인공지능 - 중간

□ 인공지능

○ 인공지능 이론

* Automation(자동) -> Autonomous(자율)로 바뀌는 추세
  - 자율 : 스스로 판단하는 것

* 인간의 변화 : 손 -> 동물 -> 기계 -> 증기 -> 전기
  - 육체 적인 것은 기계로 가고 있다.

* AI정의 4가지 카테고리
  - 인간과 같이 생각하는 시스템
  - 인간과 같이 행동하는 시스템
  - 합리적으로 생각하는 시스템
  - 합리적으로 행동하는 시스템

  - 공학적 접근방법
  - 인지 공학적 접근방법 : 심리학, 생리학 등 인간이 구성하고 있고, 어떻게 돌아가는지 파악

* Frame 이론
  - 코끼리를 생각하지 마세요 => 코끼리를 생각하게 됨

* 지식처리형 시스템
  - Logic, Expert System, Fuzzy Logic 등

□ 논 인공지능

○ 상태구동형 에이전트

* 유한상태기계
  - 빠르고 코딩이 쉽다
  - 오류 수정이 용이하다
  - 계산 부담이 없다
  - 직관적이다
  - 유현성이 있다

* FSM
  - 장점 : 이해하기 쉽고 구현도 용이
  - 단점 : 인공지능의 행동을 예측 가능, 상태 수가 많아지면 표현력의 한계
  - 특별한 인공지능 기능을 요구하지 않는 게임에서 사용

* 리포트 : 10개의 상태머신을 가져오기
  - 자판기 : 대기 - 동전 투입 - 적정 가격 불 켜기 - 음료수 버튼 누르기 - 거스름돈 - 대기

□ 미분

○

* X 와 Y가 단위벡터이고, 둘을 내적하면 X * Y = cos∂

□ Code

○ 속도 갱신

*

```
/*  */
void Vehicle::Update(double time_elapssed){
    // 이전 생략

    // 운반기 리스트 내에 있는 각 조종 행동으로부터 조합된 힘을 계산
    SteeringForce = m_pSteering->Calculate

    // 가속도 = 힘 / 질량
    Vector2D acceleration = SteeringForce / m_dMass;

    // 새로운 속도 = 현재 속도 * 시간
    m_vVelocity += acceleration  time_elapsed;

    // 운반기가 확실하게 최대 속도를 넘지 않게 한다.
    m_vVelocity.Truncate(m_dMaxSpeed);

    // 속도를 갱인
    m_vVelocity += acceleration  time_elapsed;

    // 이후 생략
}
```

○ 위치 찾아가기 (Seek)

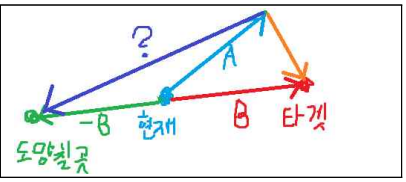
*

```
/* Seek : 위치 찾아가기 */
Vector2D SteeringBehavior::Seek(Vector2D TargetPos){
    Vector2D DesiredVelcity


}
```

○ 도망치기 (Flee)

* 타겟의 반대방향을 타겟으로 잡고 이동

```
/* Flee : 도망치기 */
Vector2D SteeringBehavior::Flee(Vector2D TargetPos){
    Vector2D DesiredVelcity


}
```



○ 도착하기 (Arrive)

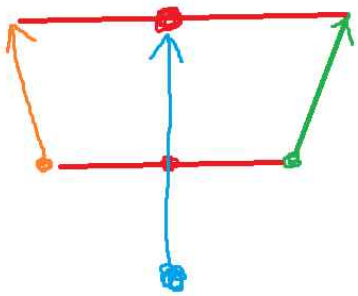
* 목표 거리에 따라 힘을 줄여줌(속도와 거리가 비례)
* 거리와 속도를 알고 있으면 도달 시간을 알 수 있음 -> 일정 범위 내에서 도달 시간을 사용하여 속도를 줄여줌
* MaxForce / MaxSpeed 지정

```
/* Arrive : 도달하기 */

Vector2D  SteeringBehavior::Arrive(Vector2D  TargetPos, Deceleration deceleration){

}
```

○ Offset

* 상대도 움직이기 때문에

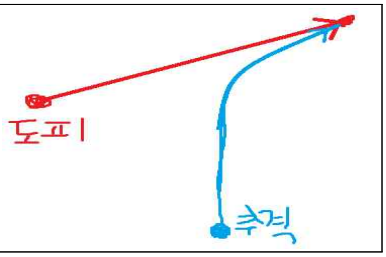
```
/* Arrive : 도달하기 */
//
// MaxForce 지정
// MaxSpeed 지정
```



○ 추격하기 (Pursuit)

