게임 수학

남동강

목차

1. 좌표계 ・・・ 4p
2. 직선 ・・・ 5p
3. 원 ・・・ 9p
4. 삼각함수 ・・・ 10p
5. 벡터 ・・・ 12p

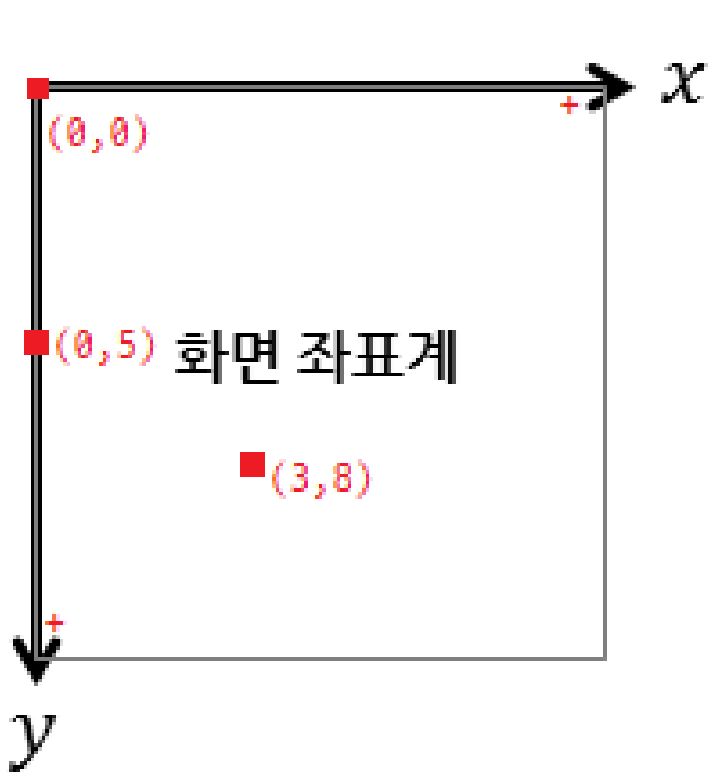
목차

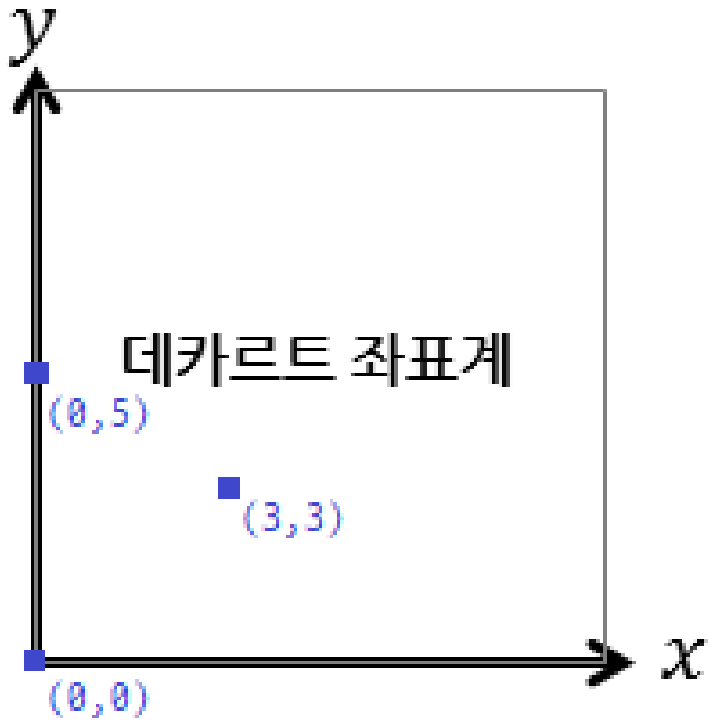
6. 극 좌표 ・・・ 17p

7. Physics Engine ・・・ 18p

8. 연습문제 ・・・ 26p

좌표계





직선

직선의 방정식

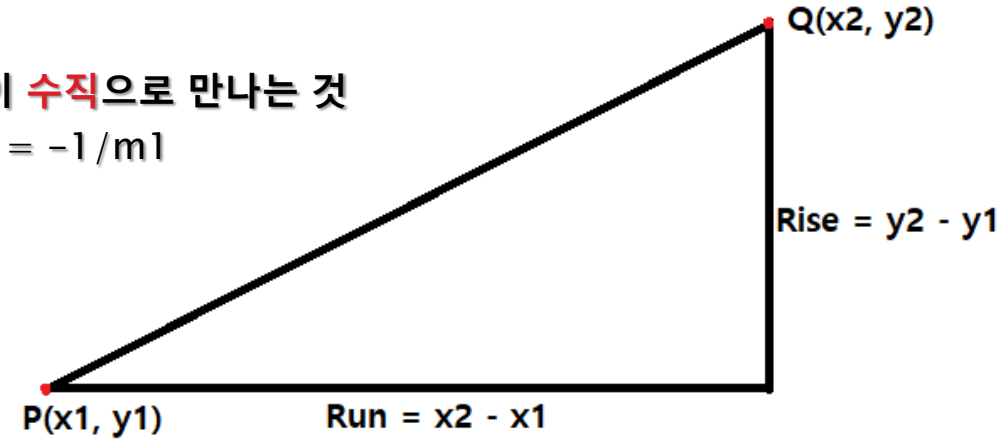
- ax + by + c = 0
- 직선의 가장 중요한 성질은 **기울기**(경사도)

