3DGP1 과제01

2022-05-05

목 차

교과목 분반	(01반)	학과	게임공학전공
학번	2018192017	이름	백승호

한국공학대학교

Ⅰ. 과제에 대한 목표

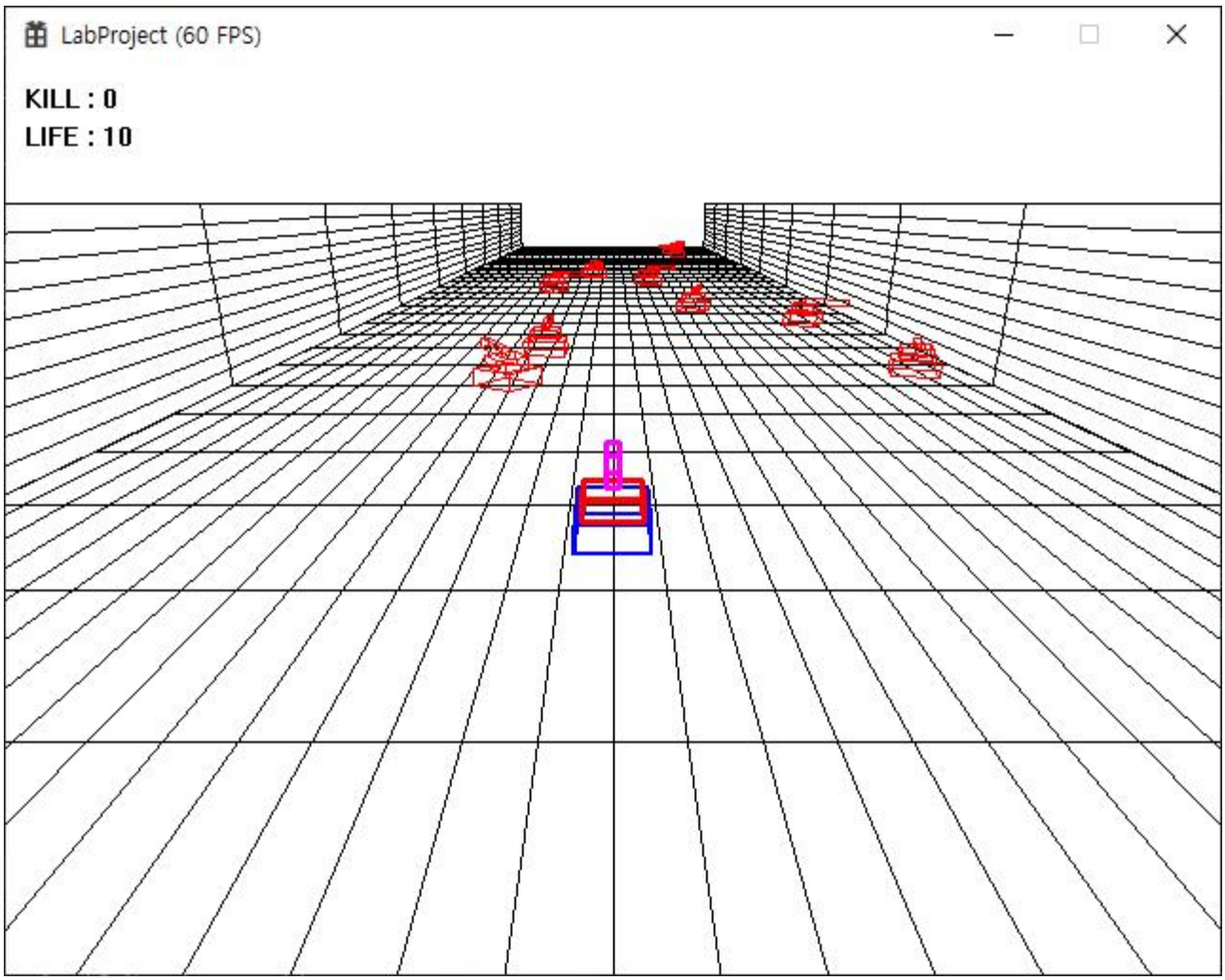
■ 과제 목표

- 3DGP1 과제01은 DirectX, DirectX Math, DirectX Collision 라이브러리의 사용법을 익히고 게임수학, 선형 대수학, 전산 물리학 등의 프로그래밍 구현을 목표로 하였다. 최대한 다양한 라이브러리 함수를 사용을 하여 디자인 패턴을 이용한 플레이어의 액션, AI등 구현을 추가 목표로 하였다.

