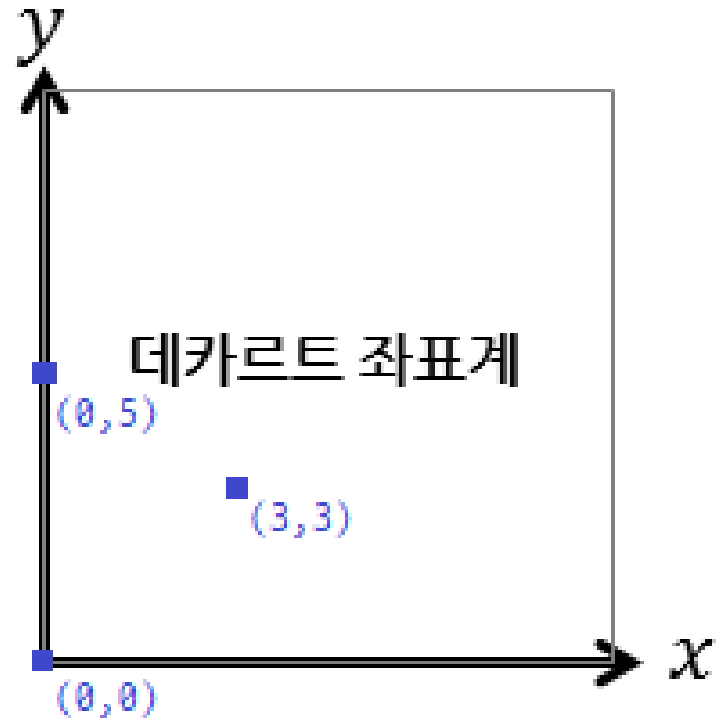
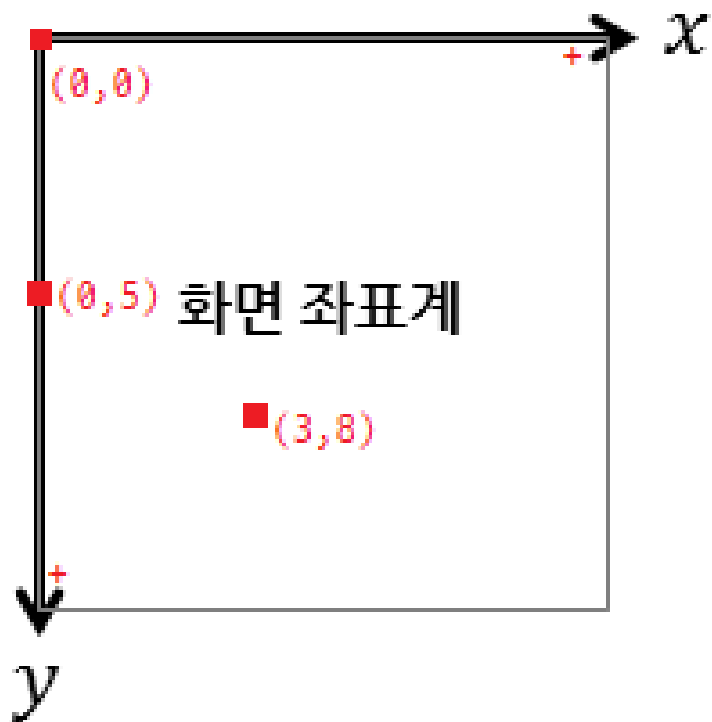