기울기

- 수평으로 이동(Run)할 때마다 일정하게 상승(Rise)하는 경사면에서 두 점의 좌표 P(x1, y1), Q(x2, y2)로 표현
- m(기울기) = (y2 – y1) / (x2 – x1)

직교

- 두 직선이 직각을 이룰 때, 즉 **두 직선이 수직으로 만나는 것**
- m1m2 = –1 or m1 = –1/m2 or m2 = –1/m1



직선

충돌 검출에 응용

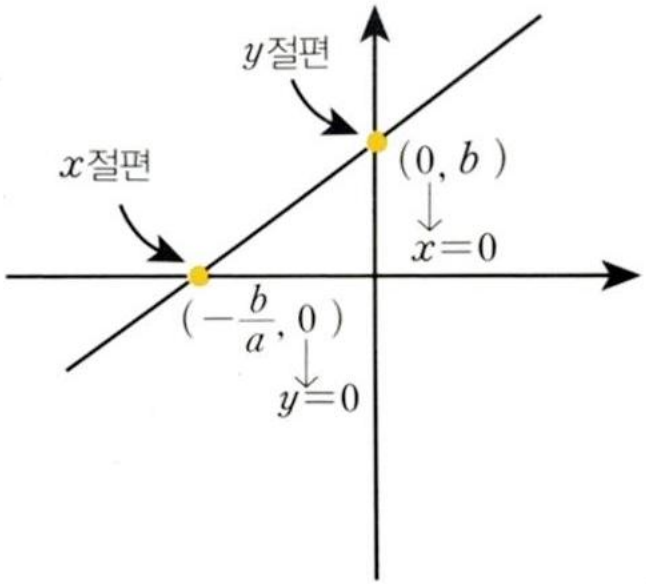
- 직선으로 **건물의 벽**이나 **물체의 이동경로**를 나타낼 수도 있는데, 이러한 두 직선의 교차점으로 충돌 확인 가능

두 직선의 교차점

- 기울기 **m1**을 지니는 직선상의 점 **P(x1, y1)**
  기울기 **m2**를 지니는 직선상의 점 **P(x2, y2)**
- X = (m1x1 – m2x2 + y2 – y1) / (m1 – m2)
- Y = m1(X – x1) + y1 or Y = m2(X – x2) + y2

두 직선의 기울기 m1, m2

- **m1 != m2** : **교차점이 한 개 존재**한다
- m1 == m2일 경우, 두 직선의 y절편 b1, b2를 구한다
  b1 != b2 : **두 선은 평행**
  b1 == b2 : **두 선은 겹쳐있다**



- **x절편** : y = 0일 때의 x의 값
- **Y절편** : x = 0일 때의 y의 값

직선

- **기울기를 이용한 두 직선의 교차점 구하기의 문제점**
  - 2개의 점을 시작과 끝을 이루는 특정한 범위를 가지는 두 직선 사이의 교차점 만을 구하는 것에 매우 비효율적이며 구하기 힘들다

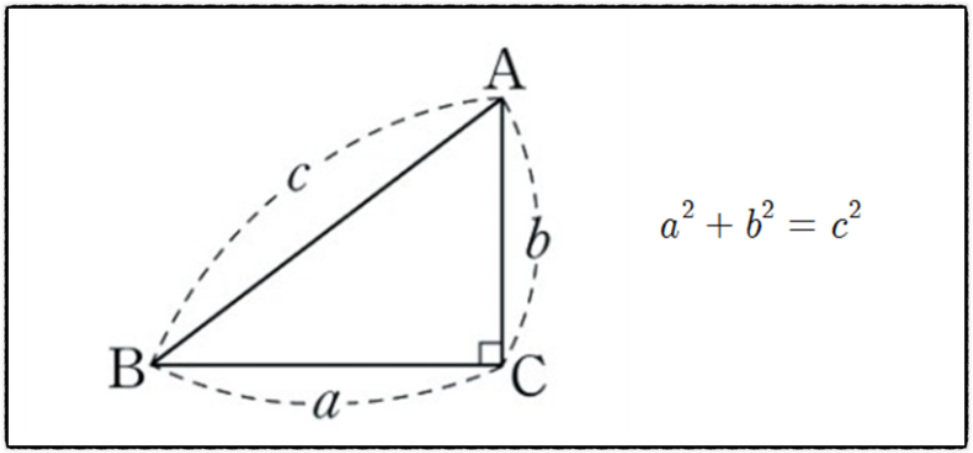
- **범위를 가지는 두 직선의 교차점**
  - 하나의 직선의 시작 점 **P(x1, y1)**와 끝 점 **P(x2, y2)**
    또 다른 직선의 시작 점 **P(x3, y3)**와 끝 점 **P(x4, y4)**

```
{
    float under = (y4 - y3) * (x2 - x1) - (x4 - x3) * (y2 - y1);

    if (0 == under) return;

    float t = (x4 - x3) * (y1 - y3) - (y4 - y3) * (x1 - x3);
    float s = (x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3);

    if (0 == t && 0 == s) return;

    t = t / under;
    s = s / under;

    if (0 > t || 1 < t || 0 > s || 1 < s) return;

    // 교차점.
    float crossPointX = x1 + t * (x2 - x1);
    float crossPointY = y1 + t * (y2 - y1);
}
```