Ⅱ. 실행 결과 및 구현 내용

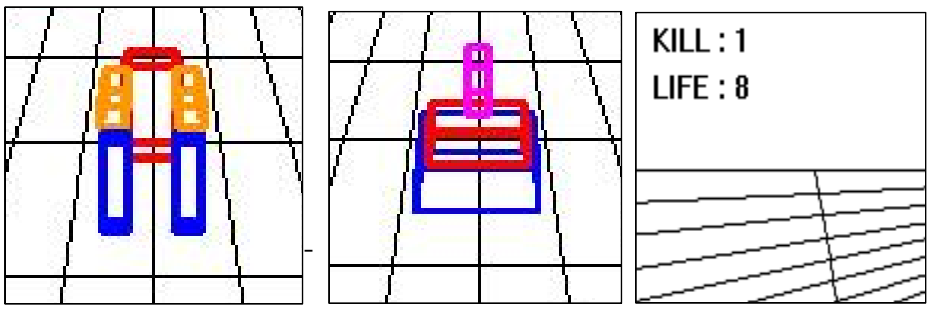
■ 초기 화면

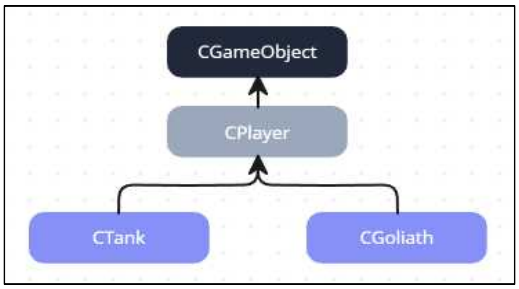
- 게임 시작 시 탱크 플레이어가 중앙에 위치하며 적 탱크 객체 10개가 랜덤하게 위치 한다. 각 양 옆은 벽으로 둘러 쌓여 있으며 충돌처리를 통해 객체들이 지나갈 수 없도록 구현하였다. GameFrameWork에 객체들이 담겨져 있으며 모두 GameObject를 상속받고 있기 때문에 GameObject로 관리한다. 또한 게임 내 모든 객체는 스마트 포인터로 이루어져 있기 때문에 자동으로 메모리를 관리한다.



■ 플레이어

- 플레이어는 총 2가지가 있으며 골리앗과 탱크를 고를 수 있다. 골리앗은 직선 방향으로 작은 총알을 빠르게 발사한다. 또한 빠른 속도를 가지고 있다. 탱크는 중력의 영향을 받으며 포물선 운동을 하며 날아가는 포탄을 멀리 발사한다. 바닥에 닿았을 시 커다란 폭발을 일으키며 또한 느린 속도를 가지고 있다.
또한 플레이어는 10의 목숨을 가지며 다 닳았을 시 초기화 되어 다시 시작한다.





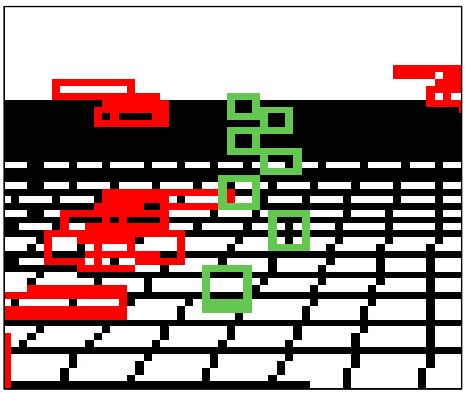
탱크와 골리앗은 플레이어를 상속받으며 플레이어는 게임 오브젝트를 상속 받는다. GetObjectType() 함수를 통해 플레이어 타입을 반환받을 수 있으며 각각의 부품에 대하여 shared_ptr<CGameObject> 혹은 vector<shared_ptr<CGameObject> 객체를 가지고 있다. 스마트 포인터이기 때문에 자동으로 메모리 관리가 된다. 또한 virtual void FireBullet함수를 통해서 각각의 플레이어에 맞는 공격을 할 수 있도록 다형성을 구현하였다.