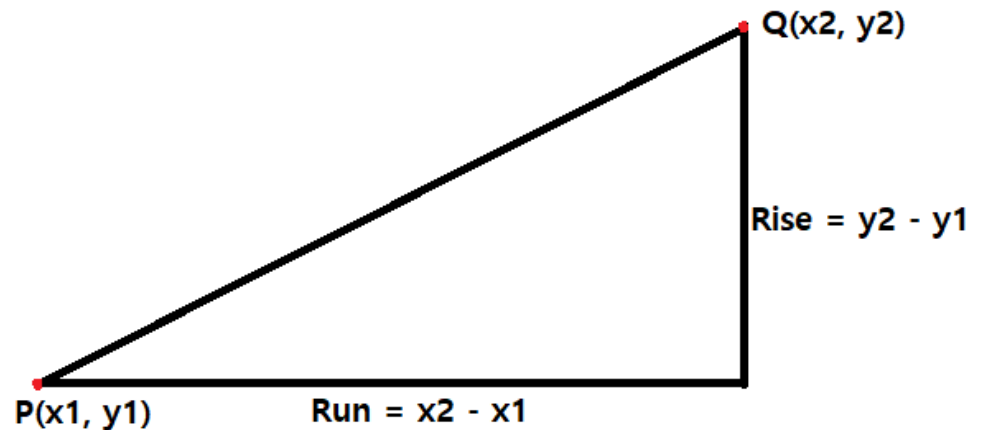
게임 수학

좌표계



직선

- ▶ 방정식 : $ax + by + c = 0$
- ▶ 직선의 가장 중요한 성질은 **기울기**, 혹은 경사도
- ▶ 수평으로 이동(Run)할 때마다 일정하게 상승(Rise)하는 경사면에서 두 점의 좌표 $P(x_1, y_1)$, $Q(x_2, y_2)$ 로 표현하면, 기울기 $m = (y_2 - y_1) / (x_2 - x_1)$ 이다
- ▶ **x절편** : $y = 0$ 일 때의 x 의 값
- ▶ **y절편** : $x = 0$ 일 때의 y 의 값



직선

```
class Point
{
private:
    float m_fX, m_fY;

public:
    Point(float x, float y) : m_fX(x), m_fY(y) {}

    Point operator+(Point& point)
    {
        return Point(this->m_fX + point.GetX(), this->m_fY + point.GetY());
    }

    float GetX() const { return m_fX; }
    float GetY() const { return m_fY; }
};

//두 점을 잇는 직선의 기울기.
float GetSlope(const Point& p1, const Point& p2)
{
    return (p2.GetY() - p1.GetY()) / (p2.GetX() - p1.GetX());
}
```

위 코드와 같이 구현이 가능하다

직선

- ▶ 두 직선이 직각을 이룰 때, 즉 **두 직선이 수직으로 만날 때**를 두 선이 “**직교한다**”라고 한다
- ▶ $m_1 m_2 = -1$ or $m_1 = -1/m_2$ or $m_2 = -1/m_1$

// 이 기울기와 직교하는 직선의 기울기.

// perpendicular

```
float PerpSlope(float slope)
```

```
{
```

```
    return -1 / slope;
```

```
}
```

// 두 직선은 직교인가?

```
bool IsPerp(float slope1, float slope2)
```

```
{
```

```
    return (-1 == slope1 * slope2);
```

```
}
```

직선

▶ 두 직선의 교차점

- $X = (m_1x_1 - m_2x_2 + y_2 - y_1) / (m_1 - m_2)$
- $Y = m_1(x - x_1) + y_1$
or $Y = m_2(x - x_2) + y_2$

// 두 직선의 교차점.

```
Point lineIntersect(const Point& p1, const float& slope1, const Point& p2, const
    float& slope2)
{
    auto x = (slope1 * p1.GetX() - slope2 * p2.GetX() + p2.GetY() - p1.GetY()) /
        (slope1 - slope2);
    auto y = slope1 * (x - p1.GetX()) + p1.GetY();
    // or y = slope2 * (x - p2.GetX()) + p2.GetY();
    return Point(x, y);
}
```

충돌 검출에의 응용

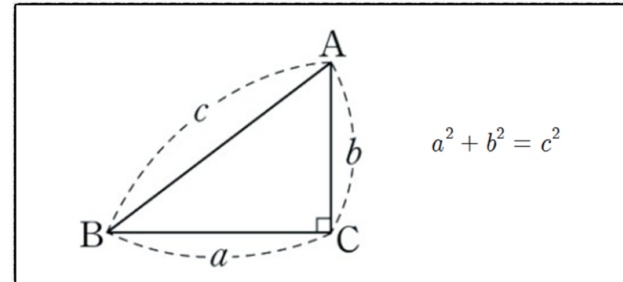
- ▶ 직선으로 건물의 벽이나 물체의 이동경로를 나타낼 수도 있는데, 이러한 두 직선이 만나는지에 따라 프로그램 작성을 할 경우
- ▶ **두 직선의 기울기 m_1, m_2**
 - $m_1 \neq m_2$ 이면, **교차점이 한 개 존재**한다
 - $m_1 = m_2$ 이면, 두 직선의 y 절편 b_1, b_2 를 구한다
 - $b_1 \neq b_2$ 이면, 두 선은 **평행**,
 - $b_1 = b_2$ 이면, 두 선은 **겹쳐있다**

피타고라스의 정리

- ▶ 화면상의 두 점 사이의 거리를 구하기 위하여 사용
 - $B(x_1, y_1)$, $A(x_2, y_2)$ 사이의 거리
 - $BA = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

