이창민 (5702600)

Retreat Protocol

기술 문서: 적 AI

본 문서에서는 게임에 구현된 적대적 NPC의 AI의 구현 과정에 대해 기술한다.

이동

NPC는 플레이어와 일정 거리를 유지한 채 플레이어를 따라다닌다. NPC도 플레이어와 마찬가지로 직진만 가능하며, 함선을 기울여 직진 방향을 변경함으로써 이동 경로를 수정한다.

NPC는 생성 시 플레이어와 유지할 최대 거리와 최소 거리를 구한다. 최소 거리는 30 ~ 50 사이의 무작위 값이며, 최대 거리는 최소 거리에 30 ~ 100 사이의 무작위 값을 더한 값이다.

이후 NPC는 매 프레임마다 NPC와 플레이어 사이의 거리를 계산하면서, 다음과 같은 조건에 맞춰 목표 방향을 설정한다:

- NPC와 플레이어 사이의 거리가 최소 거리보다 짧다면, NPC는 플레이어를 등지는 방향을 목표 방향으로 설정한다.
- NPC와 플레이어 사이의 거리가 최대 거리보다 멀다면, NPC는 플레이어를 바라보는 방향을 목표 방향으로 설정한다.
- 그 외의 경우 현재 이동 방향을 유지한다.

목표 방향은 플레이어와 NPC의 위치 벡터의 뺄셈으로 구한다. 이 때 목표 방향은 Vector3 형으로 계산되는데, 실제로 이 값을 사용하려면 Quaternion형이 필요하기 때문에 Quaternion.LoockAt()을 호출하여 Quaternion형으로 변환한다.

이렇게 목표 방향을 설정하였다면, Quaternion.Slerp 함수로 계산한 보간 값을 이용하여 목표 방향에 도달할 때까지 서서히 rotation 값을 조절하여 NPC의 이동 경로를 수정한다.

이동 (의사 코드)

```
void Update()
{
    바라보는 방향(Vector3.foward)으로 이동한다.
    목표_방향_계산_함수 호출
    목표 방향을 바라보도록 회전(Quaternion.Slerp로 계산한 보간 값 활용)
}

void 목표_방향_계산_함수()
{
```

```
    Vector3.Distance 함수를 활용하여 플레이어와 자기 자신 사이의 거리를 계산
    if( 최소 거리보다 짧은 경우 ) 플레이어를 등지는 방향으로 목표 방향을 설정
    else if( 최대 거리보다 긴 경우 ) 플레이어를 바라보는 방향으로 목표 방향을 설정
    else 아무것도 하지 않는다. (현재 이동 방향을 유지)
}
```

공격

NPC는 공격이 준비될 때마다 플레이어의 예상 위치를 계산하고, 이 위치를 목표로 직선 방향으로 날아가는 공격 투사체를 발사한다. 플레이어의 속도와 방향이 일정할 것을 전제하기 때문에, 플레이어는 이동 경로를 수정하여 NPC의 공격을 회피할 수 있다.

현재 플레이어의 위치와 이동 방향, 이동 속도, 현재 NPC의 위치와 NPC 포탄의 이동 속도를 기반으로, 다음과 같은 절차를 따라 플레이어의 미래의 예상 위치를 계산하여 목표 위치로 삼는다:

1. 플레이어가 이동 중인 방향과 이동 속도 정보를 담은 벡터 v1을 구한다.
2. NPC의 위치에서 플레이어의 위치로 향하는 벡터 v2를 구한다.
3. NPC가 발사하는 포탄의 이동속도와 벡터 v2의 크기를 기반으로, NPC의 위치에서 벡터 v2 방향으로 발사한 포탄이 플레이어의 위치에 도달하는 데 걸리는 시간 t1을 계산한다.
4. 시간 t1 이후, 플레이어가 벡터 v1을 따라 이동한 결과 위치 p1을 계산한다.
5. NPC의 위치에서 위치 p1으로 향하는 벡터 v3를 구한다.
6. NPC가 발사하는 포탄의 이동속도와 벡터 v3의 크기를 기반으로, NPC의 위치에서 벡터 v3 방향으로 발사한 포탄이 위치 p1에 도달하는 데 걸리는 시간 t2를 계산한다.
7. 시간 t2 이후, 플레이어가 벡터 v1을 따라 이동한 결과 위치 p2를 계산한다.
8. 이렇게 계산한 p2를 조준하여 포탄을 발사한다.