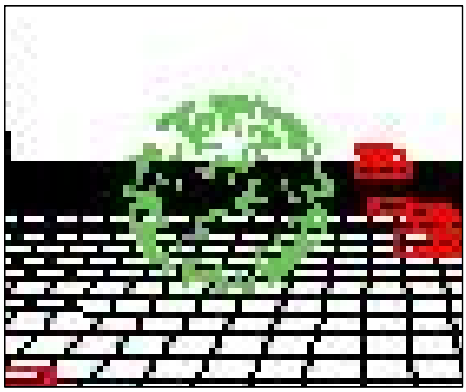
```
std::shared_ptr<CPlayer> CPlayer::ChangeToType(std::shared_ptr<CCamera> pCamera)
{
	std::shared_ptr<CPlayer> newPlayerPtr;
	if (GetObjectType() == PlayerType::Tank) {
		auto playerPtr = std::dynamic_pointer_cast<CTank>(shared_from_this());
		std::shared_ptr<CGoliath> goliath = std::make_shared<CGoliath>();
		newPlayerPtr = std::dynamic_pointer_cast<CPlayer>(goliath);
		newPlayerPtr->SetPosition(XMFLOAT3{ playerPtr->GetPosition() });
		///
	}
	if (GetObjectType() == PlayerType::Goliath) {
		auto playerPtr = std::dynamic_pointer_cast<CGoliath>(shared_from_this());
		std::shared_ptr<CTank> tank = std::make_shared<CTank>();
		newPlayerPtr = std::dynamic_pointer_cast<CPlayer>(tank);
		newPlayerPtr->SetPosition(XMFLOAT3{ playerPtr->GetPosition() });
		///
	}
	return newPlayerPtr;
}
```

위 코드를 통해 플레이어 오브젝트는 자신의 멤버함수 내에서 자식 클래스 골리앗과 탱크를 캐스팅 하여 변화가 가능하다. 모든 객체가 스마트 포인터로 구성되어 있기 때문에 CGameObject는 std::enable_shared_from_this<CGameObject>를 상속받는다.

■ 플레이어 공격

- 좌측 사진은 골리앗의 총알 공격이며 우측 사진은 탱크의 포물선 포탄 공격이다. 골리앗의 총알 공격은 각각의 총에서 번갈아가며 좌측 우측에서 총알이 나아간다. 탱크의 포탄 공격은 포물선을 따라서 중력의 영향을 받으며 바닥에 닿았을 시 터지게 된다. 적이 총알에 3번 맞게되면 사망후 리스폰하게 된다.





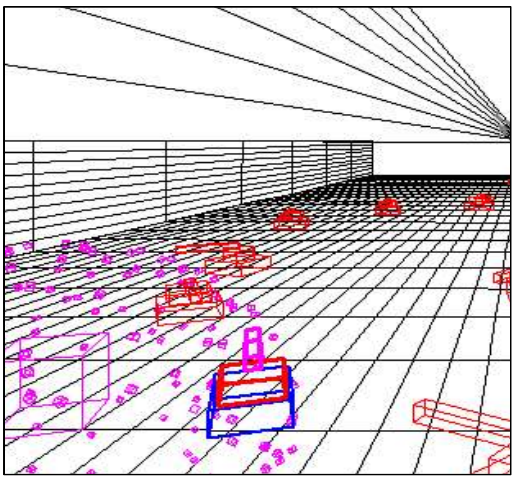
```
XMFLOAT3 gravityDirection{ 0, -1, 0 };
float gravity = cGravity * 0.5f * pow(m_fElapsedTimeAfterFire, 2) * m_fMess;
gravityDirection = Vector3::ScalarProduct(gravityDirection, gravity);
xmf3Position = Vector3::Add(xmf3Position, gravityDirection);
```

다음과 같이 간단한 수식을 통해 y축의 음의 방향으로 중력을 시간에 따라서 적용하였다.

■ 플레이어 벽 충돌

- 플레이어는 벽에 닿았을 시 벽과 충돌 처리를 하여 밀려난다. 다음과 같이 플레이어가 벽과 충돌하게 되면 자신의 진행 방향의 역방향으로 움직여지며 나아갈 수 없게 된다.

```
for (auto& object : m_vpWallObjects)
        if (object->m_xmOOBB.Intersects(m_pPlayer->m_xmOOBB)) {
                XMFLOAT3 inverseLook = Vector3::ScalarProduct(m_pPlayer->m_xmf3Shift, -1);
                m_pPlayer->Move(inverseLook, false);
        }
```