// 두 점사이의 거리.

```
float Distance2D(const Point& p1, const Point& p2)
{
    return ((float)sqrt(pow(p2.GetX() - p1.GetX(), 2) + pow(p2.GetY()
- p1.GetY(), 2)));
}
```



중점

- ▶ 화면상의 두 점 사이의 절반이 되는 정확한 위치를 구하기 위하여 사용

// 두 점 사이의 중점.

```
Point MidPoint2D(const Point& p1, const Point& p2)
{
    auto x = (p1.GetX() + p2.GetX()) * 0.5f;
    auto y = (p1.GetY() + p2.GetY()) * 0.5f;

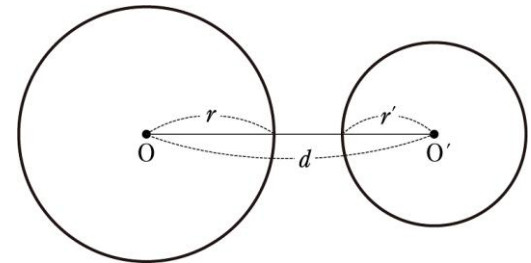
    return Point(x, y);
}
```

포물선

- ▶ 물체의 시작 위치와 속도를 알 때, 이 물체가 수평 이동을 하면서 얼마나 높이 올라 갔는지 알기 위하여 사용
- ▶ 수직축의 포물선
 - 정점(h, k)를 가지는 포물선의 방정식
 - $y = a(x - h)^2 + k$
 - $a > 0$ 이면 포물선은 아래로 $a < 0$ 이면 포물선은 위로 향한다
- ▶ 수평축의 포물선
 - 정점(h, k)를 가지는 포물선의 방정식
 - $x = a(y - k)^2 + h$
 - $a > 0$ 이면 포물선이 왼쪽으로 $a < 0$ 포물선이 오른쪽으로 향한다

원

- ▶ 사각형을 이용한 충돌처리보다 좀 더 명확한 처리를 하기 위하여 주로 사용한다
- $x^2 + y^2 = r^2$



▶ 두 원의 충돌

- 두 원 사이의 거리(d)가 두 원의 반지름의 합($r + r'$)보다 작거나 같다면 두 원은 충돌 상태이다

```
bool CircleCollision(const Circle& c1, const Circle& c2)
{
    return (Distance2D(c1.center, c2.center) <= (c1.radius + c2.radius));
}
```

도와 라디안

▶ 도(Degree)

- 원 한 바퀴를 360으로 표현하는 방법

▶ 라디안(Radian)

- 부채꼴의 중심각, **호의 길이**가 **원의 반지름**과 같을 때를 **1라디안**이라 정의 한다

```
#define Rad2Deg 57.29577951f
```

```
#define Deg2Rad 0.017453293f
```

- 라디안에서 도, 도에서 라디안을 구할 때 해당 값을 곱하여 알아올 수 있다

삼각함수

- \sin (사인), \cos (코사인), \tan (탄젠트)
- 삼각함수는 벡터, 물리학을 다루는 방법으로 자주 사용 된다
- \sin , \cos 은 흔히 진동 현상에서 사용된다

도	라디안	\sin	\cos	\tan
0	0	0	1	0
30	$\pi/6$	0.5	0.8660	0.5774
45	$\pi/4$	0.7071	0.7071	1
60	$\pi/3$	0.8660	0.5	1.7321
90	$\pi/2$	1	0	-
120	$2\pi/3$	0.8660	-0.5	-1.7321
180	π	0	-1	0
270	$3\pi/2$	-1	0	-
360	2π	0	1	0

탄젠트의 역함수

- atan(아크탄젠트)

▶ 물체와 목표의 위치를 알고 물체와 목표의 각도를 구하여야 할 경우

// 두 점 사이의 각도(Degree)

```
float CalculatorAngle2D(const Point& p1, const Point& p2)
{
    float angle = (float)atan((p2.GetY() - p1.GetY()) / (p2.GetX() - p1.GetX())) * Rad2Deg;

    if (p2.GetY() < p1.GetY() && p2.GetX() > p1.GetX())
        return angle;
    else if ((p2.GetY() < p1.GetY() && p2.GetX() < p1.GetX()) || (p2.GetY() > p1.GetY() &&
p2.GetX() < p1.GetX()))
        return angle + 180;
    else
        return angle + 360;
}
```