NPC의 명중률이 너무 높아 게임 난이도 조절에 어려움이 있을 경우 5~7번 단계를 생략하고, 4번 단계에서 계산한 p1을 조준한다. 명중률이 너무 낮은 경우, 5~7번 단계를 반복하여 더욱 정밀한 위치를 계산한다.

공격 (의사 코드)

```
void 공격_실행
{
    공격 속도만큼 시간이 지날 때가지 대기
    If( 시간이 충분히 지나면 )
    {
        공격_위치_계산 함수를 호출하여 공격 목표 위치를 계산
        목표 위치를 향해 날아가는 투사체 생성
```

```
        공격 속도 초기화
    }
}

Vector3 공격_위치_계산
{
    현재 발사할 포탄의 이동속도 참조
    플레이어의 현재 위치와 이동 방향, 속도 정보를 담은 벡터 v1 생성
    현재 위치에서 플레이어로 향하는 방향과 포탄의 이동속도 정보를 담은 벡터 v2 생성
    벡터 v2 방향으로 날아가는 포탄이 벡터 v1 위치에 도달할 시간 t 계산
    시간 t 이후 플레이어의 예상 위치 정보를 담은 위치 p1 생성
    위치 p1을 반환
    // 명중률이 낮을 경우, 플레이어의 현재 위치에 p1을 대입하여 이 과정을 반복.
}
```

Retreat Protocol

기술 문서: 플레이어 캐릭터(원거리형)

본 문서에서는 플레이어가 게임 중 선택할 수 있는 세 가지 스타일 중 하나인 원거리형 스타일의 구현 과정에 대해 기술한다.

무기

원거리형 스타일은 다음과 같은 네 가지 무기를 사용할 수 있다:

1	**철갑소이탄**	짧은 간격으로 두 번 발사하는 2연장 함포.
2	**근접유도탄**	가까운 적을 추적하여 큰 피해를 입히는 2연장 함포.
3	**EMP자폭탄**	주변의 적을 탐지하면 폭발하여 피해를 입히면서 밀어내는 포탄.
4	**플라즈마 포**	2초 동안 에너지를 충전하여 강력한 피해를 입히는 레이저 무기.

각 무기의 최대 사거리에 따로 제한이 있진 않지만, 포탄이 플레이어로부터 300거리 이상 떨어진 경우 게임에서 제거된다.

무기1 – 철갑소이탄

조준점을 기준으로 x축에서 ±1만큼 떨어진 위치에 서로 평행하게 날아가는 두 개의 포탄을 0.2초 간격으로 두 번 발사하여 총 4발의 포탄을 발사한다. 철갑소이탄에 명중한 적은 4초동안 방어력이 20% 감소한다.

철갑 소이탄에 적중한 적의 방어력을 떨어트린 후, 4초 후에 다시 방어력을 복구해야 하는데, 이 시점엔 이미 철갑소이탄이 적중하여, 방어력을 복구할 포탄 오브젝트가 사라진 상태이다. 이를 해결하기 위해 적의 코드에서 방어력을 초기화하는 함수를 구현하고, 해당 함수를 호출하는 방법으로 구현한다.

```
class 철갑소이탄_무기
{
    void 투사체_발사
    {
        투사체_생성 함수 호출
        0.2초간 대기
        투사체_생성 함수 호출
    }

    void 투사체_생성
    {
```

```
        for( int i = -1 ; i < 2 ; i += 2 )
        {
            카메라.position.x + i 위치에 카메라와 동일한 rotation을 가지는 투사체 생성
        }
    }
}

class 철갑소이탄_포탄
{
    void Update()
    {
        바라보는 방향으로 이동한다
        플레이어와 자기 자신 사이의 거리를 계산
        if( 현재 거리 > 300 ) 자기 자신을 삭제
    }

    void OnTriggerEnter(Collider 적)
    {
        적.HP -= 포탄 공격력
        적.방어력 = -20
        적.방어력_복구(4) 함수 호출
        자기 자신을 삭제
    }
}

class 적
{
    void 방어력_복구(int 대기시간(초))
    {
        대기시간(초) 동안 대기
        방어력 초기화
    }
}
```