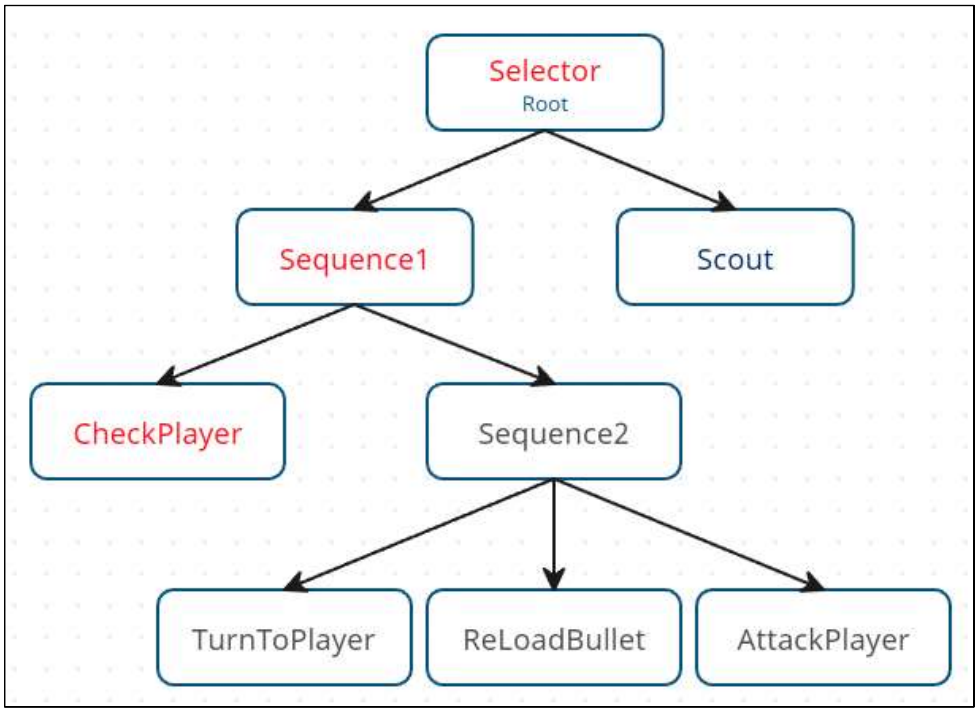
RoundRotate 함수를 를 통해 오른쪽 마우스 클릭으로 카메라만 자전을 하여 주위를 둘러볼 수 있다.

```
if (pKeyBuffer[VK_RBUTTON] & 0xF0)
        m_pPlayer->m_pCamera->RoundRotate(cxMouseDelta);
```

■ 적(AI)

- 가장 중심적으로 구현한 부분이며 행동 트리(Behavior Tree)와 명령 패턴을 이용해서 적의 행동 액션을 구현하였다. 적은 상태(Status)를 갖으며 쿨타임과 타겟을 찾았는지를 확인한다. 행동 트리는 Selector 노드와 Sequence 노드로 구성되어 있으며 Selector 노드는 행동이 통과 되지 않았다면 다른 행동을 하게 되고 Sequence 노드는 행동을 순차적으로 실행한다. 노드는 행동 트리를 적용시킬 적 객체와 적 객체의 상태를 멤버로 가지고 있기 때문에 어떠한 객체도 행동 트리에 적용이 가능하다. 각 행동들은 노드를 상속받으며 virtual run을 통해 각각의 행동에 맞는 실행이 가능하다. 또한 Composite노드는 트리를 관리하며 자식을 리스트 형태로 갖고 있고 자식을 추가하거나 반환받을 수 있다.

■ 적(AI) - 행동트리



- 적의 액션은 다음과 같이 구성되어 있는 행동 트리를 따른다. 행동 트리는 좌측에서부터 우측으로 진행되며 위 행동을 해석해보면 우선 Root 노드가 Selector로 되어 있기 때문에 타겟을 확인하고 자신의 유효거리에 들어오지 못했다면 정찰을 한다. 만약 자신의 유효 사거리에 들어 왔다면 다음 시퀀스로 넘어가며 시퀀스2에서는 차례대로 행동을 실행한다. 우선 자신과 타겟의 위치 거리 관계를 확인하여 포신을 타겟쪽으로 돌린다. 모두 돌렸다면 현재 장전이 되어 있는 상태인지를 확인한다. 만약 장전이 되어 있다면 타겟에 공격한다. 공격을 한 후 다시 처음으로 돌아가는데 이 때, 쿨타임을 확인하며 쿨타임이 만약 일정한 시간 내라면 포신을 돌리지 않고 가만히 놔둔다. 그리고 쿨타임이 모두 찼으면 다시 포신을 돌리고 장전을 하며 타겟에게 다시 포탄을 발사한다. 시퀀스에서는 하나라도 실행되지 못했다면 return false를 반환하여 상위 부모의 Selector 형제로 돌아가 다른 행동을 실행한다.

```
Selector* root = new Selector;
Sequence* sequence1 = new Sequence, * sequence2 = new Sequence;
Status* status = new Status{0.f, 3.f, false};
BT_CheckPlayer* checkPlayer = new BT_CheckPlayer(status, this);
BT_ScoutOn* scoutOn = new BT_ScoutOn(status, this);
BT_TurnToPlayer* turnToPlayer = new BT_TurnToPlayer(status, this);
BT_Reload* reLoad = new BT_Reload(status, this);
BT_Attack* attackPlayer = new BT_Attack(status, this);
```