직선

- **피타고라스의 정리**
  - 화면상의 **두 점 사이의 거리를 구한다**

- **B(x1, y1), A(x2, y2) 사이의 거리**
  - BA = sqrt((x2 – x1)^2 + (y2 – y1)^2)



원

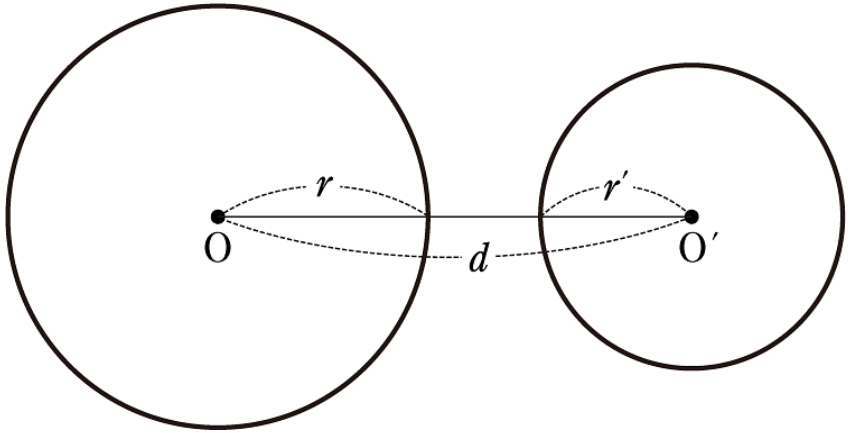
- 사각형을 이용한 충돌처리보다 **좀 더 명확한 충돌 처리**를 하기 위하여 주로 사용

방정식

- 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



삼각함수

- 삼각함수는 벡터, 물리학을 다루는 방법으로 자주 사용

도(Degree)

- 원 한 바퀴를 360으로 표현하는 방법

라디안(Radian)

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

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

삼각함수

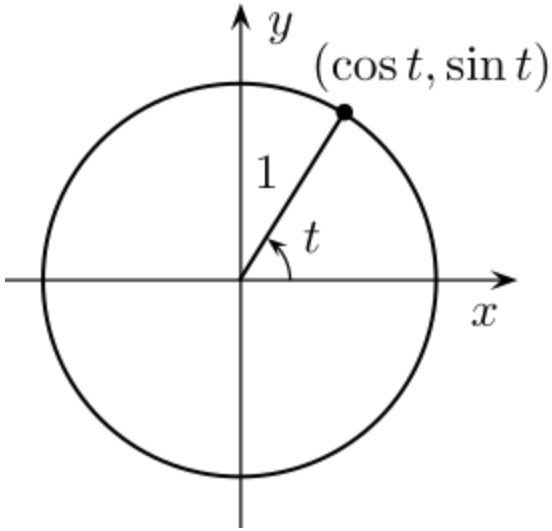
단위 원

- **원의 반지름**을 **1**로 하여 x, y의 값은 -1 ~ 1의 값을 가진다
- 물체가 나아갈 **방향**을 구하는데 사용

데카르트 좌표 기준

x = cos(radian)

y = sin(radian)



화면 좌표 기준

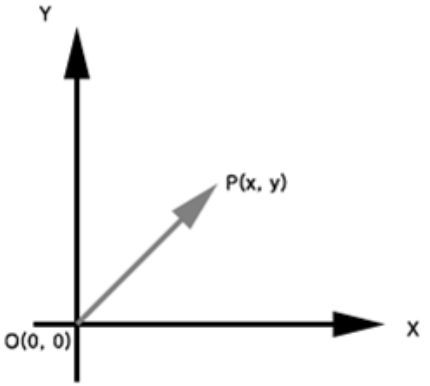
x = cos(radian)

y = -sin(radian)

```
#define Rad2Deg 57.29577951f
#define Deg2Rad 0.017453293f
```

