 수 있는 것들에 대한 이야기는 이런 공식과 결과 도출 과정에서 일어나는 일들을 가지고 응용해서 나타나는 것들이다.
- 공식보다 중요한 것은 내적의 의미에 있다.
- **A**와 **B**의 내적은 벡터 **B**를 벡터 **A**의 지평선으로 끌어내리 는 연산이다.



벡터 **B**의 끝점이 수직 낙하하듯이 벡터 **A**로 끌어내려지고 있다. 즉, 상대방 벡터가 자신의 지평선으로 끌어내려 졌을 때(투영 되었을 때) 가지는 길이가 내적의 결과가 된다.

→자신과 상대방 사이의 각도가 벌어질수록 투영된 길이가 짧아지는 현상을 이용해 둘 사이의 각도를 알 수 있기 때문에 내적을 사용한다.

→해시계를 비유하여 생각하면 빛이 수직으로 비춰질수록 그림자가 짧아진다.

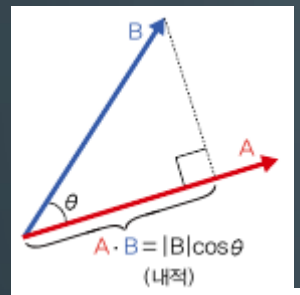


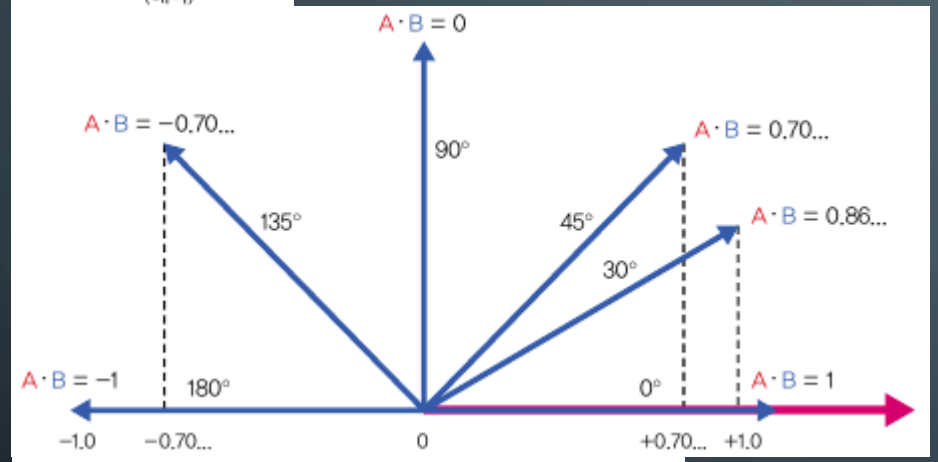
두 벡터의 방향이 일치하면 내적의 결과가 **1**이 되었다가, 두 벡터 사이의 각도가 증가하면 점점 내적 값이 줄어드는 걸 볼 수 있다.

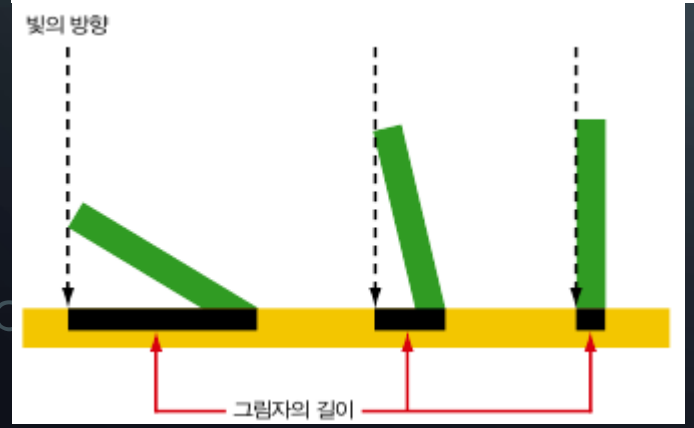
둘 사이의 각도	내적 결과
0°	+1
0° ~ 90°	+1 ~ 0
90°	0
90° ~ 180°	0 ~ −1
180°	−1

6. 벡터의 내적

- 어떤 벡터를 다른 벡터로 '투영'하는 연산 → 점 연산**(dot product)**라고 부른다.
- 두 벡터 **A**와 **B**사이의 내적은 **A • B**로 표현한다.







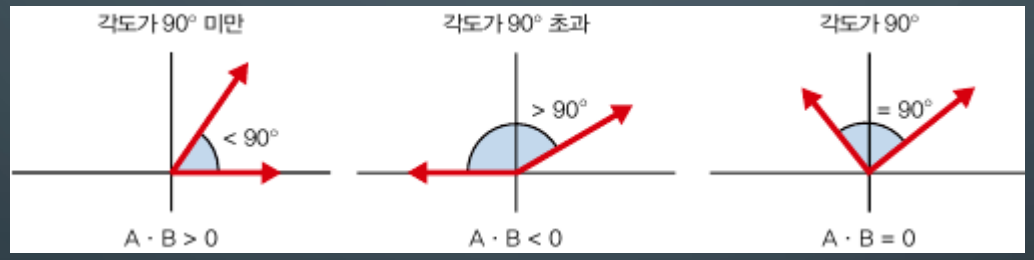
- 내적 공식이나 각도에 따른 결과는 무조건 외울 필요는 없다.
- 다만 내적으로 할 수 있는 것들에 대한 이야기는 이런 공식과 결과 도출 과정에서 일어나는 일들을 가지고 응용해서 나타나는 것들이다.
- 공식보다 중요한 것은 내적의 의미에 있다.
- **A**와 **B**의 내적은 벡터 **B**를 벡터 **A**의 지평선으로 끌어내리 는 연산이다.

- 벡터 **B**의 끝점이 수직 낙하하듯이 벡터 **A**로 끌어내려지고 있다. 즉, 상대방 벡터가 자신의 지평선으로 끌어내려 졌을 때(투영 되었을 때) 가지는 길이가 내적의 결과가 된다.
  →자신과 상대방 사이의 각도가 벌어질수록 투영된 길이가 짧아지는 현상을 이용해 둘 사이의 각도를 알 수 있기 때문에 내적을 사용한다.
  →해시계를 비유하여 생각하면 빛이 수직으로 비춰질수록 그림자가 짧아진다.