무기2 – 근접유도탄

조준점을 기준으로 x축에서 ±1만큼 떨어진 위치에 서로 평행하게 날아가는 두 개의 포탄을 발사하여 총 2발의 포탄을 발사한다. 근접유도탄은 날아가는 도중 주변의 적을 발견하면 해당 적의 위치로 방향을 변경한다.

```
class 근접유도탄_무기
{
    void 투사체_발사
    {
        for( int i = -1 ; i < 2 ; i += 2 )
        {
            카메라.position.x + i 위치에 카메라와 동일한 rotation을 가지는 투사체 생성
        }
    }
}

class 근접유도탄_포탄
{
    void Update()
    {
        바라보는 방향으로 이동
        if( 적 발견 = false ) 적_탐색 함수 호출
    }

    void 적_탐색
    {
        Enemy 태그를 가진 모든 오브젝트를 검색하여 적_리스트 생성
        foreach( 적 in 적_리스트 )
        {
            자기 자신과 오브젝트 사이의 거리를 측정(Vector3.Distance)
            if( 거리 < 탐색 거리 ) LookAt(오브젝트)
        }
    }

    void OnTriggerEnter(Collider 적)
    {
        적.HP -= 포탄 공격력
        스스로를 삭제
    }
}
```

무기3 - EMP자폭탄

조준점 방향으로 정확하게 날아가는 하나의 포탄을 발사한다. EMP자폭탄은 날아가는 도중 주변의 적을 발견하면 폭발하며, 폭발에 휘말린 적들을 피해를 입고, 폭발의 중심부로부터 외각 방향으로 밀쳐진다.

```
class EMP자폭탄_무기
{
    void 투사체_발사
    {
        EMP자폭탄 오브젝트를 생성하여 바라보는 방향으로 rotation 설정
    }
}

class EMP자폭탄_포탄
{
    void Update()
    {
        바라보는 방향으로 이동한다
        적_탐색 함수 호출
        플레이어와 자기 자신 사이의 거리를 계산
        if( 현재 거리 > 300 ) 스스로를 삭제
    }

    void 적_탐색
    {
        Enemy 태그를 가진 모든 오브젝트를 검색하여 적_리스트 생성
        foreach( 적 in 적_리스트 )
        {
            자기 자신과 오브젝트 사이의 거리를 측정(Vector3.Distance)
            if( 거리 < 탐색 거리 ) EMP폭발(적_리스트) 함수 호출
        }
    }

    }

    void EMP폭발(list 적_리스트)
    {
        폭발 연출용 오브젝트를 생성
        foreach( 적 in 적_리스트 )
        {
```

```
            자기 자신과 오브젝트 사이의 거리를 측정(Vector3.Distance)
            if( 거리 < 폭발 거리 )
            {
                적.HP -= 포탄 공격력
                폭발의 외각 방향으로 밀쳐낸다(AddForece.Impulse)
            }
        }
        스스로를 삭제
    }
}
```

무기4 – 플라즈마 포

발사 시 조준 방향을 기억하고, 2초 후 기억한 방향으로 발사되어 접촉하는 모든 적에게 피해를 입히는 레이저 포를 발사한다.

일반적인 포탄과 달리 투사체가 아니고, 접촉하는 모든 적에게 피해를 입혀야 하기 때문에 BoxCollider 컴포넌트를 가진 광선 오브젝트를 생성하는 방식으로 구현한다.