- 라디안에서 도, 도에서 라디안 값으로 **변환**, 해당 값을 곱하여 알 수 있다

벡터



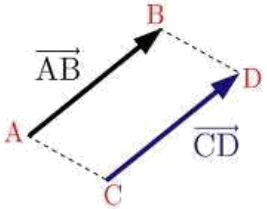
2차원 평면상의 벡터

벡터와 스칼라

- **벡터** : **크기** + **방향**
- **스칼라** : **크기**만 가진다

벡터의 상등

- 벡터는 위치를 나타내는 개념이 아니다

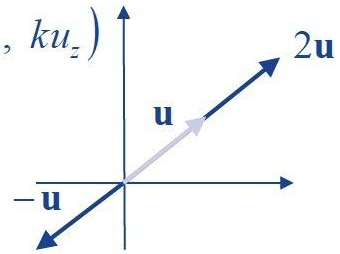


"벡터AB와 벡터CD는 **크기와 방향이 같으므로 서로 상등**하다"

벡터와 스칼라의 곱

- **벡터**는 가려는 **방향**, **스칼라**는 **속력**

$$k\mathbf{u} = (ku_x,\ ku_y,\ ku_z)$$



벡터

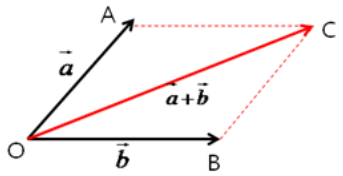
- **벡터의 정규화**
  - **단위 원** 값으로 변환
  - 결과 값을 **단위 벡터(normal vector)**라 한다

```
const float epsilon = 0.000001f; // 부동소수점.

float x, y;
float sqrMagnitude = (x * x) + (y * y);
float magnitude = std::sqrt(sqrMagnitude);
if (epsilon < magnitude) // if(0 < magnitude)
{
    float invMagnitude = 1.0f / magnitude;
    x *= invMagnitude;
    y *= invMagnitude;
}
```

벡터

- **벡터의 더하기**
  - **오브젝트(물체)의 이동**, 물리 힘 관련 처리시 사용
- **벡터의 빼기**
  - 내가 타겟을 향해 가기 위한 **방향**을 **구하기** 위하여 사용

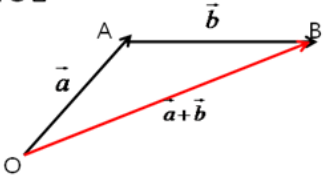
평행사변형법



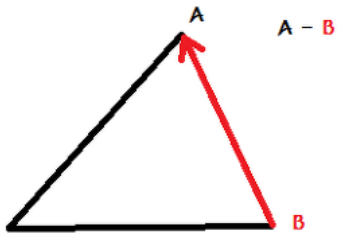
$\vec{a}$와 $\vec{b}$의 시점 일치시:
평행사변의 대각선이두 벡터의 합을 의미
$|\vec{a}+\vec{b}|$: 두벡터 합의크기 $\rightarrow \overline{OC}$대각선길이를의미

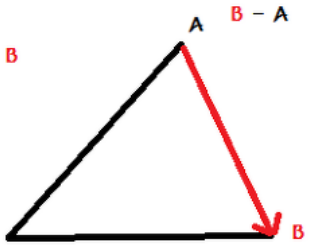
벡터의 더하기

삼각형법



$\vec{a}$와 $\vec{b}$의 종점을 연결시:
시점과 종점을 연결한 벡터가 두 벡터의 합을 의미
$|\vec{a}+\vec{b}|$: 두벡터 합의크기 $\rightarrow \overline{OB}$ 길이를의미



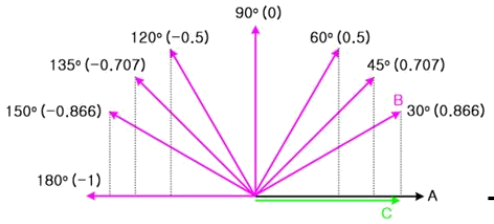


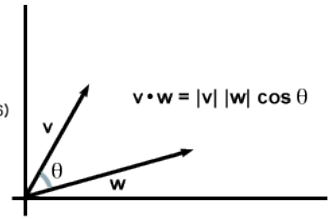
벡터의 빼기

벡터

벡터의 내적

- **두 벡터 사이의 각도**로 **cosθ** 값이 나온다
  - v·w = 0 두 벡터는 직각
  - v·w > 0 두 벡터의 내각은 90보다 작다
  - v·w < 0 두 벡터의 내각은 90보다 크다
- 시야에 대상이 있는지 확인
- 투영 벡터를 구하여 그림자를 그릴 수 있다