6. 벡터의 내적

- 두 벡터의 방향이 일치하면 내적의 결과가 **1**이 되었다가**,** 두 벡터 사이의 각도가 증가하면 점점 내적 값이 줄어드는 걸 볼 수 있다**.**
- 두 벡터가 서로 수직이 되면 내적 값은 **0**이 되고**,** 그 이상 각도가 더 벌어지면 내적 값은 음수가 된다**.**
- 두 벡터가 반대 방향을 가리키게 되면**(**둘 사이의 각도가 **180**도가 되면**)** 내적 결과는 **-1**이 된다**.**

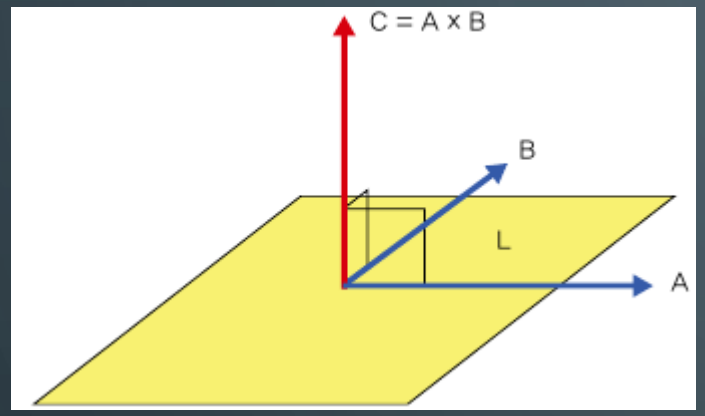
둘 사이의 각도	내적 결과
0°	+1
0° ~ 90°	+1 ~ 0
90°	0
90° ~ 180°	0 ~ −1
180°	−1



- 내적을 이용하면 어떤 물체 사이의 각도가 얼마만큼 벌어져 있는지 쉽게 파악할 수 있다**.**
- 게임에서 내적을 사용하면 탱크의 몸체와 탱크의 포신이 얼마만큼 벌어졌는지**,** 플레이어의 시선 방향과 플레이어가 실제로 이동하는 방향 사이의 각도가 얼마나 벌어졌는지 등을 파악할 수 있다**.**

7. 벡터의 외적

- 두 벡터를 모두 수직으로 통과하는 동일방향의 벡터를 구하는 연산
- 벡터 곱 또는 교차 곱**(cross product)**로 부른다**.** → **A X B**
- 당장의 공식의 이해보다 그 결과가 무엇이며 어떻게 활용할 수 있는지 알아야 한다**.**
- 공간상의 화살표로 표현해야 할 정도로 당장 머리속으로 떠오르지 않는 내용이다**.**



벡터 **A**와 벡터 **B**를 외적한 결과인 **C**를 나타내고 있다**.**

벡터 **A**와 벡터 **B**는 평면 **L**에 속한다**.**
벡터 **C**는 벡터 **A**와 벡터 **V** 모두에 수직이다**.** 즉**,** 두 벡터 사이의 외적 결과는 두 벡터에 모두 수직인 벡터이다**.**
A X B = C이고 **B X A = -C**가 된다 그러므로 교환법칙은 성립하지 않는다**.**→ 직각여부와 함께 방향을 알려준다**.**

외적을 이용하면 어떤 표면에 수직인 방향을 구할 수 있다**.**
→표면 역시 점과 선이 모여서 만들어진 것이고 평면의 정의에 의해 평면에 속하는 두 벡터의 외적은 평면상의 모든 점**,** 선과 모두 수직이므로 평면과도 수직인 결과가 된다**.**
→평면이 바라보는 방향을 나타낸다**.**

어떤 평면과 수직이라서 해당 평면의 정방향이 향하는 방향을 나타내는 벡터를 노말벡터**(Normal Vector)** 또는 법선벡터라고 한다**.**

방향, 크기 - UNITY C# 벡터

1. VECTOR 타입

Vector2, Vector3, Vector4 Type을 지원하고 있다.
→new Vector2(x, y); new Vector3(x, y, z); new Vector4(x, y, z, w);

```
Vector3 a = new Vector3(1, 2, 3); //(1, 2, 3)벡터 생성
//(1, 2, 3)
a.x = 10;
a.y = 20;
a.z = 30;
```

Vector 타입은 내부 클래스가 아닌 구조체로 선언되어 있다.

```
public struct Vector3{
    public float x;
    public float y;
    public float z;
}
```

만약, **Vector3**가 클래스(참조 타입)이라면

```
Vector3 a = new Vector3(0, 0, 0);
Vector3 b = a;
b.x = 100;
```

하면 **b.x**가 **100**이 되면서 **a.x**도 바뀌게 된다.

실제코드에서 이와 같은 사실을 확인 할 수 있으며, 그 결과로 구조체, 값 타입임을 확인 할 수 있다.

2. VECTOR3 연산

스칼라 곱

Vector3 * 스칼라;

```
Vector3 b = new Vector3(3, 6, 9);
b = b * 10; //a는 (30, 60, 90);
```

벡터의 덧셈과 뺄셈

Vector3 + Vector3;

```
Vector3 a = new Vector3(2, 4, 8);
Vector3 b = new Vector3(3, 6, 9);

Vector3 c = a - b;//c는 (-1, 2, -1)
```

Vector3 – Vector3;

```
Vector3 a = new Vector3(2, 4, 8);
Vector3 b = new Vector3(3, 6, 9);

Vector3 c = a + b;//c는 (5, 10, 17)
```

벡터의 정규화(방향벡터)

Vector3.normalized;

```
Vector3 a = new Vector3(3, 3, 3);
Vector3 b = a.normalized;//b는 대략(0.6, 0.6, 0.6)이 됨
```

2. VECTOR3 연산

벡터의 크기

Vector3.magnitude;

```
Vector3 a = new Vector3(3, 3, 3);
float b = a.magnitude;//b는 대략 5.19
```

벡터의 내적

Vector3.Dot(c, d);

```
Vector3 c = new Vector3(0, 1, 0);//위쪽으로 향하는 벡터
Vector3 d = new Vector3(1, 0, 0);//오른쪽으로 향하는 벡터
float e = Vector3.Dot(c, d);//수직인 벡터끼리 내적하면 결과는 0
```