단위 원

- 원의 반지름을 1로 하여 x , y 의 값은 $-1 \sim 1$ 의 값을 가진다
- 흔히 물체가 나아갈 방향을 구하는데 도움이 된다

▶ 데카르트 좌표 기준

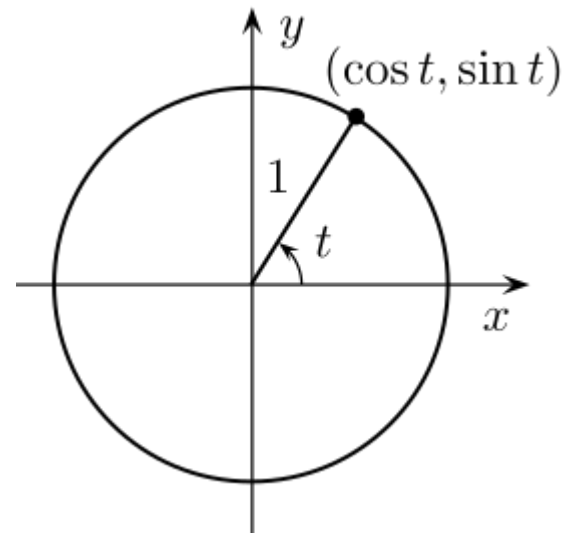
$$x = \cos(\text{radian})$$

$$y = \sin(\text{radian})$$

▶ 화면 좌표 기준

$$x = \cos(\text{radian})$$

$$y = -\sin(\text{radian})$$

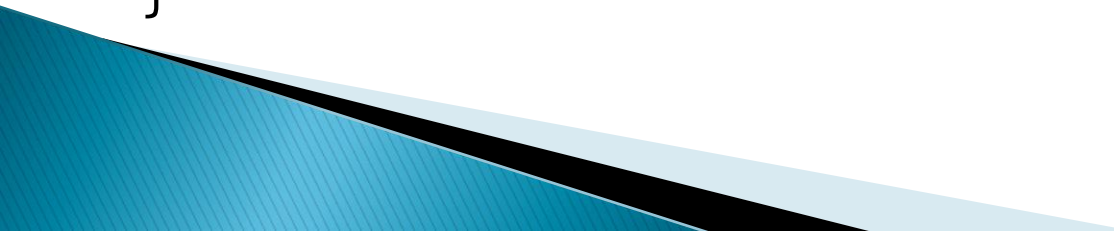


단위 원

```
Point Direction2DWithRad(const float& radian)
{
    auto x = cos(radian);
    auto y = -sin(radian);

    return Point(x, y);
}
```

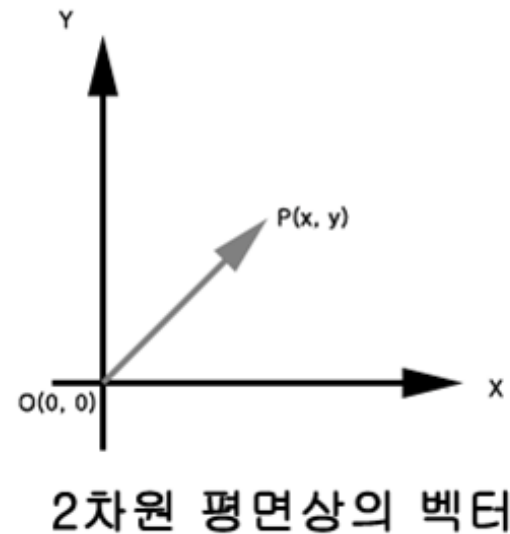
```
Point Direction2DWithDeg(const float& degree)
{
    return Direction2DWithRad(degree * Deg2Rad);
}
```



벡터

- ▶ 스칼라 : 크기만 가진다
- ▶ 벡터 : 크기 + 방향

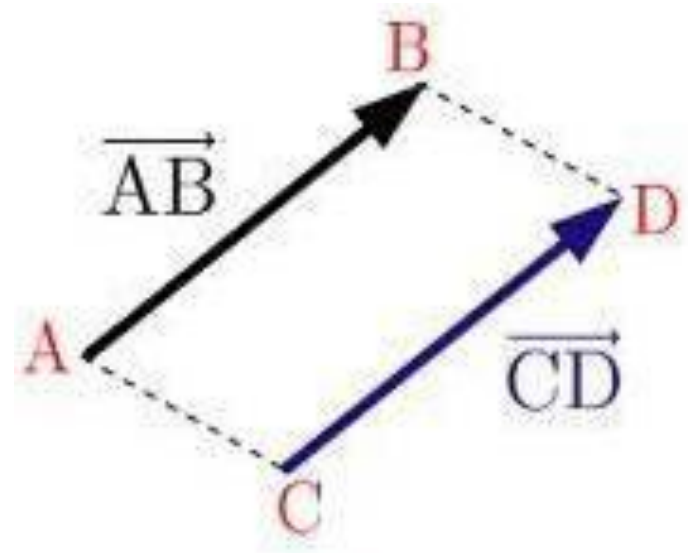
```
struct Vector2  
{  
    float x, y;  
};
```