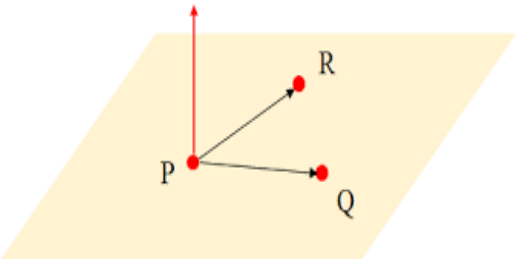




벡터의 내적에서 각도(Degree) 구하기

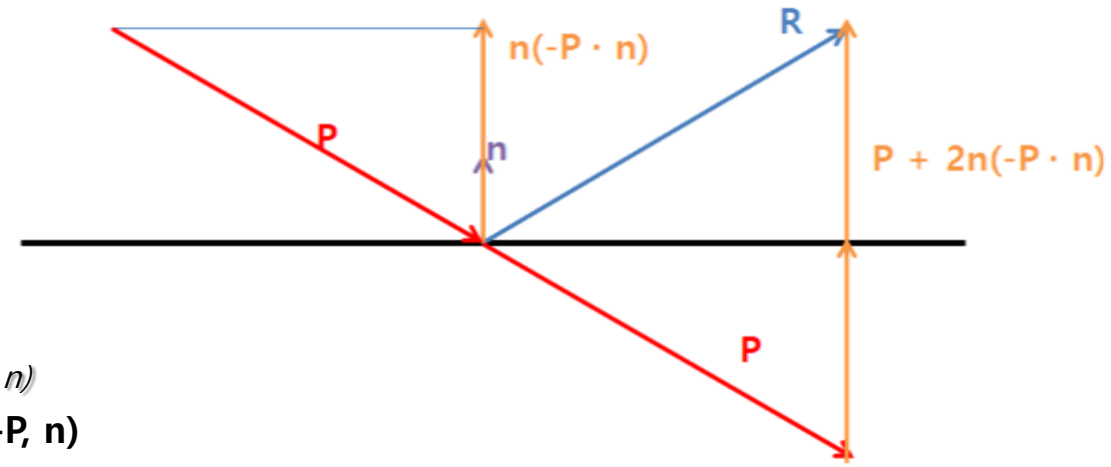
- angle = acos(dot) * Rad2Deg
- **acos** : 아크 코사인, cosθ 값에서 라디안(θ) 값으로 역으로 구한다

벡터의 외적



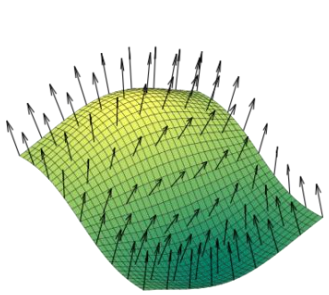
- 두 벡터의 곱
- 3차원 벡터에서 있는 개념
- **두 벡터에 수직인 벡터**를 구할 수 있으며, **단위 벡터(Normal Vector) 변환하여 사용**

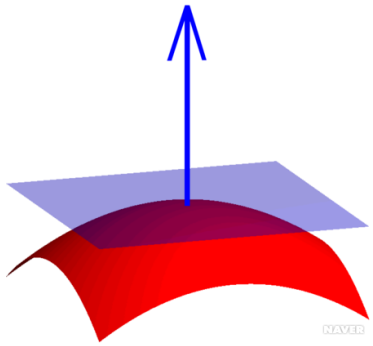
벡터



- **반사 벡터**
  - 입사 벡터 : P
  - 법선 벡터 : n
  - 반사 벡터 : R
  - 투영 벡터 : *n * DotProduct(-P, n)*
  - **R = P + 2n * DotProduct(-P, n)**

- **빛의 반사 벡터**를 구하기 위해서 **반드시 법선 벡터가 필요**하다
- 법선 벡터(normal vector) : 한 면에 수직이 되는 벡터
  벡터의 외적을 이용하여 구할 수 있다