벡터의 외적

Vector3.Cross(f, g);

```
Vector3 f = new Vector3(0, 0, 1);//앞쪽(z) 방향벡터
Vector3 g = new Vector3(1, 0, 0);//오른쪽(x) 방향벡터
//외적 결과 c는 앞쪽과 오른쪽 모두에 수직인 위쪽(y) 방향 벡터
Vector3 h = Vector3.Cross(f, g);//c는 (0, 1, 0)
```

3. VECTOR3 응용

두 지점 사이의 거리

```
//현재 위치(currentPos)에서 목적지(destPos)를 향해 10만큼 이동한 위치 구하기
Vector3 currentPos = new Vector3(1, 0, 1); // 현재 위치
Vector3 destPos = new Vector3(5, 3, 5); // 목적지

//currentPos에서 destPos으로 향하는 방향벡터
Vector3 direction = (destPos - currentPos).normalized;

//목적지를 향해 10만큼 현재 위치에서 이동한 새로운 위치
Vector3 newPos = currentPos + direction * 10;
```

제공된 **Distance() Method** 사용하기

```
// 현재 위치(currentPos)에서 목적지(destPos)까지의 거리 구하기
Vector3 currentPos = new Vector3(1, 0, 1);
Vector3 destPos = new Vector3(5, 3, 5);

// currentPos에서 destPos로 향하는 벡터
Vector3 delta = destPos - currentPos;

// currentPos에서 destPos까지의 거리(크기)
float distance = delta.magnitude;
```

현재 위치 에서 목적지로 향하는 방향

(destPos - currentPos).normalized;

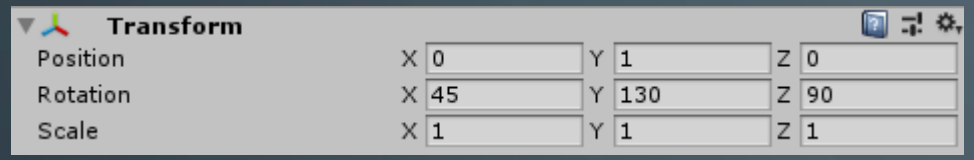
```
Vector3 currentPos = new Vector3(1, 0, 1); // 현재 위치
Vector3 destPos = new Vector3(5, 3, 5); // 목적지

// currentPos에서 destPos까지의 거리
float distance = Vector3.Distance(currentPos, destPos);
```

회전 – UNITY C# 쿼터니언

1. UNITY의 회전

- **Quaternion**은 회전을 나타내는 타입이다.
- **Inspector View**에서 **Transform Component X, Y, Z**를 가지는 **Vector3**값을 대입하게 되어있다.



- **Rotation**의 대입 값은 **Vector**로 보여지지만 사용할 때는 **Quaternion Type**으로 변환해서 사용한다.
  → **Code**에서 사용할 때 착각할 경우 컴파일 에러가 발생한다.

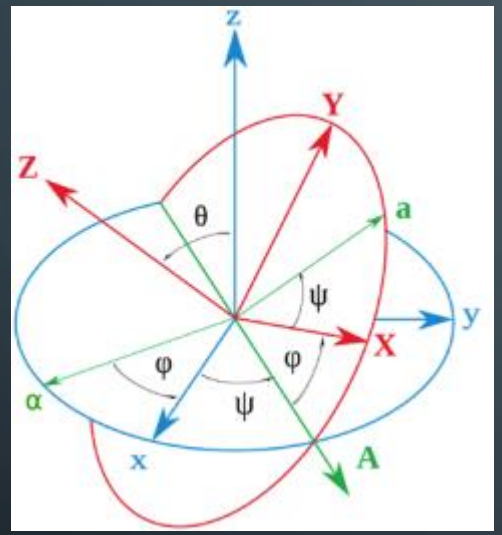
```
transform.position = new Vector3(0, 0, 10);
transform.localScale = new Vector3(1, 1, 1);

//rotation은 Vector3 타입이 나닌 Quaternion 타입이므로 에러 발생
transform.rotation = new Vector3(30, 60, 90);
```

Quaternion이라는 값은 사용하는 사람들이 직접 알아보기에는 굉장히 직관적이지 못하다 그래서 **Inspector View**서 우리가 사용할 때는 직관적으로 보이는 육십분법 표기를 사용하고 적용을 할 때 **Quaternion**으로 변환해서 적용시켜 준다.

2. 짐벌락(GIMBAL LOCK)

- **3D Vector**를 이용하여 회전을 표현하는 오일러 각의 문제점이 있기 때문에 **Quaternion**을 사용한다.
  →수학자가 오일러가 고안한 표현법
  →물체가 회전하기 전의 좌표계에서 회전한 다음의 좌표계로 바뀌려면 기존 좌표계를 세번에 걸쳐 각각 얼 마만큼 회전시키면 되는지 세 각도로 물체의 회전을 표현한다.
  →회전하기 전 상태에서 회전한 다음 상태가 되려면 세 방향으로 나누어 각각 얼마만큼 회전하면 되는지 계산하여 회전을 표현하는 방식

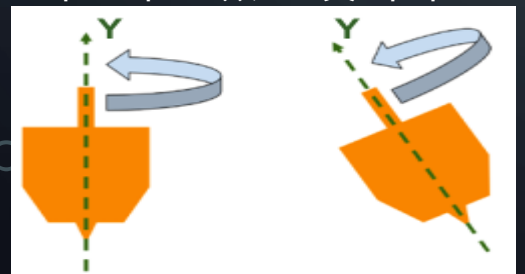


오일러각의 문제점.

회전을 한번에 계산하지 않고, 세 번 나누어서 순서대로 계산하기 때문에 축이 겹치는 문제가 생긴다.

ex) 오일러각에서 **(30, 60, 90)** 회전은
Z축을 기준으로 **90**도 회전
X축을 기준으로 **30**도 회전
Y축을 기준으로 **60**도 회전 순서로 계산된다.

