3DGP1 과제03

2023-06-23

목 차

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

한국공학대학교

Ⅰ. 과제에 대한 목표

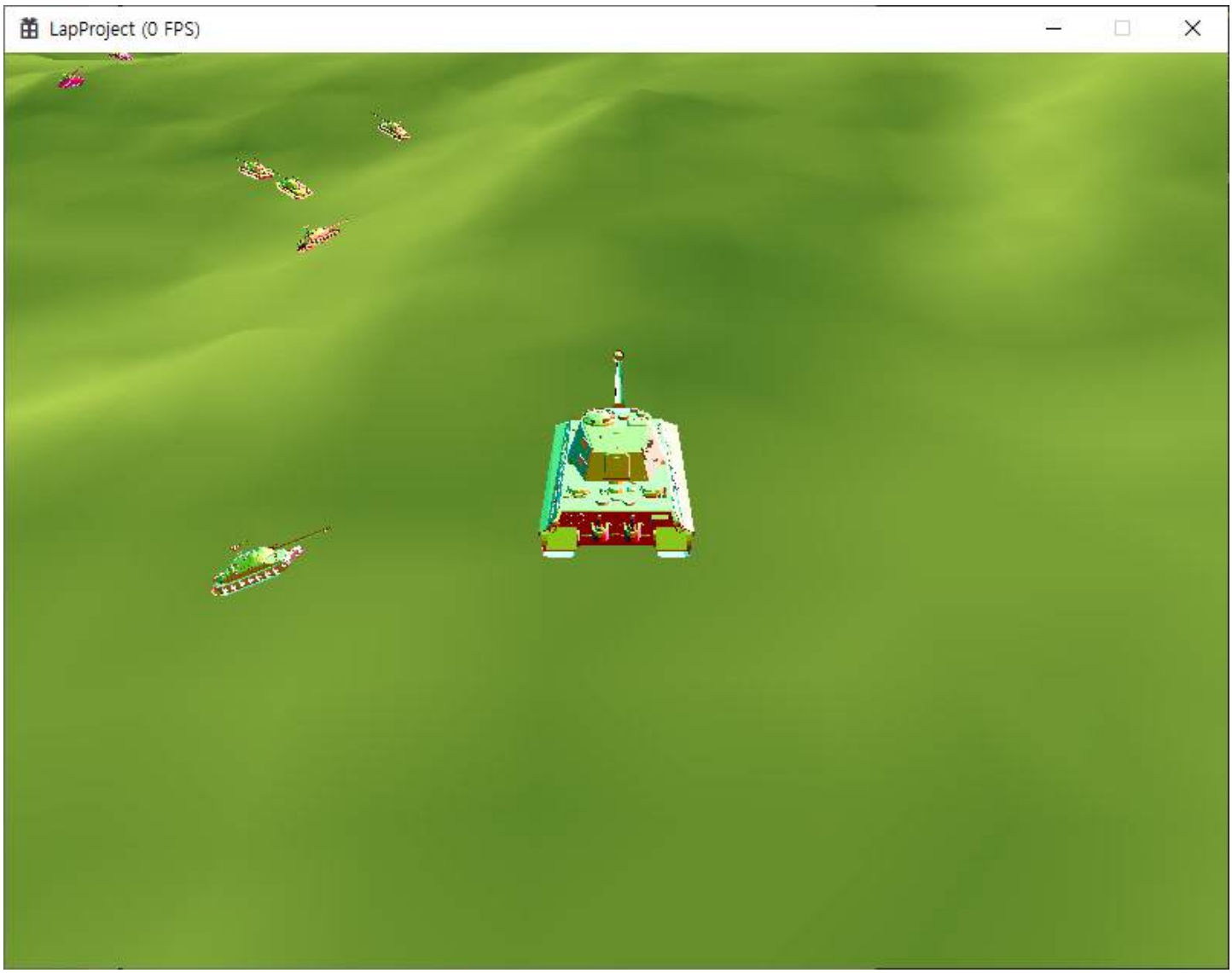
■ 과제 목표

- 3DGP1 과제03은 지형 로드와 지형 생성 및 오브젝트와의 충돌 처리를 중심으로 목표하였으며 쿼드 트리를 만들어서 충돌 처리 혹은 플레이어, 총알의 위치에 따라 오브젝트 개수를 최적화하는 것을 목표로 하였다.

Ⅱ. 실행 결과 및 구현 내용

■ 초기 화면

- 게임 시작 시 플레이어가 상단에서 아래로 중력의 영향을 받아 떨어지며 지형에 도착하였을 경우 지형 위에 안착한다. 그리고 주변에 랜덤한 위치에서 적들이 생성된다. 적은 지형의 기울기에 따라 기울어져 있으며 전방을 향해 총알을 발사한다.