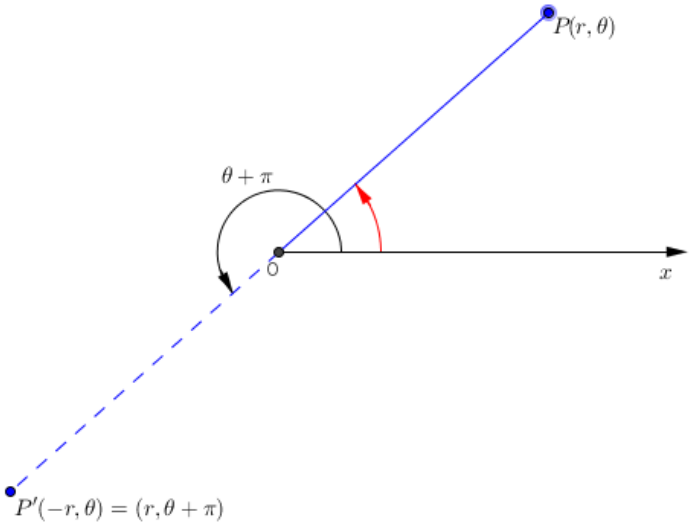




극 좌표

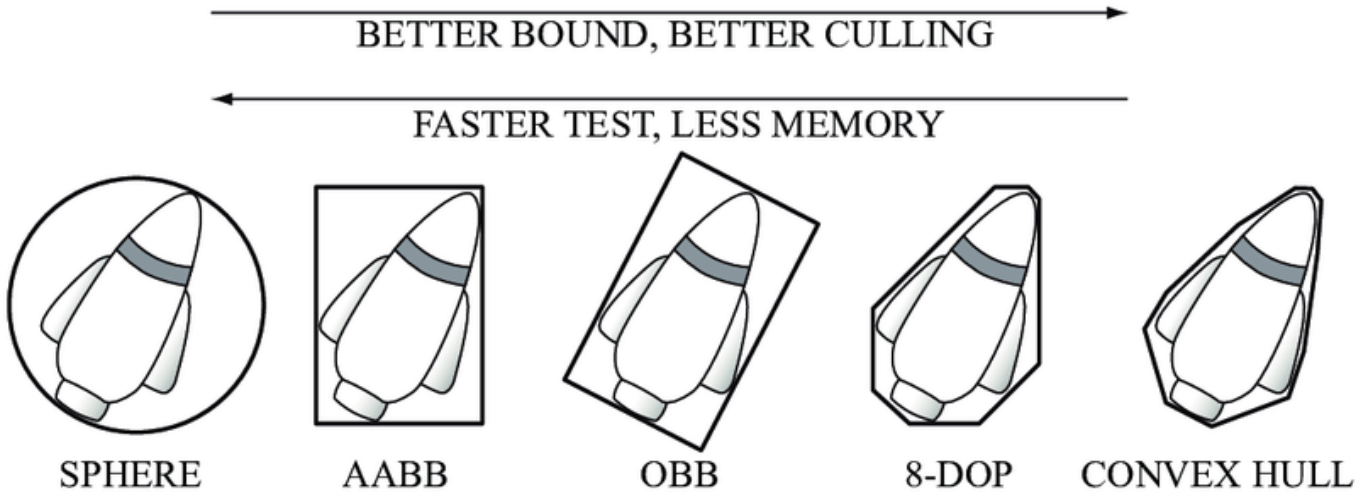
- **Polar(극 좌표)**
- **크기**(magnitude)와 **각도**(angle)를 지닌다
- vector와 같은 **데카르트 좌표에 비해 더 직관적**이다

- 투영 벡터를 이용하여 그림자의 크기와 방향을 구하고 Polar의 magnitude를 이용하여 그림자의 길이를 구할 수도 있다



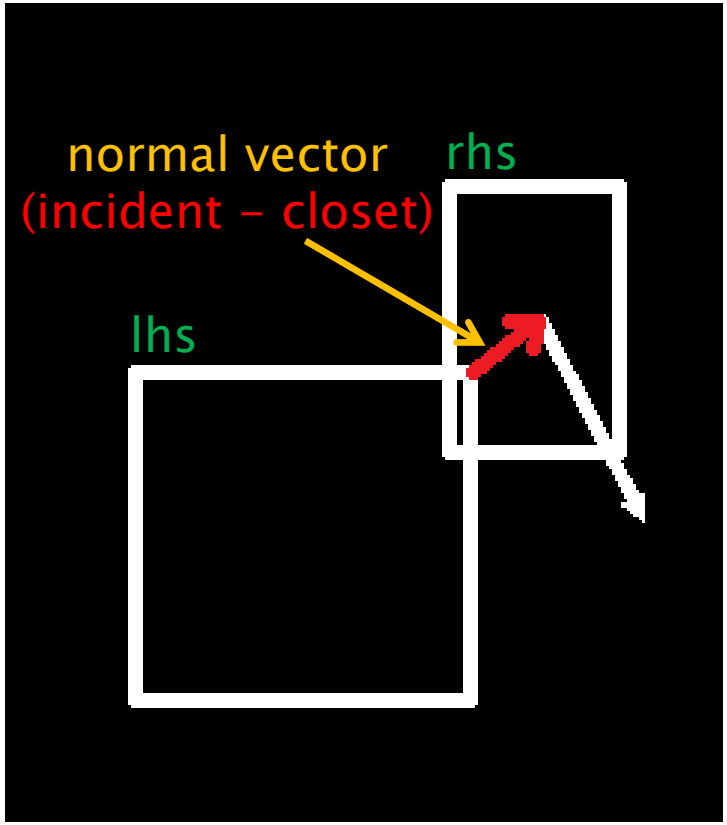
Physics Engine

- **Collider**
  - **AABB**(Axis Aligned Bounding Box) : 축이 정렬되어 있어 회전을 고려하지 않는다
  - **OBB**(Oriented Bounding Box) : 박스와 함께 축이 회전한다
  - **8-DOP, CONVEX HULL** : 여러 개의 정점을 지니며 OBB처럼 축이 회전한다