다음과 같이 CEnemy를 상속 받는 CETank 클래스는 위 행동 트리를 갖는다.

```
root->addChild(sequence1);
root->addChild(scoutOn);
sequence1->addChild(checkPlayer);
sequence1->addChild(sequence2);
sequence2->addChild(turnToPlayer);
sequence2->addChild(reLoad);
sequence2->addChild(attackPlayer);
```

```
void CETank::Animate(float fElapsedTime)
{
    if (m_bActive) {
        CEnemy::Animate(fElapsedTime);

        status->elapsedTime = fElapsedTime;
        status->coolTime += fElapsedTime;
        root->run();

        UpdateBoundingBox();
        ComputeWorldTransform(NULL);
```

CETank 객체는 앞서 말한 행동 트리를 갖으며 위 사진과 같이 각각의 노드에 자식 노드를 설정할 수 있고 Animate 함수 내에서 root->run()을 한번만 실행하게 되면 트리에 맞게 각각의 virtrual run을 실행하게 된다.

■ 적(AI) - 액션

다음 액션은 적이 주변을 정찰하는 액션 클래스이다. 중요한 부분이니 차례대로 설명하도록 한다.

```
bool BT_ScoutOn::run()
{
        if (!status->checkTarget) {
                m_pSelf->Scout();
        }

        return true;
}
```

모든 액션은 bool 값을 리턴하며 시퀀스 노드가 성공했는지 실패했는지를 반환하며 실패했다면 다음 액션으로 넘어가지 않고 다시 행동 트리를 처음부터 시작하게 된다.

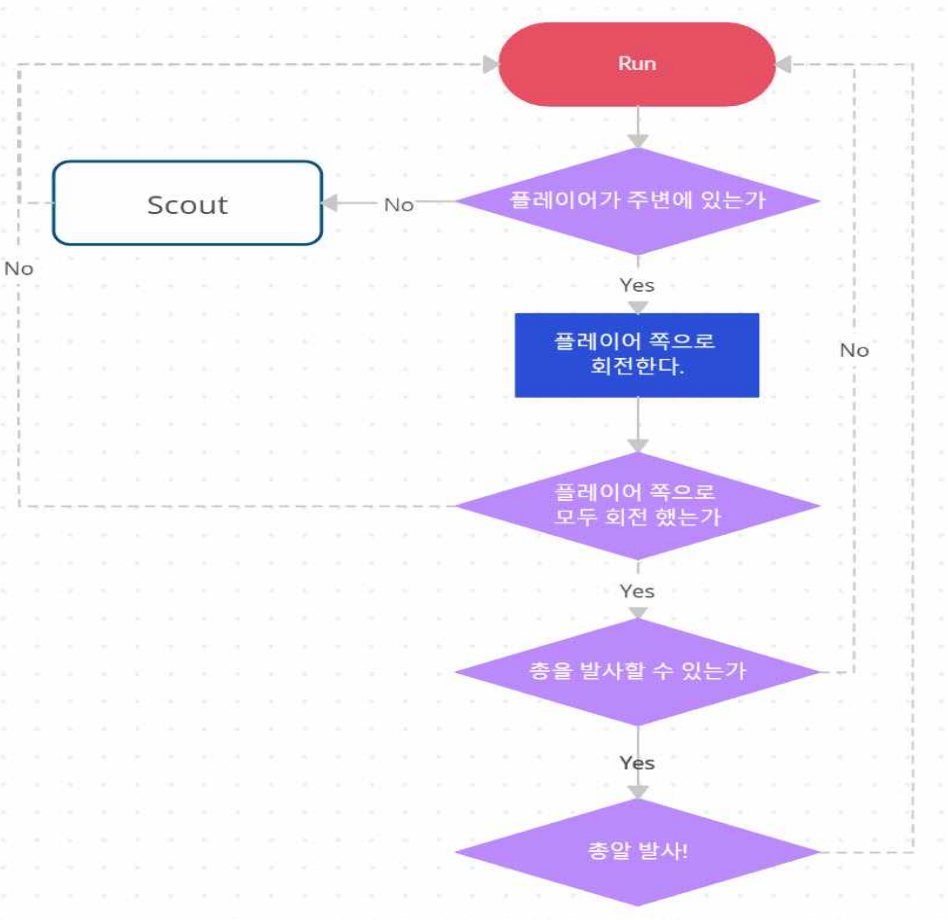
```
void CETank::Scout()
{
    // 정찰중일 경우 새로운 정찰 좌표만큼 회전하고 회전이 끝났을시 이동한다.

    XMFLOAT3 ScoutPoint = Vector3::Subtract(m_xmf3ScoutPoint, m_xmf3Position);
    XMFLOAT3 ScoutDirection = Vector3::Normalize(ScoutPoint);

    // 스칼라 삼중적의 부호 확인 0보다 작을 경우 반시계 회전
    float scalarTripleProduct = Vector3::CrossProduct(m_xmf3Look, ScoutDirection, false).y;
    float angle = XMConvertToDegrees(Vector3::Angle(m_xmf3Look, ScoutDirection));
    if (angle > 1.f) {
        if (scalarTripleProduct < 0.f)
            Rotate(0, -m_fRotationSpeed * status->elapsedTime, 0);
        else
            Rotate(0, m_fRotationSpeed * status->elapsedTime, 0);
    }
    else {
        if (Vector3::Length(ScoutPoint) < 0.3f) {
            XMFLOAT3 newPointValue = XMFLOAT3{ (float)m_urd(m_dre), 0, (float)m_urd(m_dre) };
            m_xmf3ScoutPoint = Vector3::Add(m_xmf3ScoutPoint, newPointValue);
        }
        Move(m_xmf3Look, m_fSpySpeed);
    }
}
```