벡터의 상등

- 벡터는 위치를 나타내는 개념이 아니다

“벡터AB와 벡터CD는 크기와 방향이 같으므로 서로 상등하다”



벡터의 더하기

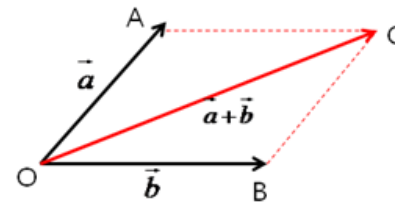
- 오브젝트(물체)의 이동, 물리 힘 관련 처리시 사용

```
class Vector2
{
public:
    float x, y;

    Vector2 operator+(const Vector2& v)
    {
        Vector2 temp;
        temp.x = this->x + v.x;
        temp.y = this->y + v.y;

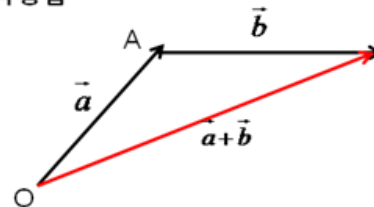
        return temp;
    }
};
```

평행사변형법



\vec{a} 와 \vec{b} 의 시점 일치시:
평행사변형의 대각선이 두 벡터의 합을 의미
 $|\vec{a}+\vec{b}|$: 두 벡터 합의 크기 $\rightarrow \overline{OC}$ 대각선 길이를 의미

삼각형법



\vec{a} 와 \vec{b} 의 종점을 연결시:
시점과 종점을 연결한 벡터가 두 벡터의 합을 의미
 $|\vec{a}+\vec{b}|$: 두 벡터 합의 크기 $\rightarrow \overline{OB}$ 길이를 의미

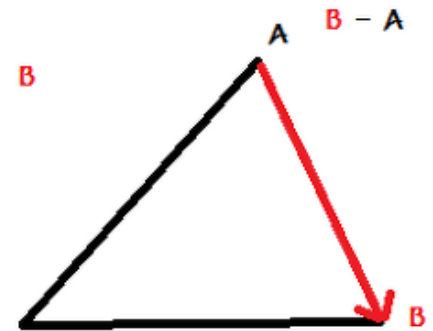
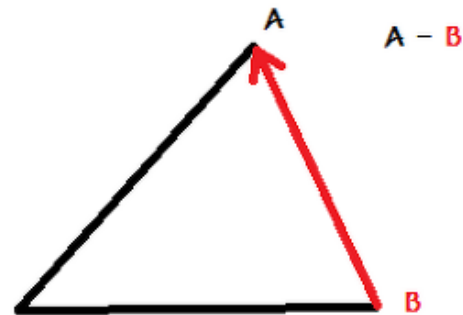
벡터의 빼기

- 내가 타겟을 향해 가기 위한 **방향을 구하기** 위하여 사용

```
class Vector2
{
public:
    float x, y;

    Vector2 operator-(const Vector2& v)
    {
        Vector2 temp;
        temp.x = this->x - v.x;
        temp.y = this->y - v.y;

        return temp;
    }
};
```



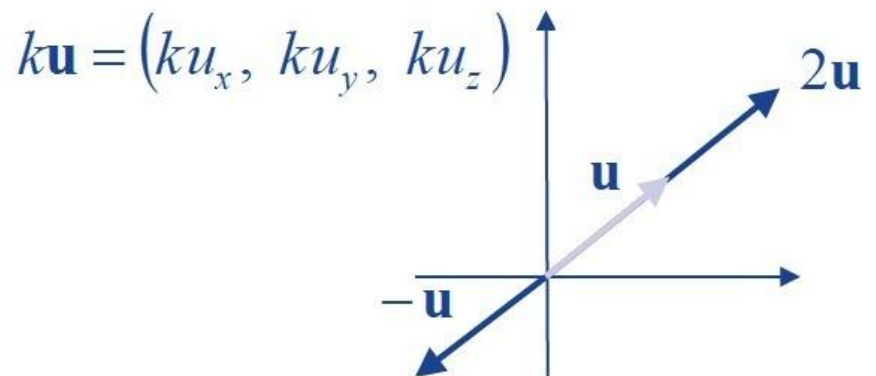
벡터와 스칼라의 곱

- 벡터는 가려는 방향, 스칼라는 속력

```
class Vector2
{
public:
    float x, y;

    Vector2 operator*(const float& value)
    {
        Vector2 temp;
        temp.x = this->x * value;
        temp.y = this->y * value;

        return temp;
    }
};
```