■ 오브젝트 배치

- 플레이어 객체는 플레이어가 쉐이더를 가지고 있으며 적 객체는 쉐이더에 포함된다. 적이 개수가 900개이기 때문에 인스턴싱을 위해서 적은 인스턴싱 쉐이더를 사용하였으며 터레인은 프레임 워크에서 생성한 다음 인스턴스 쉐이더를 생성하여 객체들을 빌드할 때, 지형을 넘겨준다. 지형은 노멀에 대한 색상 처리를 따로 하지 않았기 때문에 일반 기본적인 쉐이더를 만들어서 포지션과 색상만 넘겨줬다.
- 플레이어 객체 또한 다음 오브젝트 생성 방식과 같이 지형의 높이와 너비, 길이를 받아줬다.

```
void CInstancingShader::BuildObjects(ComPtr<ID3D12Device> device, ComPtr<ID3D12GraphicsCommandList> cmdList, void* pContext)
{
    CHeightMapTerrain* terrain = (CHeightMapTerrain*)pContext;
    float terrainWidth = terrain->GetWidth(), terrainLength = terrain->GetLength();

    std::shared_ptr<CMesh> bodyMesh = std::make_shared<CMesh>(device, cmdList, "Models/IS7/Body.bin", false);
    std::shared_ptr<CMesh> turretMesh = std::make_shared<CMesh>(device, cmdList, "Models/IS7/Turret.bin", false);
    std::shared_ptr<CMesh> gunMesh = std::make_shared<CMesh>(device, cmdList, "Models/IS7/Gun.bin", false);
    std::shared_ptr<CMesh> bulletMesh = std::make_shared<CMesh>(device, cmdList, "Models/Bullet.bin", false);
    std::shared_ptr<CShader> shader = std::make_shared<CDiffusedShader>();
    shader->CreateShader(device, _rootSignature);
    shader->CreateShaderVariables(device, cmdList);

    _objectNum = 900;

    UINT xValue = sqrt(_objectNum);
    UINT yValue = sqrt(_objectNum);
    _objects.resize(_objectNum);

    Vec3 terrainPosition = terrain->GetPosition();
    for (int i = 0; i < xValue; ++i) {
        for (int j = 0; j < yValue; ++j) {
            _objects[i * xValue + j] = std::make_shared<CIS7Enemy>();
            std::shared_ptr<CGameObject> _turret = std::make_shared<CGameObject>();
            std::shared_ptr<CGameObject> _gun = std::make_shared<CGameObject>();
            _turret->SetChild(_gun);
            _objects[i * xValue + j]->SetChild(_turret);

            int randValueX = terrainWidth * urd(dreS);
            int randValueZ = terrainLength * urd(dreS);
            float fHeight = terrain->GetHeight(randValueX, randValueZ);
            _objects[i * xValue + j]->SetPosition(Vec3(randValueX, fHeight, randValueZ));

            Vec3 surfaceNormal = terrain->GetNormal(randValueX, randValueZ);
            Vec3 rotateAxis = Vector3::CrossProduct(Vec3(0.f, 1.f, 0.f), surfaceNormal);
            if (Vector3::IsZero(rotateAxis))
                rotateAxis = Vec3(0.f, 1.f, 0.f);
            Vec3 upVector = Vec3(0.f, 1.f, 0.f);
            float angle = acos(Vector3::DotProduct(Vec3(0.f, 1.f, 0.f), surfaceNormal));
            _objects[i * xValue + j]->Rotate(&upVector, uid(dreS));
            _objects[i * xValue + j]->Rotate(&rotateAxis, XMConvertToDegrees(angle));

            _objects[i * xValue + j]->_bullet->SetMesh(0, bulletMesh);
            _objects[i * xValue + j]->_bullet->SetShader(shader);
        }
    }
```

■ 쿼드트리

- 지형과 적 오브젝트는 움직이지 않고 고정되어 있는 상태이기 때문에 많은 갱신이 필요 없는 쿼드 트리가 적합하다고 판단하였다. 쿼드 트리를 통해서 지형을 나누고 플레이어의 위치를 받아서 플레이어의 일정한 거리 내에 있는 적 오브젝트만 충돌 처리를 하는 것이다.
- 다음은 쿼드 트리를 구성하는 방법이다.