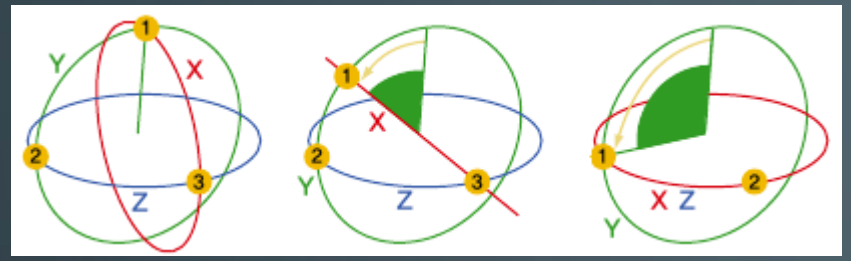
물체의 **Y**축은 '물체가 서 있는' 방향이라고 생각 할 수 있다. 똑바로 서있는 물체가 **Y**축을 기준으로 **60**도 회전하는 것과 서있지 않고 기울어진 물체가 **Y**축을 기준으로 회전 하는 것은 실제로는 다른 방향으로 회전하고 있는 것이다.



이미 **Z**축으로 조금 기울어진 회전된 상태인데 그 상태에서 **Y**축 회전은 **Z**축 회전의 영향을 받는다. 그러므로 오른쪽 팽이는 **Y**축 회전은 왼쪽 팽이의 **Y**축 회전과 다르다.
회전 순서에 의해 뒤로 갈수록 다른 축의 영향을 받게 되므로 마지막 순서의 회전은 자유도를 상실하고 마지막 축의 회전을 사용할 수 없게 되는 현상이 발생

2. 짐벌락(GIMBAL LOCK)

- 오일러각에 의해 발생하는 축이 잠기는 현상을 말한다



Y축을 기준으로 **90**도 회전하는 과정을 보여주고있다**.**

1번위치에서 **Y**축 기준으로 회전을 하면 **X**축도 물려 있기 때문에 **X**축도 회전을 한다**.** 그러다 **Y**축을 기준으로 **90**도 회전하면 끌려온 **X**축이 **Z**축과 겹치게 된다**.**
그 순간 **X**축 회전을 진행하면 **Z**축 회전을 하는 것과 차이가 없어지게 된다**.**

즉**,** 회전을 진행하게 되면 절차와 값은 다르지만 같은 회전이 되는 경우가 생긴다**.**
X축으로 **30**도 회전 후 **Z**축으로 **30**도 회전
X축으로 **60**도 회전
Z축으로 **60**도 회전

이 상황에서는 이미 축회전에 **X, Z**의 축이 겹쳤기 때문에 축 정보 하나가 사라져 더 이상 삼차원 회전을 제대로 할 수 없게 된다**.**

짐버락 현상은 어떤 축을 90도 회전할 때 특히 자주 발생한다. 그렇기 때문에 아주 오래 전에 만들어진 시뮬레이션 게임들은 **90**도 회전을 사용하지 않고 **89.9x**도 같은 값으로 회전을 처리 하기도 했다**.**

3. 쿼터니언(QUATERNION)

- 사원수라고도 부르고 **x, y, z, w 4**개의 원소를 사용한다.
- **‘**한 번에 회전하는 ' 방식이기 때문에 오일러각과 달리 짐벌락 현상이 없으며 **90**도 회전을 제대로 표현할 수 있다**.**
- 게임에서 회전을 구현할 때 사용한다**.**
  →직관적이지 않고 복잡한 계산 단계를 가지고 있기 때문에 직관적인 오일러각의 값들을 받고 내부에서 쿼터니언으로 처리를 지원하고 있다**.**
  →**Unity Code** 상에서 쿼터니언을 직접 생성하고**,** 쿼터니언 내부를 직접 수정하는 것을 허용하지 않고**, Vector3 Type**의 오일러각이나 다른 참고 값을 사용해 쿼터니언을 쉽게 생성하는 **Method**를 제공한다**.**

새로운 회전 데이터 생성

```
Quaternion.Euler(new Vector3(0, 0, 0));
```

Vector3 값을 이용해서 새로운 **Quaternion**값을 생성할 수 있다**.**

```
//Y축을 중심으로 하는 60도 회전을 표현하고 있다.
Quaternion rotation = Quaternion.Euler(new Vector3(0, 60, 0));
```

회전을 Vector3(오일러각)로 가져오기

```
Quaternion rotaion = Quaternion.Euler(new Vector3(0, 60, 0));

//Vector3 타입의 값으로(0, 60, 0)이 나온다.
Vector3 eulerRotation = rotation.eulerAngles;
```

3. 쿼터니언(QUATERNION)

현재 회전에 회전을 더하기

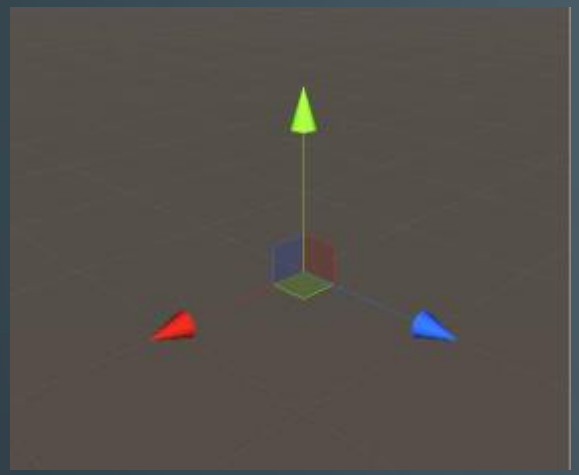
```
Quaternion a = Quaternion.Euler(30, 0, 0);
Quaternion b = Quaternion.Euler(0, 60, 0);

//a만큼 회전한 상태에서 b만큼 더 회전한 회전값을 표현
Quaternion rotation = a * b;
```

- 회전을 추가하는 거라면 기본적인 사칙연산은 **+**가 되어야 하는데 *로 연산하고 있다.
- 이것은 **3D**공간을 구성하는 파이프라인에서 정의된 공간변환**(**월드변환**)**에서 정해진부분이다.
  → **Direct3D** 와 **OpenGL**에서의 **3D** 렌더링 파이프라인에서 월드를 구성하거나 변환할 때 행렬을 사용해서 복잡한 연산을 처리한다. 그렇기 때문에 변환에 적용될 행렬에 회전 값이 적용되어야 하므로 정의된 곱셈연산을 하게 된다.