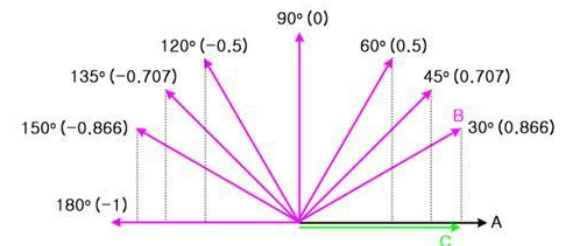
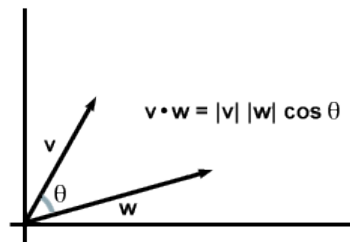


벡터의 내적

- **두 벡터 사이의 각도로 $-1 \sim 1$ 사이의 값이 나온다**
적의 시야에 내가 들어갔는지 등을 확인할 때 사용
 $v \cdot w = 0$ 두 벡터는 직각
 $v \cdot w > 0$ 두 벡터의 내각은 90보다 작다
 $v \cdot w < 0$ 두 벡터의 내각은 90보다 크다
- 빛의 방향 벡터, 물체의 수직 벡터를 이용하여 그림자의 길이를 구할 수 있다
(1.0f - |DotProduct(빛의 방향 벡터, 물체의 수직 벡터)|) * 물체의 높이 = 그림자의 길이

// 벡터의 내적.

```
float DotProduct(const Vector2& v1, const Vector2& v2)
{
    return v1.x * v2.x + v1.y * v2.y;
}
```



벡터의 정규화

- ▶ 값을 $-1 \sim 1$ 사이의 값으로 만든다
 - 단위 원과 같은 값을 가진다
 - 결과 값을 단위 벡터(normal vector)라한다

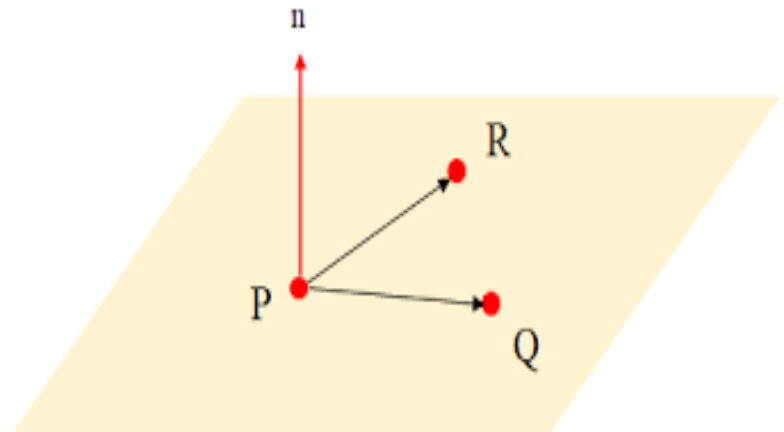
```
Vector2 normalize()  
{  
    float magnitude = sqrtf(pow(x, 2) + pow(y, 2));  
    return Vector2(x / magnitude, y / magnitude);  
}
```

벡터의 외적

▶ 두 벡터의 곱하기

- 3차원 벡터에서 있는 개념
- 두 벡터에 수직인 벡터를 구할 수 있으며, **단위 벡터(Normal Vector) 변환하여 사용**

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



극 좌표

- 크기와 각도를 지닌다
- vector와 같은 데카르트 좌표에 비해 더 직관적이다

// 극 좌표.