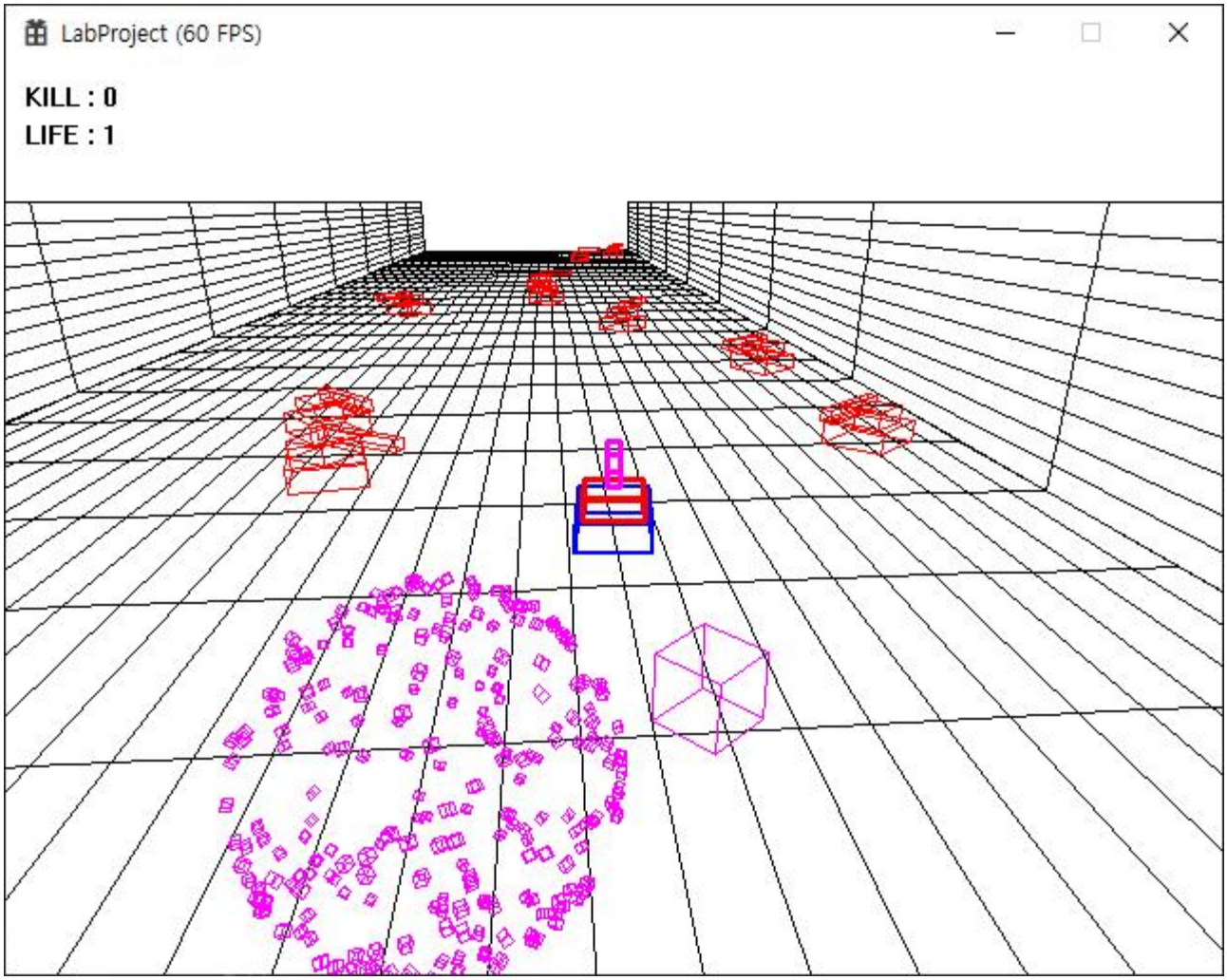
위 함수는 CETank의 정찰하는 액션으로 우선 목표지점과 자신의 거리와 방향을 저장한다. 다음 자신의 진행 방향과 목표지점까지의 거리를 외적하여 값을 저장한다. 스칼라 삼중적이지만 y축이 없기 때문에 외적만으로도 회전 방향을 정할 수 있다. 만약 삼중적 값이 음수이면 반시계로 회전하고 양수면 시계방향으로 회전하여 타겟으로부터 가장 빠른 회전을 구한다. 목표지점까지의 회전이 완료 되었다면 이때 목표지