공간과 움직임

1. 유니티 공간



좌표계

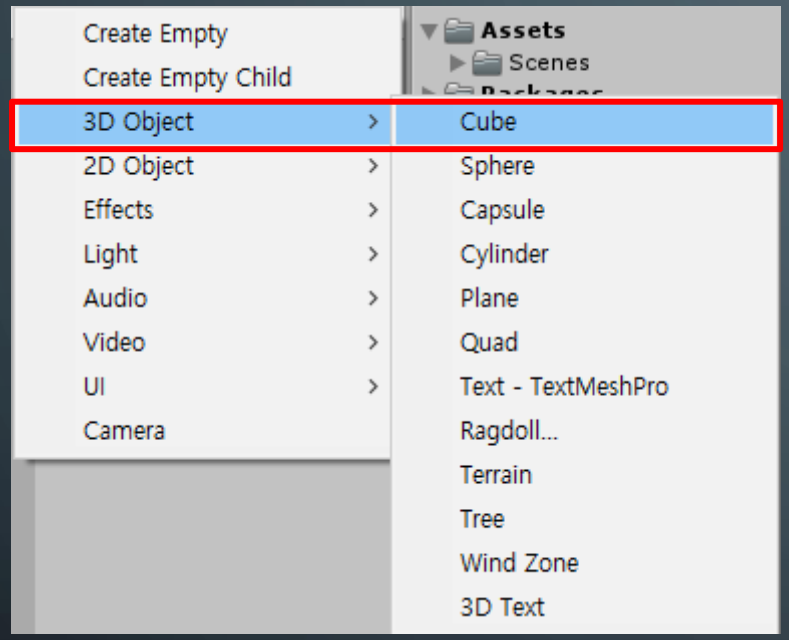
- **3D**공간에 배치하는 **3D** 오브젝트는 위치를 표현한 값**,** 즉 좌표를 가진다**.**
- 원점을 기준으로 **(x, y, z)**에서 **X**방향은 오른쪽**, Y** 방향은 위쪽**, Z**방향은 앞쪽으로 이동한 위치**.**
- 좌표를 측정할 기준이 될 원점의 위치와 **X, Y, Z**축 방향을 설정하여 물체가 어디에 배치되어 있는지 표현하는 기준과 체계를 좌표계라 부른다**.**
- **'**어떤 방향으로 **'** 얼마만큼 이동한 거리에 배치할 것인지 결정하는 기준**.**

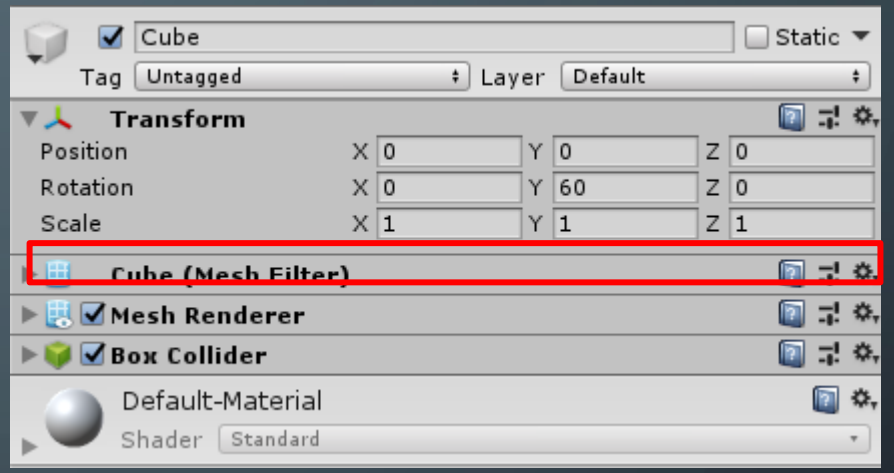
공간

좌표계를 사용하여 물체를 배치하는 틀이며**,** 어떤 좌표계를 사용 하느냐에 따라 공간의 종류가 달라진다**.**
게임 월드는 하나지만 하나의 좌표계**(**공간**)**로는 게임 월드의 모든 속성을 표현할 수 없다**.**
→전역 공간**,** 오브젝트 공간**,** 자식 공간은 하나의 게임 월드를 서로 다른 좌표계로 관측하여 표현한 것이다**.**
→예를 들어 전역공간에서는 게임 월드의 앞쪽 방향은 나타낼 수 있지만 자식 오브제트가 부모 오브젝트에 상대적으로 움직이는 방향은 나타낼 수 없다**.**
→오브젝트 공간에서는 어떤 오브젝트의 앞쪽 방향은 알 수 있지만 게임 월드의 앞쪽 방향은 알 수 없다**.**

2. 전역(GLOBAL) 공간

- 월드의 중심이라는 절대 기준이 존재하는 공간이며 월드 공간이라고 부르기도 한다.
- **X, Y, Z** 방향을 정하고 좌표를 계산하는 기준을 전역 좌표계라고 한다.
- 게임 월드의 원점**(0, 0, 0)**이 존재하며 모든 오브젝트가 원점에서 얼마만큼 떨어져 있느냐가 오브젝트의 좌표가 된다.





Unity 상단에 **Transform Gizmo Toggle** 버튼이 있다. 여기에서 **Local / Global** 전환 버튼을 눌러서 확인 해보자.

2. 전역(GLOBAL) 공간

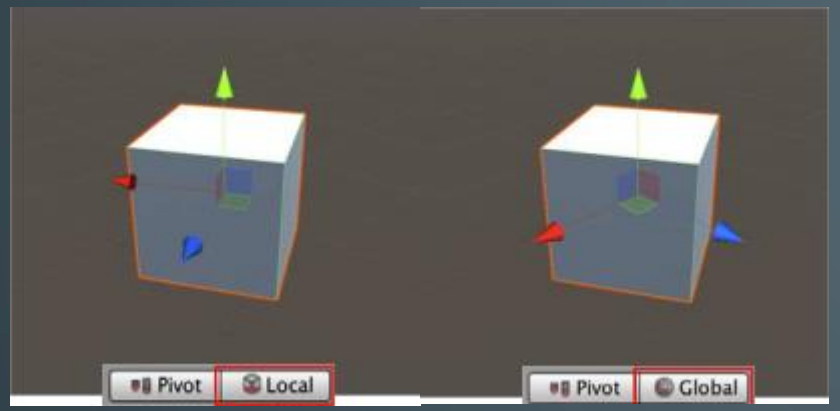
Pivot / Center

- 기본값은 **Pivot**이고, **Local / Global** 전환 버튼의 기본값은 **Local**이다. 평상시에는 기본값인 **Pivot / Local**을 사용한다.
- **Pivot**은 오브젝트의 실제 기준점을 기준으로 오브젝트를 배치한다.
- **Center**는 눈으로 보이는 중점을 기준으로 오브젝트를 배치한다.