class Polar

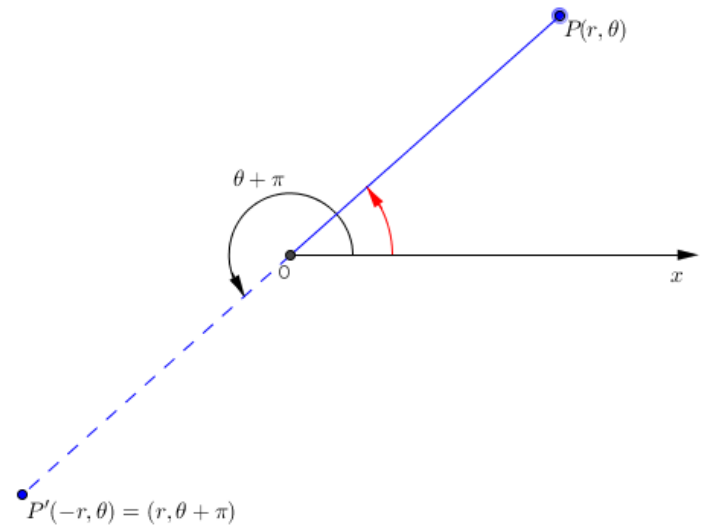
{

public:

float magnitude, angle;

Polar() : magnitude(0), angle(0) {}

};



데카르트 좌표와 극 좌표

```
Vector2 Polar2Comp(const Polar& p)
{
    Vector2 temp;
    temp.x = p.magnitude * cos(p.angle * Deg2Rad);
    temp.y = p.magnitude * -sin(p.angle * Deg2Rad);

    return temp;
}

Polar Comp2Polar(const Vector2& v)
{
    Polar temp;
    temp.magnitude = sqrtf(pow(v.x, 2) + pow(v.y, 2));
    if (0 == temp.magnitude) return temp;
    temp.angle = Rad2Deg * (float)asin(v.y / temp.magnitude);
    if (0 > v.x) temp.angle += 180;
    else if (0 < v.x && 0 > v.y) temp.angle += 360;
    return temp;
}
```

일차원 운동

▶ 등속운동

- 거리 = 속도 * 시간

// 단위 시간당 속도에 의한 이동 거리.

```
float Displacement(const float& velocity, const float& time)
{
    return velocity * time;
}
```

▶ 평균 속도 계산

// 시작점과 도착점, 이동한 시간의 평균 속도.

```
float AvgVelocity(const float& start, const float& end, const float& intervalTime)
{
    return (end - start) / intervalTime;
}
```

▶ 가속도 계산

// 단위 시간당 가속도를 구한다.

// 시작 속도, 현재 속도, 시작 부터 현재까지 걸린 시간.

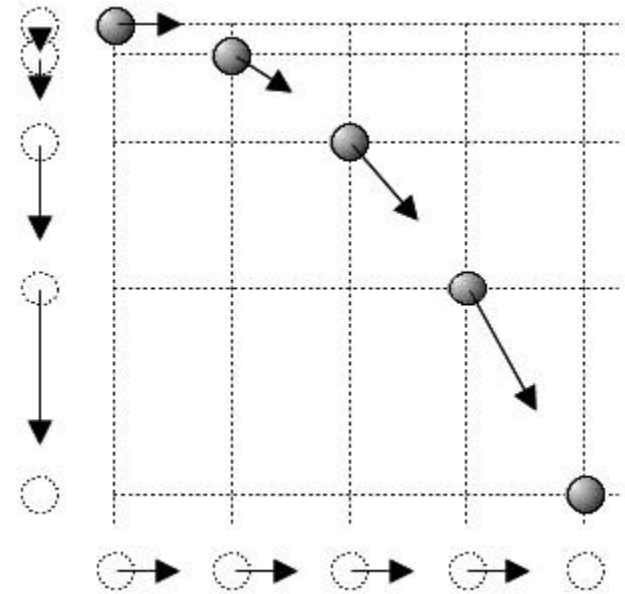
```
float AccelerationSeconds(const float& startVel, const float& endVel, const float& intervalTime)
{
    return (endVel - startVel) / intervalTime;
}
```

이차원 운동

```
class GameObject
{
private:
    Vector2 pos;
    Vector2 velocity;

public:
    void UpdatePos()
    {
        pos += velocity;
    }

    // 수직 방향 가속도 업데이트.
    // 데카르트 좌표에 따라 가속도가 양수면 아래로 떨어진다.
    void UpdateVelocity(const float& accel, const float& deltaTime)
    {
        velocity.y = velocity.y + accel * deltaTime;
    }
};
```



이차원 운동

- ▶ 중력과 질량에 의한 무게 계산

```
#define GRAVITY 9.81f
Vector2 Weight(const float& mass)
{
    Vector2 weight;

    weight.y = mass * GRAVITY;

    return weight;
}
```

마찰력과 경사면

// 물체가 경사면을 미끄러지는지 체크.