점으로 이동한다. 위 코드에서 ScoutPoint는 최종 정찰 지점의 위치이다. 따라서 CETank객체는 ScoutPoint 방향으로 회전한 후 회전이 완료 되었다면 그 정찰 지점으로 이동한다. 만약 이동도 완료되었다면 새로운 값을 정찰 현재 정찰 지점에 더해줌으로써 새로운 정찰 지점을 확보한다. 이 후 다시 반복한다. ScoutOn 함수가 실행되기 위한 조건은 CETank객체의 타겟과 자신의 거리가 유효 범위 내에 들어오지 못하였을 경우이다. 만약 유효 범위 내에 들어온 상태라면 Selector의 다른 함수 BT_CheckPlayer를 호출한다.

```
bool BT_CheckPlayer::run()
{
	float distance = Vector3::Distance(m_pSelf->GetPosition(), m_pSelf->m_pTarget->GetPosition());
	if (distance < 50.f) {
		status->checkTarget = true;
		return true;
	}
	status->checkTarget = false;
	return false;
}
```

위 CheckPlayer 액션은 타겟과 자신의 거리를 계산하여 거리가 50.f 이내일 경우 자신의 현재 상태인 checkTarget을 true 혹은 false로 설정한다. 만약 거리가 50.f 이내가 되었을 경우 checkTarget의 플래그가 켜지며 true를 리턴하고 다음 시퀀스로 넘어가게 되는데 다음 시퀀스에 가장 왼쪽 첫 번째에 해당하는 TurnToPlayer가 실행되게 된다. 위 행동 또한 앞서 설명한 정찰에서의 스칼라 삼중적, 이동과 비슷하므로 넘어가도록 하겠다. 만약 플레이어 쪽으로 적 탱크가 모두 회전이 완료되었다면 true를 리턴하고 다음 시퀀스 재장전으로 넘어가게 된다. 쿨타임은 시간이 계속 더해지며 업데이트 되고 만약 쿨타임이 일정한 시간이 지났다면 총알을 발사할 수 있는 상태가 되고 총알을 발사하였다면 쿨타임이 다시 0.f로 넘어가며 다시 시간이 더해지게 된다. 따라서 class BT_Reload에서 true를 반환하였다면 총알을 발사해도 된다는 것이고 결국 다음 시퀀스 BT_Attack으로 넘어가게 된다. BT_Attack에서는 단순히 m_pSelf->FireBullet() 함수를 통해 자신에 등록되어 있는 객체가 총알을 발사하게 된다. 다음과 같이 플로우 차트로 간단하게 표현한다.