일반적으로 사람형태의 **3D** 모델은 발바닥 근처를 기준점으로 모델링 한다. 그러한 **3D** 모델을 유니티로 가져와서 **Pivot**모드로 **Scene**에서 보면 **Gizmo**가 모델링의 원래 기준점으로 배치된다.

센터 모드로 전환하면 오브젝트의 실제 기준점은 무시하고 겉으로 보이는 형태의 중심에 **Gizmo**가 표시된다.

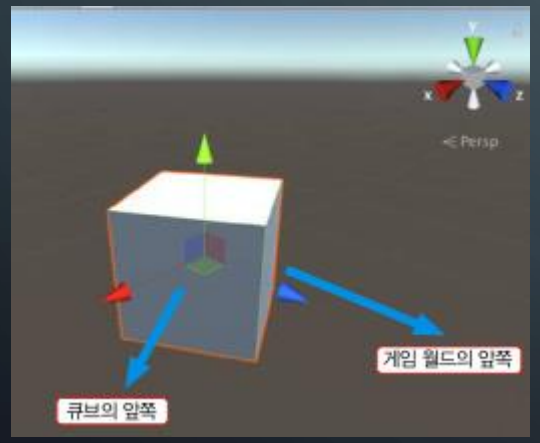
2. 전역(GLOBAL) 공간



Toggle 버튼의 **Local**과 **Global**을 왔다 갔다 전환할 때마다 **Gizmo**가 변한다.

Global로 전환되면 **Gizmo**가 **Cube**가 회전한 것을 완전히 무시하고 게임 월드를 기준으로 바뀌게 된다.
→**Y**축 기준으로 **60**도 회전된 큐브를 배치할 때 **Global Mode**에서는 **Cube**회전을 무시하고 게임 월드를 기준으로 배치된다.

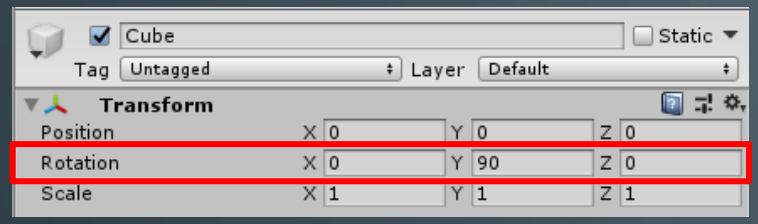
- 전역 좌표계에서는 특정 오브젝트 공간의 **X, Y, Z** 방향이 아니라 전역 공간의 **X, Y, Z** 방향을 좌표계의 기준 방향으로 삼는다.
- **Global Mode**에서 **Cube**를 '앞쪽'으로 옮기면 큐브의 앞쪽이 아니라 게임 월드의 기준에서 앞쪽으로 옮긴다는 의미.

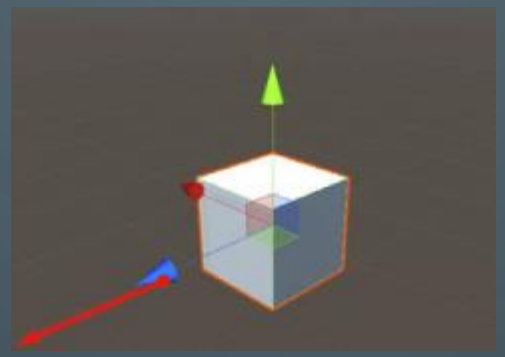


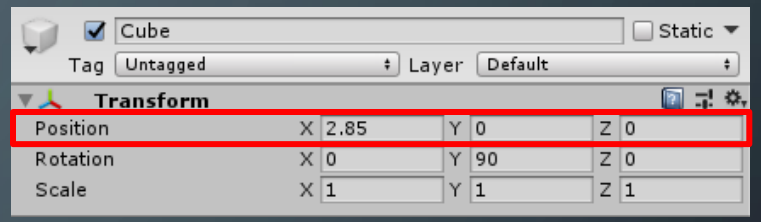
Global 공간에서의 앞쪽과 오브젝트 공간에서의 앞쪽은 각 면의 앞쪽이지만 **Transform**에서 수치를 대입해서 확인해보거나 직접 **Gizmo**를 조작하여 변화를 확인하면 차이점을 쉽게 알 수 있다.

2. 오브젝트 공간

- **Local**이라고 표시되어 있지만 실제로는 오브젝트 공간이다.
- 쉬운 접근성을 위해 오브젝트 공간의 일부 개념을 **Local**에 포함 시켜 사용하기 때문이다.
- 오브젝트 공간은 자기 자신이 기준이 된다. 따라서 물체가 앞쪽으로 평행이동 할 때 스스로의 방향을 기준으로 평행이동 한다. → 월드의 앞쪽과 다르다.







- **Toggle**을 **Local**로 전환해서 **Y**축으로 **90**회전 후 **Z**축을 향하는 **Gizmo**를 움직이면 **Z**값 대신 **X**값이 변하는 모습을 볼 수 있다.