// 미끄러지면 true, 아니면 false

```
bool IsSlopeSlip(const float& angle, const float& weight, const float& coeffFric)
{
    float normal = weight * cosf(angle * Deg2Rad);
    float perpForce = weight * sinf(angle * Deg2Rad);
    float staticFriction = coeffFric * normal;
    return (perpForce > staticFriction);
}
```

// 경사면에서 미끄러져 내려가는 가속도 계산.

```
float SlopeSlipAccel(const float& angle, const float& weight, const float& coeffFric,
    const float& mass)
{
    float normal = weight * cosf(angle * Deg2Rad);
    float perpForce = weight * sinf(angle * Deg2Rad);
    float kineticFriction = coeffFric * normal;
    float totalForce = perpForce - kineticFriction;
    return (totalForce / mass);
}
```

일

- ▶ 물체에 힘을 작용해 힘의 방향으로 물체를 이동시키는 것

// 일의 양.

```
float Work(const float& force, const float& friction, const float& displacement)
{
    float netForce = force - friction;
    return displacement * netForce;
}
```

// 각을 고려하여 힘을 계산.

```
float AngleWork(const Polar& p, const float& friction, const float& displacement)
{
    float temp = cos(p.angle * Deg2Rad);
    float horizForce = p.magnitude * temp;

    return Work(horizForce, friction, displacement);
}
```

일

```
// initial: 물체에 작용된 힘, chage: 외부의 요인으로 인하여 실제 작용한 힘.
float AngleWorkFromVector(const Polar& initial, const Polar& chage)
{
    // 극 좌표를 데카르트 좌표로 변경.
    auto force = Polar2Comp(initial);
    auto displacement = Polar2Comp(chage);

    // 변위의 정규화.
    auto displacementNormal = displacement.normalize();

    // 투영된 길이. 변위의 단위벡터와 힘의 내적.
    auto projectionLenth = DotProduct(force, displacementNormal);

    // 투영된 길이와 변위의 단위벡터의 곱.
    auto temp = displacementNormal * projectionLenth;
    // 일을 계산하기 위한 극좌표 형식으로 다시 변환.
    auto polarTemp = Comp2Polar(temp);

    return polarTemp.magnitude * chage.magnitude;
}
```


일-에너지 정리

▶ 운동에너지

- 질량 m , 속력 v
- $KE = 1/2mv^2$
- 운동 에너지는 속도가 아닌 속력을 사용
- 운동 에너지는 스칼라이기에 방향을 신경 쓰지 않는다

// 질량과 속력을 받아 운동 에너지 계산.

```
float KineticEnergy(const float& mass, const float& speed)
{
    return (mass * 0.5f) * (pow(speed, 2));
}
```

// 일-에너지 정리.

// 물체의 최종 속도를 계산.

```
float WorkEnergy(const float& force, const float& mass, const float& displacement, const float& velocityInitial)
{
    // 마찰력이 존재하지 않는 상황을 가정.
    float work = Work(force, 0, displacement);
    float result = work / (mass * 0.5f) - KineticEnergy(mass, velocityInitial);
    return sqrtf(result);
}
```

중력 포텐셜 에너지

▶ Gravitational Potential Energy(GPE)

- 물체를 들고 있다가 놓게 되면 떨어지는데, 이 때 물체가 잡혀있는 동안 정장되어 있는 에너지
- 질량 m , 중력가속도 g , 높이 y
- $GPE = mgy$

// 포텐셜 에너지를 줄 단위로 계산.

```
float PotentialEnergy(const float& mass, const float& height)
{
    return mass * GRAVITY * height;
}
```

물체의 충돌과 반사

- ▶ 축에 정렬되지 않은 반사, 정지한 물체의 경계가 x축이나 y축에 평행하지 않을 경우의 반사 처리

// 축이 정렬된 상태의 반사 방향.

```
Vector2 AxisAlignedCollision(const Vector2& v, bool isHorizen)
{
    Vector2 temp = v;
    if (!isHorizen) temp.x *= -1;
    else temp.y *= -1;

    return temp;
}
```

// 축이 비정렬 상태의 반사.

```
Vector2 NonAxisAlignedCollision(const Vector2& v1, const Vector2& v2)
{
    auto normal = Normalize(v1);
    auto revalsal = ScalarMultiply(v2, -1);
    auto projection = DotProduct(revalsal, normal);
    auto length = ScalarMultiply(normal, projection);
    auto reflection = ScalarMultiply(length, 2);

    return SumVector(reflection, revalsal);
}
```