기본적으로 BoxCollider는 박스의 한 가운데를 중점으로 삼기 때문에 광선이 플레이어로부터 양옆으로 발사되는 모양을 보이는데, 이는 Center 값을 조절함으로써 콜라이더의 중점 위치를 변경하여 해결할 수 있다.

```
class 플라즈마포_무기
{
    void 광선_발사
    {
        현재 카메라의 각도 저장
        2초간 대기
        저장해둔 각도와 동일한 rotation을 가지는 광선 오브젝트 생성
    }
}

class 플라즈마포_광선
{
    void Start()
    {
        4초간 대기 후 스스로를 삭제 (Invoke)
    }
```

```
    void OnTriggerEnter(Collider 적)
    {
        적.HP -= 광선 공격력
        스스로를 삭제
    }
}
```

스킬

원거리형 스타일은 다음과 같은 두 가지 스킬을 사용할 수 있다:

Q	**전자기장**	함선 주변에 있는 적 포탄의 이동을 멈춘다.
E	**비상 발전**	다음 번에 사용하는 무기의 대기시간을 제거하여 곧바로 충전시킨다.

스킬1 – 전자기장

사용 시 씬 내의 적 포탄을 모두 검색한 후, 플레이어 함선 주변의 포탄만 골라 이동속도를 0으로 만들어 이동을 멈추게 한다. 사용 후 12초가 지나면 스킬을 재사용할 수 있다. 전자기장에 의해 이동이 멈춘 포탄은 시간이 지나도 이동속도를 회복하지 않는다.

```
void 사용
{
    EnemyBullet 태그를 가진 모든 오브젝트를 검색하여 적포탄_리스트 생성
    foreach( 적_포탄 in 적포탄_리스트 )
    {
        플레이어와 적_포탄 사이의 거리를 측정 (Vector3.Distance)
        if( 측정 거리 < 기준 거리 ) 적_포탄.moveSpeed = 0
    }
}
```

스킬2 – 비상 발전

사용 시 플레이어는 비상 발전 상태에 돌입하고, 비상 발전 상태에서 발사하는 첫번째 무기의 대기시간을 즉시 0으로 만들어 제거한다. 이후 비상 발전 상태를 해제한다. 사용 후 (비상 발전 상태에 돌입한 순간으로부터) 12초가 지나면 재사용할 수 있다.

비상 발전을 사용한 후 12초 동안 아무런 무기도 발사하지 않고 있다가, 한 번 더 비상 발전을 사용하더라도 대기시간이 두 번 제거되진 않는다.

플레이어가 무기를 선택하여 발사한 후, 무기 코드에서 스스로 자신의 대기시간(쿨타임)을 초기

화 하는 게 아니라, 플레이어 코드에서 방금 발사한 무기의 대기시간을 초기화하는 함수를 호출해주는 구조로 구현되어 있다. 여기서, 비상발전을 사용한 후에는 쿨타임을 초기화하지 않도록 조건문을 추가하는 방법으로 구현한다.

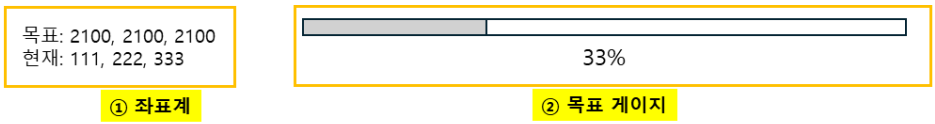
```
class 비상발전
{
    void 사용
    {
        플레이어.비상발전_버프 = true
    }
}

class 플레이어
{
    void 공격
    {
        선택된_무기.포탄_발사 함수 호출
        if( 비상발전_버프 == false ) 선택된_무기.쿨타임_시작 함수 호출
        else 비상발전_버프 = false
    }
}
```