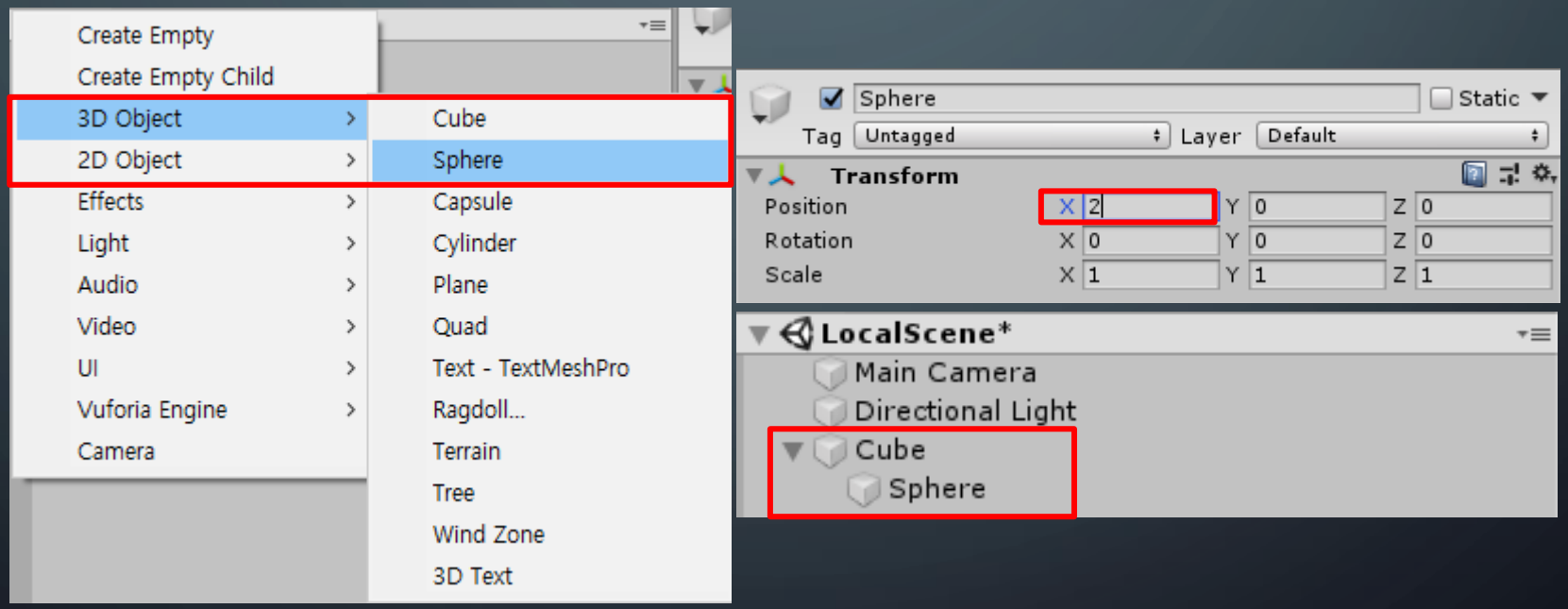
- **Cube**의 **Y**축 회전을 **90**도로 설정하면 **Global**공간의 **X**축과 **Cube**에 대한 오브젝트 공간의 **Z**축이 일치하게 된다. → 오브젝트의 '앞쪽' == 게임월드의 '오른쪽'

오브젝트 공간과 전역 공간은 다르게 적용되고 있으며 Inspector View에서 볼 수 있는 오브젝트 공간을 측정하고 있는 것이 아니라는 것을 알 수 있다.
오브젝트 공간은 오브젝트 자신을 원점으로 삼는다. 따라서 게임 월드에서의 오브제트 실제 위치가 어디든 상관없이 평행이동은 오브젝트는 가만히 있고, 주변 풍경이 움직이는 것으로 이해할 수 있다.

배가 앞으로 움직이는 것이 아닌 지구를 뒤로 움직여서 배가 움직이는 것처럼 보이는 것.

3. 지역(LOCAL) 공간

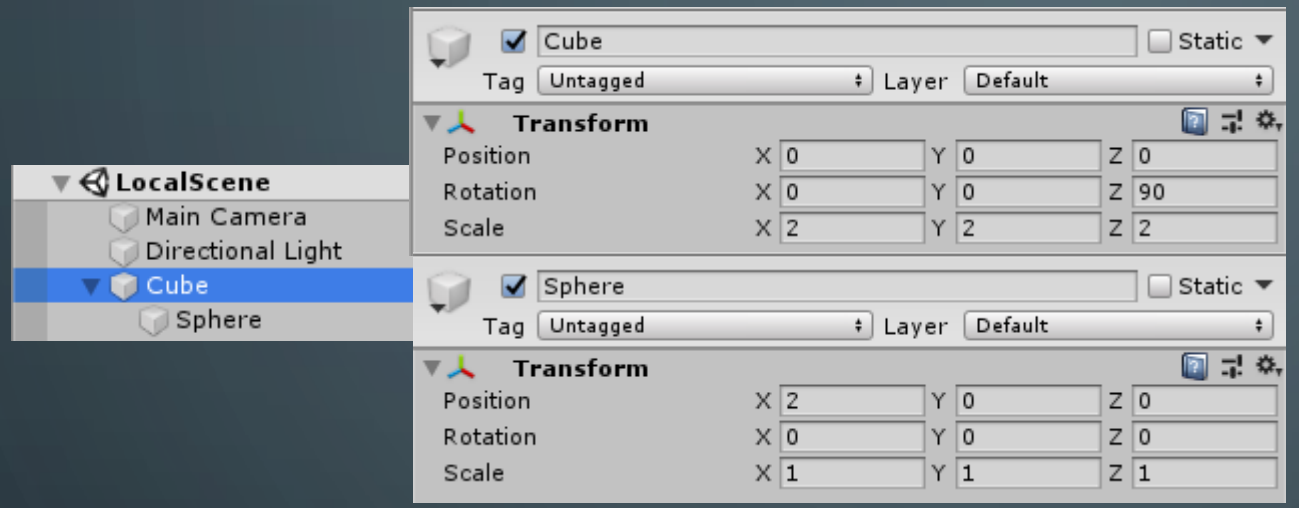
- **Inspector View**에 표시된 위치는 전역 공간을 기준으로 측정된 것이다.
  →부모 오브젝트가 존재하지 않으면 지역 좌표계와 전역 좌표계가 일치하기 때문에 지역 공간상의 위치가 곧 전역 공간상의 위치가 된다.
- 게임 월드나 오브젝트 자신이 아닌 자신의 부모 오브젝트를 기준으로 한 지역 좌표계로 좌표를 측정한다.
  →부모가 상위 **Root**이기 때문에 월드상에 존재하는 좌표를 나타내고 하위 자식들은 각자의 부모 아래의 지역적인 좌표 안에서 위치하고 있기 때문이다.
- **Inspector View**에 표시되는 게임 오브젝트의 위치, 회전, 스케일은 모두 지역 공간에서 측정된 값이다.