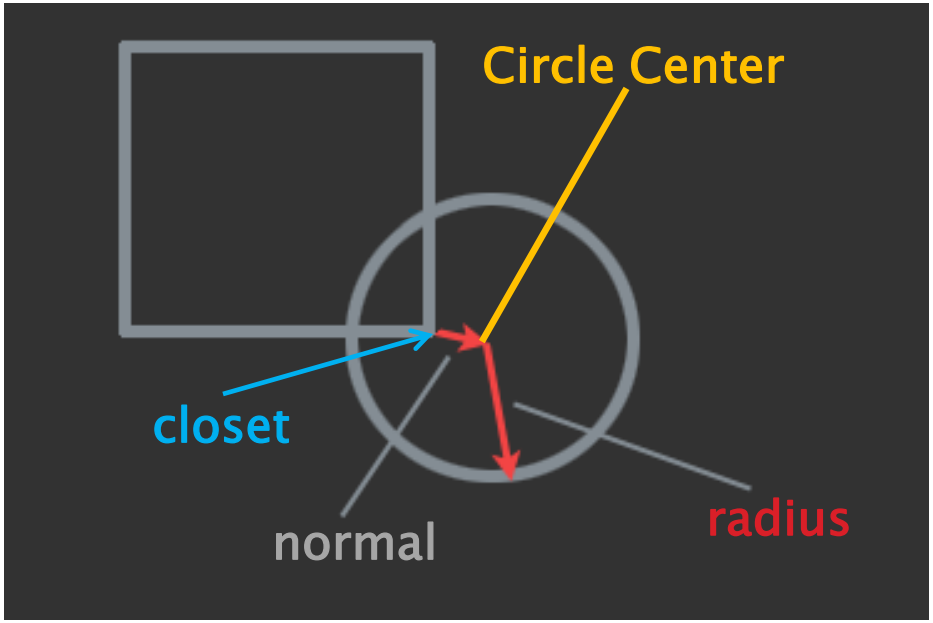
Physics Engine

```
Vector2 normal;    // 충돌 방향 normal 벡터.

bool AABBvsAABB(Box* lhs, Box* rhs)
{
    // 충돌 방향 벡터, 상대 위치(lhs를 원점으로 rhs의 위치)
    const Vector2 incident = rhs->position - lhs->position;
    // 각 Box의 너비(width)의 반(half width)을 구한다.
    float lhs_extent = lhs->halfWidth;
    float rhs_extent = rhs->halfWidth;
    float overlapX = lhs_extent + rhs_extent - std::abs(incident.x);
    if (0 < overlapX)// x축으로 겹치는 위치에 있다.
    {
        Vector2 closet;
        float offset = max(lhs_extent, rhs_extent);
        closet.x = clamp(incident.x, -offset, offset);
        // 각 Box의 높이(height)의 반(half height)을 구한다.
        lhs_extent = lhs->halfHeight;
        rhs_extent = rhs->halfHeight;
        float overlapY = lhs_extent + rhs_extent - std::abs(incident.y);
        if (0 < overlapY)// y축으로 겹치는 위치에 있다.
        {
            closet.y = clamp(incident.y, -lhs_extent, lhs_extent);
            normal = incident - closet;
            normal.Normalize();

            return true;
        }
    }
    return false;
}
```



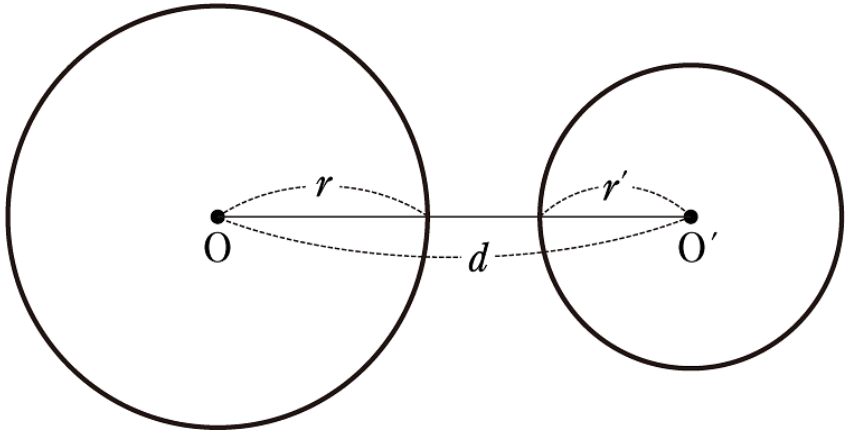
Physics Engine

```
bool AABBvsCircle(Box* lhs, Circle* rhs)
{
    const Vector2 incident = rhs->position - lhs->position; // Box를 중심으로 하는 Circle의 상대 위치.
    // Box의 너비(width)와 높이(height)의 반(half width, half height)을 구한다.
    float extentX = lhs->halfWidth;
    float extentY = lhs->halfHeight;

    Vector2 closet;
    closet.x = clamp(incident.x, -extentX, extentX);
    closet.y = clamp(incident.y, -extentY, extentY);

    bool inside = false;
    if (incident == closet) // Box와 Circle이 반 이상 침투했다.
    {
        inside = true;
        if (std::abs(incident.x) > std::abs(incident.y))
        {
            closet.x = (0 < closet.x) ? extentX : -extentX;
        }
        else closet.y = (0 < closet.y) ? extentY : -extentY;
    }

    normal = incident - closet;
    float dist = normal.SqrMagnitude();
    float radius = rhs->radius;
    if ((radius * radius) < dist && !inside) return false;

    if (inside) normal = -normal; // 반 이상으로 침투하면 침투 방향이 뒤집히기 때문에 다시 방향을 되돌려 준다.
    normal.Normalize();

    return true;
}
```