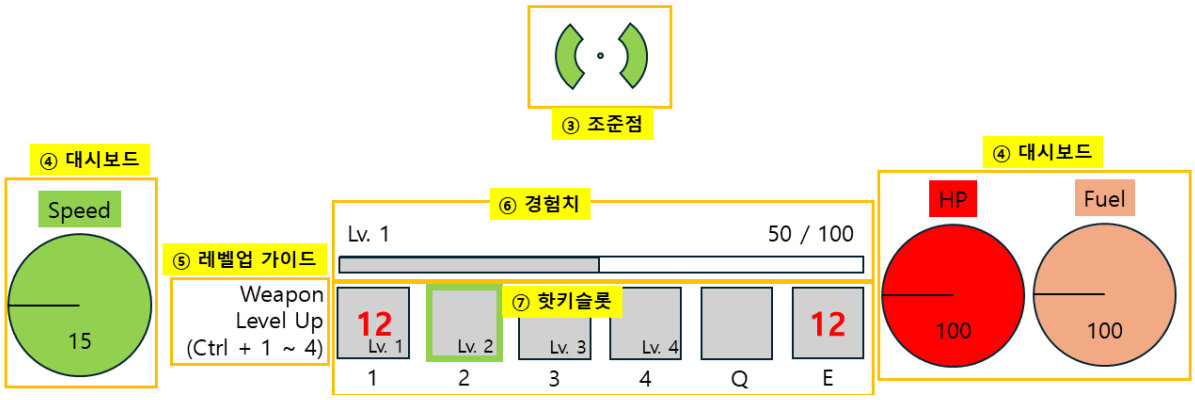
Retreat Protocol

기술 문서: UI

본 문서에서는 게임에 사용된 UI의 구현 과정에 대해 기술한다. 특별한 기술 없이 유니티에서 제공하는 버튼 컴포넌트만으로도 구현 가능한 타이틀 화면과 함선 선택 화면의 UI는 생략하고, 게임 중 화면 UI에 대해서만, 그 중에서도 특별한 기술을 사용한 UI의 구현 과정만 기술한다.

게임 중 화면





목표 게이지

현재 게임 진행도(플레이어가 목표에 가까운 정도)를 직관적으로 표시하기 위한 게이지이다.

게임 시작 시 플레이어와 목표 위치 사이의 거리 md를 계산하여 저장하고, 매 프레임마다 현재 플레이어와 목표 위치 사이의 거리 cd를 계산한다. 이후 md와 cd의 비를 계산하여 현재 게임 진행도(백분율)를 산출한다.

게이지는 이렇게 계산한 현재 진행도를 기반으로 표시된다. 0%일 경우 게이지가 완전히 비어 있고, 100%일 경우 완전히 차오른다. 그 외의 경우 현재 진행도에 맞춰 적절히 게이지를 채운다. 현재 게임 진행도를 보다 정확하게 알 수 있도록 게이지 하단에 텍스트(정수)로도 표시한다.

게이지는 두 개의 막대 이미지, 게이지의 배경이 될 이미지 BG와 게이지 내부에서 차오를 이미지 G로 구성된다. G의 x pivot을 0으로 설정하면 x축의 왼쪽 끝을 중점으로 삼게 되는데, 이 상태

에서 막대 이미지의 width 값을 조절할 경우, 중점을 기준으로 늘었다 작아지므로 게이지가 차는 연출을 구현할 수 있다.

```
void Update()
{
    게이지_출력 함수 호출
}

int 백분율_계산
{
    현재 플레이어와 골 오브젝트 사이의 거리를 계산(Vector3.Distnace 활용)
    현재 거리와 최대 거리의 백분율을 계산
    if( 현재 거리 값 < 최대 거리 값 ) 백분율 = 0
    이렇게 계산한 백분율을 반환
}

void 게이지_출력
{
    백분율_계산 함수 호출하여 백분율을 반환 받음
    백분율의 비율만큼 G의 width를 BG의 n%로 계산
    계산한 G의 width를 실제로 적용(Mathf.lerp 보간 활용)
    현재 백분율을 텍스트로도 출력(TextMeshProUGUI 활용)
}
```

대시보드

함선의 속도와 HP, 연료를 직관적으로 표시하는 대시보드다. 속도 대시보드는 화면의 좌측 하단에, HP와 연료 대시보드는 화면의 우측 하단에 표시한다. 대시보드는 바늘과 텍스트로 구성되어 있다. 바늘은 플레이어에게 현재 값을 추상적으로 보여주고, 텍스트는 정확한 값으로 보여준다.