- **Sphere Position**을 **(2, 0, 0)**으로 변경하고 **Cube**의 자식으로 추가 해보면 부모가 있기 때문에 **Transform**은 지역공간을 표현하게 된다.

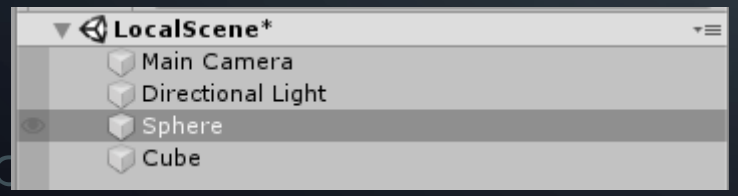
3. 지역(LOCAL) 공간

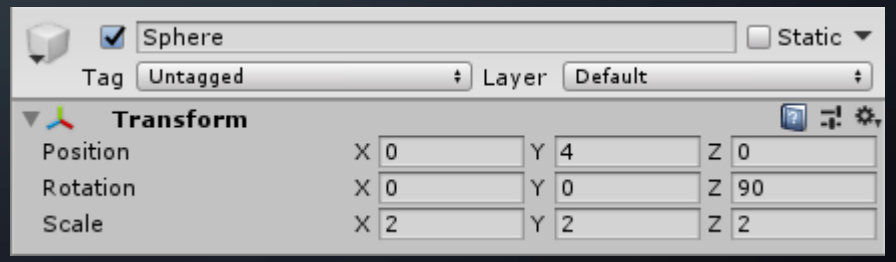
- 부모 **GameObject**가 없는 경우에 한해서 **Inspector View**에 표시되는 지역 위치, 지역 회전, 지역 스케일이 전역 공간에서의 전역 위치, 전역 회전, 전역 스케일과 일치하게 된다.



Cube에서 위치와 **Scale**을 바꿔도 **Sphere**의 위치와 **Scale**이 바뀌지 않는다.

- **Sphere**를 **Cube**에서 빼서 **Root**로 만들어 버리면 **Cube**에 의해서 이미 스케일이 늘어난 상황이기 때문에 **(2, 2, 2)**로 스케일 값이 조정된다.





4. 오브젝트의 이동과 회전

Script로 전역과 지역공간을 구분하여 움직여 보자**.**

자기 자신의 **Transform**은 직접 쓸 수 있으니 자식의 **Transform**을 받을 수 있게 하고 위치와 회전값을 설정한다**.**

```
public Transform childTransform; //움직일 자식 게임 오브젝트의 트랜스폼

void Start()
{
    //자신의 전역 위치를 (0, -1, 0)으로 변경
    transform.position = new Vector3(0, -1, 0);
    //자식의 지역 위치를 (0, 2, 0)으로 변경
    childTransform.localPosition = new Vector3(0, 2, 0);
    //자신의 전역 회전을 (0, 0, 30)으로 변경
    transform.rotation = Quaternion.Euler(new Vector3(0, 0, 30));
    //자식의 지역 회전을 (0, 60, 0)으로 변경
    childTransform.localRotation = Quaternion.Euler(new Vector3(0, 60, 0));
}
```

키보드 위쪽 방향으로 초당 **1**의 속도로 평행이동**,** 자신을 **Z**축 기준으로 초당 **180**도**,** 자식을 **Y**축 기준으로 초당 **180**도 반시계 방향으로 회전**.**

```
if(Input.GetKey(KeyCode.UpArrow))
{
    //위쪽 방향키를 누르면 초당(0, 1, 0)의 속도로 평행이동
    transform.Translate(new Vector3(0, 1, 0) * Time.deltaTime);
}
```

```
if(Input.GetKey(KeyCode.LeftArrow))
{
    //왼쪽 방향키를 누르면
    //자신을 초당 (0, 0, 180)회전
    transform.Rotate(new Vector3(0, 0, 180) * Time.deltaTime);
    //자식을 초당 (0, 180, 0)회전
    childTransform.Rotate(new Vector3(0, 180, 0) * Time.deltaTime);
}
```

4. 오브젝트의 이동과 회전

전역 평행이동과 지역 평행이동

Translate() Method에 의한 평행이동은 전역공간이 아니라 지역 공간을 기준으로 이루어 진다.

ex) transform.Translate(0, 0, 1);을 실행하면 자신의 앞쪽 방향으로 **1**만큼 움직인다.
→게임 월드의 앞쪽과 다른 방향으로 움직일 수도 있다는 의미이다.

두 번째 매개변수 **Space Type**을 받아 평행이동의 기준을 전역 공간으로 할지 지역 공간으로 할지 지정할 수 있다.

전역 공간 기준으로 평행이동

```
transform.Translate(new Vector3(0, 1, 0) * Time.deltaTime, Space.World);
```

지역 공간 기준으로 평행이동

```
transform.Translate(new Vector3(0, 1, 0) * Time.deltaTime, Space.Self);
```

Rotate() Method에 역시 전역 공간 기준 회전과 지역 공간 기준 회전을 이용할 수 있다.

전역 공간 기준으로 회전

```
transform.Rotate(new Vector3(0, 0, 180) * Time.deltaTime, Space.World);
```

지역 공간 기준으로 회전

```
transform.Rotate(new Vector3(0, 0, 180) * Time.deltaTime, Space.Self);
```

5. 벡터 연산으로 평행이동

벡터의 프로퍼티

get형식으로만 되어 있고 **set**을 이용할 수 없다.
자주 사용되는 **Vector3** 값은 프로퍼티 형태로 제공하고 있다.

```
//Vector3 position = new Vector3(0, 1, 0);과 같은 동작
Vector3 position = Vector3.up;
```

Vector3.forward : new Vector3(0, 0, 1);
Vector3.back : new Vector3(0, 0, -1);
Vector3.right : new Vector3(1, 0, 0);
Vector3.left : new Vector3(0, 1, 0);
Vector3.down : new Vector3(0, -1, 0);

모두 크기가 **1**인 벡터이다.

Transform의 방향

transform.forward : 자신의 앞쪽을 가리키는 방향벡터
transform.right : 자신의 오른쪽을 가리키는 방향벡터
transform.u : 자신의 위쪽을 가리키는 방향벡터

자신의 뒤쪽 **: -1 * transform.forward**
자신의 왼쪽 **: -1 * transform.right**
자신의 아래쪽 **: -1 * transform.up**