다음과 같이 플레이어가 적의 유효 범위 내에 들어오면 적은 포신을 플레이어 쪽으로 돌리고 모두 돌렸다면 총알을 발사하며 3초 후에 다시 플레이어가 유효 범위 내에 들어온다면 적은 포신을 돌리고 총알을 쏘게 되는 것이다. 그 밖의 유효 범위에 들어오지 않는 적들은 자신의 주변을 정찰하고 있다. 현재는 플레이어가 벽에 부딪히면 밀려나게 되지만 적이 만약 벽에 부딪히게 된다면 적은 아래 코드와 같이 자신의 진행 방향의 반사 벡터에 스칼라 10.f를 곱하여 자신의 새로운 정찰 범위에 더해준다. 따라서 적이 벽에 부딪히면 새로운 정찰 지점을 설정하여 벽 밖에 나가지 않도록 설정한 것이다.

```
for (auto& wallObject : m_vpWallObjects) {
    for (auto& enemyObject : m_vpEnemyObjects) {
        if (wallObject->m_xmOOBB.Intersects(enemyObject->m_xmOOBB)) {
            XMVECTOR xmvNormal = XMVectorSet(wallObject->m_pxmf4WallPlanes.x,
                wallObject->m_pxmf4WallPlanes.y,
                wallObject->m_pxmf4WallPlanes.z, 0.0f);
            XMVECTOR xmvReflect = XMVector3Reflect(XMLoadFloat3(&enemyObject->GetLook()), xmvNormal);
            XMFLOAT3 xmf3Reflect = Vector3::XMVectorToFloat3(xmvReflect);
            enemyObject->Move(xmf3Reflect, 1.f);
            enemyObject->m_xmf3ScoutPoint = XMFLOAT3{ Vector3::Add(enemyObject->m_xmf3Position,
                Vector3::ScalarProduct(xmf3Reflect, 10.f)) };
        }
    }
}
```

앞서 기술하지 않은 플레이어와 적 포탄의 충돌 처리는 플레이어의 바디 바운딩 박스만으로는 크기가 작다 판단하여 메쉬를 상속받는 CTransparent 메쉬를 추가하여 바운딩 박스는 업데이트 하되 렌더링은 하지 않도록 구현하였다.

Ⅲ. 게임적 요소 및 조작법

■ 게임적 요소

- 플레이어는 적의 공격을 피해 최대한 많은 적들을 처치해야 한다. 먼 거리를 포물선을 이용한 포탄으로 적을 처치할 수 있는 재미가 있다. 또한 다양한 파티클을 총알에 적용하여 시각적 요소도 추가하였다. 스스로 움직이는 AI를 구현하여 혼자 게임을 하더라도 흥미를 떨어지지 않게 구성하였다.

■ 조작법

- 플레이어는 적의 공격을 피해 최대한 많은 적들을 처치해야 한다. 먼 거리를 포물선을 이용한 포탄으로 적을 처치할 수 있는 재미가 있다. 또한 다양한 파티클을 총알에 적용하여 시각적 요소도 추가하였다. 스스로 움직이는 AI를 구현하여 혼자 게임을 하더라도 흥미를 떨어지지 않게 구성하였다.

상하좌우키 : 플레이어 z 를 상하좌우로 이동
E : 플레이어 포신 우측으로 회전
Q : 플레이어 포신 좌측으로 회전
W : 플레이어 포신 위로 회전
S : 플레이어 포신 아래로 회전
R : 플레이어 리셋
T : 플레이어 직업 변경
Ctrl : 플레이어 총알 발사
ESCAPE : 게임 종료
마우스 좌클릭 : 플레이어 회전 및 카메라 회전
마우스 우클릭 : 카메라만 회전(둘러보기)

Ⅳ. 후기

- DirectX의 라이브러리에 대해 처음엔 막막했다. 너무 많은 함수들이 있고 머릿속에 바로 들어오지 않아 매우 힘들었다. 그러나 직접 과제를 진행하며 DirectX 라이브러리를 사용하다보니 자연스레 함수들이 익숙해졌고 알지 못했던 함수도 이런 함수가 있겠지 라는 생각에 찾아보면 실제로 이미 구현되어 있는 함수가 존재하였으며 네임 스페이스로 함수를 사용하니 매우 머릿속에 잘 들어왔다. 그러나 가장 재미있고 중요하게 생각했던 부분은 행동 트리를 이용해서 AI를 만드는 것이었다. 행동 트리를 그저 책으로만 봤고 직접 구현해보는 것은 처음이었기 때문에 구현하는데 시간이 오래걸리고 잘 안되는 부분도 많았다. 지금도 만족하지 못하는 수준이긴 하지만 그래도 꾸역꾸역 구현하고 나니 굉장히 재미있었고 객체에 생명을 불어준 것 같은 느낌에 매우 뿌듯했다. 다음에는 더 다양한 패턴을 이용해서 게임을 구현해 보고 싶었으며 아쉬웠던 점은 게임 프레임 워크를 잘 이해하며 어떤 부분에 뭐가 필요한지를 정확히 깨달아야 한다는 것이었다.