대시보드가 가리키는 변수의 현재값과 최대값의 비를 백분율로 계산하고, 계산한 백분율에 맞춰 적절한 값을 가리키도록 바늘의 각도를 조정한다. 바늘의 각도가 변경될 때, 보간을 사용하여 부드럽게 변경되도록 한다.

대시보드의 바늘은 긴 막대 이미지를 사용하는데, 막대 이미지의 x pivot을 1로 설정하면 x축의 오른쪽 끝을 중심으로 삼을 수 있다. 막대를 회전시키면 중점을 중심으로 회전하기 때문에, 대시보드의 바늘이 움직이는 연출을 구현할 수 있다.

하나의 코드로 서로 다른 값을 가지는 세 가지 대시보드를 출력하기 위해, 오브젝트의 이름을 기반으로, 해당 대시보드가 보여야 하는 변수를 따로 참조하여 출력하게 한다.

```
void Start()
{
    gameObejct.name을 기반으로 자신이 담당할 변수의 이름을 string 형으로 저장
    string 값을 기준으로 변수를 PlayerCtrl 클래스에서 검색 (typeof().GetField() 활용)
    검색한 변수의 최대값, 최소값 변수도 함께 검색
    검색한 변수들을 float 형으로 변환하여 저장
}

void Update()
{
    바늘_각도_적용 함수 호출
}

int 바늘_각도_계산
{
    대시보드로 출력할 변수의 최대값과 현재 값의 비를 백분율로 계산
    if( 현재값 < 최소값 ) 백분율 = 0
}

void 바늘_각도_적용
{
    바늘_각도_계산 함수를 호출하여 백분율을 반환 받음
    백분율을 기반으로 0에서 180 사이의 각도 값을 계산
    계산한 각도 값을 바늘 이미지의 rotation에 적용(Mathf.lerp 보간 활용)
}
```

핫 키 슬롯

플레이어가 사용 가능한 무기와 스킬의 아이콘, 재사용 대기시간, 레벨, 단축키를 표시한다.

무기와 스킬의 레벨이 1미만인 경우, 아이콘에 (1,1,1,0.5f) 색상을 적용하여 흐리게 표시한다. 레벨이 1이상인 경우, 아이콘에 (1,1,1,1) 색상을 적용하여 또렷하게 표시한다.

무기는 4레벨로 분화되기 때문에, 아이콘의 우측 하단 모서리에 현재 레벨을 텍스트로 표시한다. 스킬은 레벨이 분화되지 않기 때문에, 따로 레벨을 표시하지 않는다.

현재 선택된 무기의 경우, 아이콘 테두리의 색상을 변경하여 강조한다.

무기 또는 스킬이 재사용 대기상태(쿨타임)인 경우, 남은 대기시간을 아이콘의 중앙에 텍스트(정수)로 표시한다. 대기시간은 1초마다 1씩 줄어든다.

```
void Update()
{
    if( 플레이어의 무기 선택 감지 시 )
    {
        if( 지금 선택한 무기가 이미 선택된 무기와 다른가? )
        {
            현재 선택된 무기 = 지금 막 선택한 무기
            현재 선택된 무기의 아이콘 테두리를 강조
        }
        else 아무것도 하지 않음
    }

    무기_쿨타임_표시 함수 호출
    무기_레벨_표시 함수 호출
}

void 무기_쿨타임_표시
{
    if( 현재 쿨타임 > 0 ) 현재 쿨타임을 텍스트로 출력 (TextMeshProUGUI 활용)
    else 현재 쿨타임을 공백으로 출력 (TextMeshProUGUI)
}

void 무기_레벨_표시
{
     if( 현재 레벨 < 0 ) 아이콘 색상을 new Color(1,1,1,0.5f)로 변경
     else 아이콘 색상을 new Color(1,1,1,1)로 변경

     현재 레벨을 텍스트로 출력 (TextMeshProUGUI)
}
```