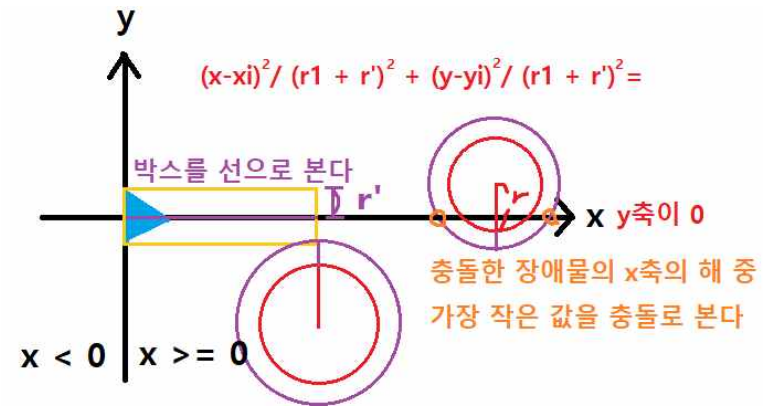
* 상대도 움직이기 때문에 상대의 현재 위치를 타겟으로 지정하면 안 됨
* 추격자와 도망치는 자의 거리만큼 걸리는 경과시간에 비례하여 도망치는 자가 도착할 곳을 타겟으로 잡아 이동한다.
* 거리 / 타겟의 maxspeed = 경과 시간 => 이 시간을 이용하여 타겟의 다음 위치를 파악
* 추적자는 내적 계산을 통해 도피자의 위치를 파악
* 미래시점 까지 예측해야 함

```
/* Arrive : 도달하기 */
//
// MaxForce 지정
// MaxSpeed 지정
```

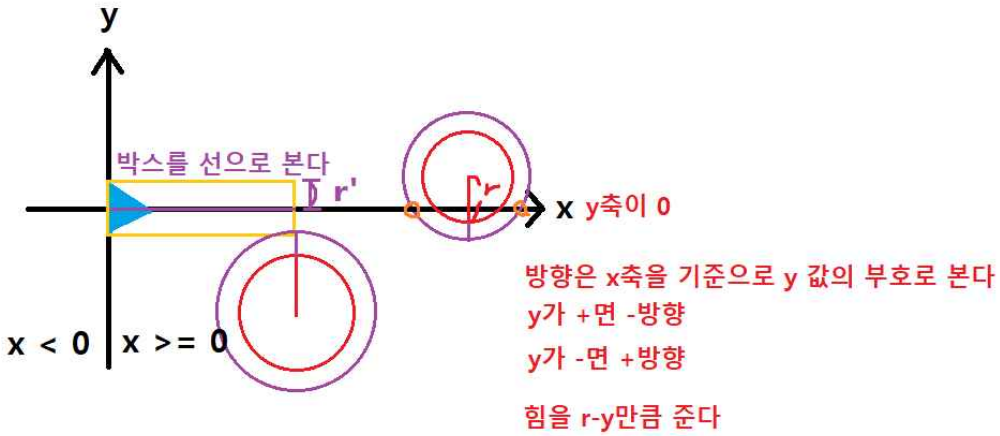


○ 장애물 피하기 (장애물 충돌 체크)

* 충돌 체크
  - AABB : 축 정렬
  - OBB : 바운딩 박스(콜리전 박스)
  - 범위를 정한다. :: disntance
  - 에이전트를 중심(0, 0)으로 두고, X축을 바라보게 둔다.
    => 장애물들의 x가 0보다 작으면 체크 대상에서 제외한다.
    => 0보다 크면 체크 대상
  - 그 만큼 장애물의 반지름을 늘려 충돌하는지 본다.
    y <= r+r' 충돌, y < r+r' 충돌X

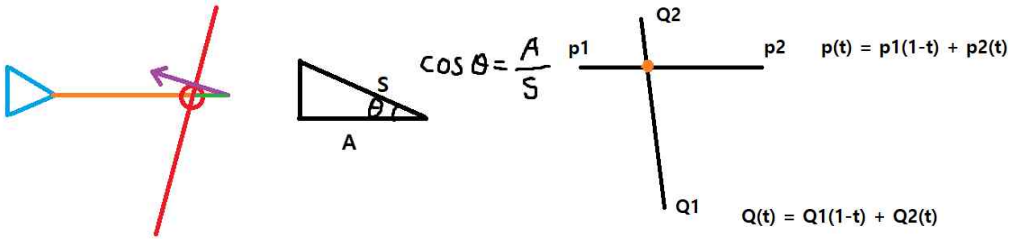


* 충돌 처리
  - 충돌한 장애물의 y 값이 +인지 -인지를 확인한다.
  - y가 +면 -방향으로, -면 +방향으로 방향을 준다. (적게 들어간 쪽으로 방향을 준다)
  - 힘은 r-y만큼 준다.