Physics Engine

```
bool CirclevsCircle(Circle* lhs, Circle* rhs)
{
    // lhs를 중심으로 하는 rhs의 상대 위치.
    const Vector2 incident = rhs->position - lhs->position;
    float radiusSum = lhs->radius + rhs->radius;

    // 두 원의 반지름의 합보다 두원 사이의 거리가 크면 충돌하지 않은 상태.
    if (incident.SqrMagnitude() > (radiusSum * radiusSum)) return false;

    float dist = incident.Magnitude();
    if (0 != dist)
    {
        normal = incident;
    }
    else
    {
        // 두 원은 같은 위치에 있다, 겹쳐있다.
        normal = Vector2(1, 0);
    }
    normal.Normalize();

    return true;
}
```



Physics Engine

- **이차원 벡터의 내적**

```
float Dot(const Vector2& lhs, const Vector2& rhs) { return lhs.x * rhs.x + lhs.y * rhs.y; }
```

- **부피**

- 사각형 : 부피(volume) = 너비(w) * 높이(h)
- 원 : 부피(volume) = 반지름(r) * 반지름(r) * 원주율(PI)

```
float Volume(float w, float h) { return w * h; }
float Volume(float r) { return r * r * 3.141592f; }
```

- **질량**

- 질량(m) = 부피(volume) * 밀도(d)

```
float Mass(float volume, float d) { return volume * d; }
```

- **상대속도**

- A가 B로 향하는 상대속도(rv) = B의 속도(bv) - A의 속도(av)

```
Vector2 RelativeVelocity(const Vector2& av, const Vector2& bv) { return bv - av; }
```

- **프레임당 중력가속도**

- 속도(gravity) = 중력(g) * FPS(deltaTime)

```
Vector2 GravityVelocity(const Vector2& g, float deltaTime) { return g * deltaTime }
```

Physics Engine

충돌 처리 판별하기

- v·n(dot) = (상대속도(rv) · 충돌 방향(n))
- **충돌 처리 판별** : 두 물체가 접촉하고 있더라도 충돌 처리를 하지 않는 경우가 있다
  (내적 > 0) : 두 물체가 멀어지는 중, **충돌 처리 하지 않는다**
  (내적 < 0) : 두 물체가 가까워 지는 중, **충돌 처리를 한다**

```
float VelocityDotNormal(const Vector2& rv, const Vector2& normal) { return Dot(rv, normal); }
```

충돌

- 충돌(im) = -(1 + 반발계수(e)) * v·n(dot) / ((1 / 물체1의 질량(m1)) + (1 / 물체2의 질량(m2)))
- 반발계수(e) : [0.0f ~ 1.0f]의 값을 가진다(예제에서는 0.02f가 사용되었음)

```
float ImpulseMagnitude(float e , float dot, float m1 , float m2)
{
    return -(1.0f + e) * dot / ((1 / m1) + (1 / m2));
}
```

충돌에 의해 발생한 속도

- 속도 (impulse) = 충돌 방향(normal) * 충돌 (im)

```
Vector2 Impulse(const Vector2& normal, float im) { return (normal * im); }
```

Physics Engine

- **가속도**
  - 가속도(**a**) = 힘(f) / 질량(m)
  - Ex) *힘(f) = 충돌에 의해 발생한 속도(impulse)*

```
Vector2 Acceleration(const Vector2& f, float m) { return f / m; }
```

- **물체의 현재 프레임 속도**
  - 현재 프레임의 속도(**v**) += 가속도(**a**);

```
void ApplyImpulse(Vector2& v, const Vector2& a) { v += a; }
```

- **프레임 당 물체의 속도**
  - 프레임 당 속도(**velocity**) = 현재 프레임의 최종 속도(**v**) * FPS(deltaTime)

```
Vector2 CalculateVelocity(const Vector2& v, float deltaTime) { return v * deltaTime; }
```

- **물체의 이동**
  - 물체의 위치(position) += 프레임 당 속도(**velocity**)

```
void ApplyVelocity(Vector2& position, const Vector2& velocity) { position += velocity; }
```

연습문제

중력이 적용되어
BOX, Circle **오브젝트가 서로 충돌**하는
SimplePhysics2D
프로젝트를 만들어 보자

- **실행 순서**

1. 물체의 속도에 **중력가속도 적용**
2. **충돌 확인**
3. 충돌한 물체들의 **충돌 가속도 적용***(3.을 여러 번 반복할 수록 더욱 세밀한 충돌 구현이 가능)*
4. **물체 이동**
5. **1.에서 4.를 반복**

- 참고 출처 : The Basics and Impulse Resolution