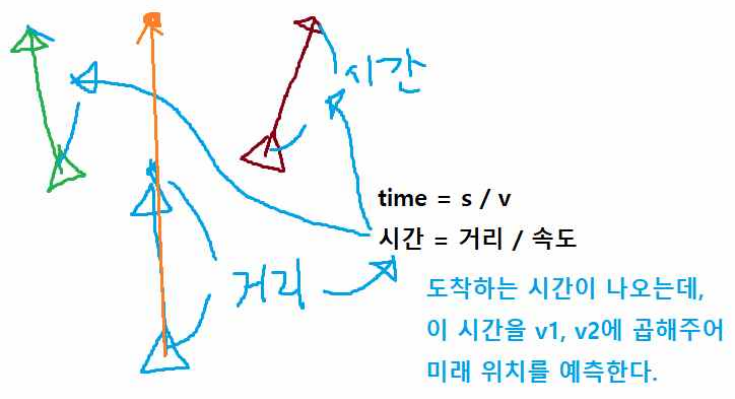
○ 벽피하기

* 벽 피하기의 경우 범위가 넓기 때문에 경계선(센서)을 지정하여 벽의 수직인 방향으로, 벽에 들어간 만큼 힘을 준다.
* 파라미터 방정식 : p(t) = p1(1-t) + p2(t) // t::0일 경우 p1(1), t::1일 경우 p2(1), t::0.5일 경우 p1(0.5)+p2(0.5)



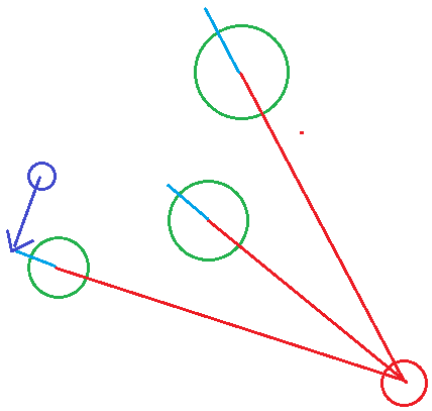
○ 끼워넣기

* v1과 v2보다 끼어드는 v3의 속도가 더 빠르다는 전제가 있다.
* v1과 v2의 미래 위치를 예측하여 그 중간지점을 목표로 둔다.



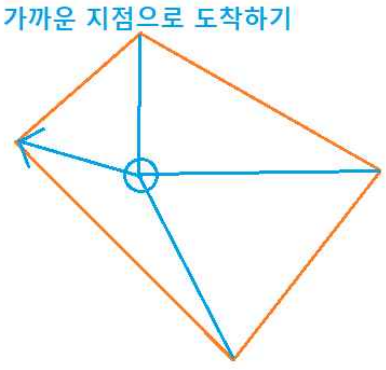
○ 숨기(Hide)

* 목표물과 숨는 벽 사이를 일직선으로 긋고, 벽 뒤에 숨는다.



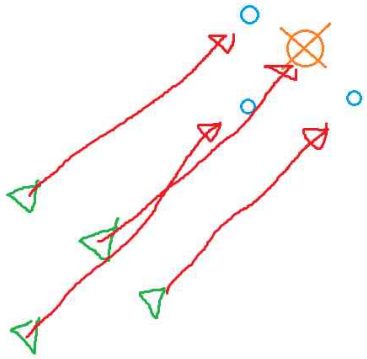
○ 경로 따라가기(Path following)

* 현재 위치에서 가장 가까운 점을 도착하기로 이동한다.



○ 오프세 추격하기(편대)(Offset Pursuit)

* 도착 위치에 따라 각각 도착하는 위치를 지정하여 도착하기로 이동한다.



○ 정렬

* 에이전트들의 무게 중심을 구하여 그 방향에 수직인 벡터로 이동하게 만든다.

○ 분리

* 에이전트들의 벡터들을 다 합쳐 그것의 반대 방향을 멀어지는 방향으로 잡는다.

○ 심플 사커

* 준비물
  - 공 : 규칙이 필요가 없고,
  - 선수
  - 골대
  - 축구장
  - 규칙

* 축구장
  - 생성자
  - 소멸자
  - void Update()
  - bool Render()

* 골대(Class Goal)
  - Vector2D m_vLeftPost;
  - Vector2D m_vRightPost;
  - 골의 방향을 나타내는 벡터
  - 골라인의 중심 위치
  - 골이 기록 될 때마다 Scored()에서 증가

* 축구공(Class SoccerBall)
  - MovingEntity에서 제공하는 기능을 덧붙여 사용
  - 공의 마지막 위치를 저장할 데이터 멤버와 공을 차고, 충돌 검사 및 공의 다음 위치를 계산하기 위한 메소드
  - 일정한 감속(음의 감속)을 적용함
  - 감속량은 초기 parameter파일 의 Friction으로 설정

* SocclerBall:FuturePosition
  - $\triangle x = u\triangle t + 1/2a\triangle t^2$
    -> $\triangle v = u + a\triangle t$을 적분해서 나온 값

    -> $\triangle x$: 움직인 거리
    -> u : 공의 속도
    -> a : 마찰로 인한 감속(-)

    -> $u\triangle t$: 거리
    -> $1/2a\triangle t^2$: u + at를 t에 대해서 적분한 값