■ 쿼드트리 구성

```
class CQuadTreeNode{
public:
    CQuadTreeNode(Point _position, float _size) : position(_position), size(_size) { }
    ~CQuadTreeNode() { }

public:
    Point position;
    float size;
    std::vector<std::shared_ptr<CQuadTreeNode>> children;
    std::vector<std::shared_ptr<CEnemy>> objects;
};
```

- 우선 쿼드트리 노드는 노드의 포지션, 사이즈, 오브젝트를 받으며 4개의 쿼드트리 노드를 가진다.

```
void InsertRecursive(std::shared_ptr<CQuadTreeNode> node, std::shared_ptr<CEnemy> object) {
    // 현재 노드의 사이즈가 1 이하이면 종료

    if (node->size <= 100.0f) {
        node->objects.push_back(object);
        return;
    }

    if (node->position._x - node->size / 2 > object->GetPosition().x || node->position._x + node->size / 2 < object->GetPosition().x ||
        node->position._z - node->size / 2 > object->GetPosition().z || node->position._z + node->size / 2 < object->GetPosition().z)
        return;

    // 현재 노드에 자식 노드가 없으면 생성
    if (node->children.empty()) {
        float childSize = node->size / 2.0f;
        float childX = node->position._x - childSize / 2.0f;
        float childY = node->position._z - childSize / 2.0f;

        node->children.emplace_back(std::make_shared<CQuadTreeNode>(Point(childX, childY), childSize));
        node->children.emplace_back(std::make_shared<CQuadTreeNode>(Point(childX + childSize, childY), childSize));
        node->children.emplace_back(std::make_shared<CQuadTreeNode>(Point(childX, childY + childSize), childSize));
        node->children.emplace_back(std::make_shared<CQuadTreeNode>(Point(childX + childSize, childY + childSize), childSize));
    }

    // 자식 노드에 포인트 삽입
    for (auto& child : node->children) {
        InsertRecursive(child, object);
    }
}
```

- 재귀 함수를 돌며 노드의 사이즈가 100이 될 때까지 자식 노드를 생성한다. 결국 플레이어의 주변 100 범위 내에 있는 적들을 판별하기 위한 것이다.

```
std::vector<std::shared_ptr<CEnemy>> SearchQuadTree(float x, float z) {
    return SearchObject(x, z, root);
}

std::vector<std::shared_ptr<CEnemy>> SearchObject(float x, float z, std::shared_ptr<CQuadTreeNode> node) {
    std::vector<std::shared_ptr<CEnemy>> objects;

    // 현재 노드의 사이즈가 1 이하이면 현재 노드의 오브젝트들을 반환
    if (node->size <= 100.0f) {
        return node->objects;
    }

    // 주어진 포지션 (x, z)가 현재 노드의 영역을 벗어나면 빈 벡터 반환
    if (x < node->position._x - node->size / 2.0f ||
        x > node->position._x + node->size / 2.0f ||
        z < node->position._z - node->size / 2.0f ||
        z > node->position._z + node->size / 2.0f) {
        return objects;
    }

    // 자식 노드에 대해 재귀적으로 호출하여 오브젝트들을 가져옴
    for (auto& child : node->children) {
        std::vector<std::shared_ptr<CEnemy>> childObjects = SearchObject(x, z, child);
        objects.insert(objects.end(), childObjects.begin(), childObjects.end());
    }

    return objects;
}
```

- 마지막으로 해당 좌표 범위를 값으로 받아 해당하는 노드의 오브젝트들을 반환하게 된다. 따라서 플레이어의 해당 위치 좌표를 넘겨주게 되면 해당 노드의 오브젝트를 반환받을 수 있을 것이다.

■ 쿼드트리 사용

```
void CScene::CheckObjectByPlayerCollisions()
{
    std::vector<std::shared_ptr<CEnemy>> objects = _quadTree->SearchQuadTree(_player->GetPosition().x, _player->GetPosition().z);

    for (auto& object : objects) {
        if (object->_boundingBox.Intersects(_player->_boundingBox) && !_player->_death) {
            object->_death = true;
        }
    }
}
```

- 다음과 같이 플레이어의 위치를 넘겨주고 해당하는 노드의 오브젝트들(적 오브젝트)을 vector로 반환받게 되면 플레이어와 충돌처리를 할 적 객체들은 개수가 적어진다. 만약 적 객체가 10000개가 골고루 맵 전반에 펼쳐 있다면 쿼드트리를 사용하지 않는다면 10000개의 모든 적과 플레이어를 충돌처리를 해야 한다. 그러나 다음과 같이 쿼드트리를 사용함으로써 대략 100개 이내의 충돌처리 대상으로 변경할 수 있다.
- 이와 같이 플레이어와 적의 충돌처리 뿐만 아니라 총알과 적의 충돌처리 등 모든 충돌처리에 쿼드트리를 사용하면 충돌처리를 최적화할 수 있을 것이다.

Ⅲ. 게임적 요소 및 조작법

■ 게임적 요소

- 플레이어가 넓은 지형을 돌아다닐 수 있으며 넓은 지역에 랜덤한 위치에서 적이 생성된다. 적은 전방으로 총알을 발사하기 때문에 이 총알을 피하면서 최대한 오래 살아남는 것이 목표다.

■ 조작법

마우스 클릭 회전 : 플레이어 터렛과 포신, 카메라를 회전
W : 플레이어 앞으로 전진
S : 플레이어 뒤로 후진
A : 플레이어의 전체를 왼쪽으로 회전
D : 플레이어의 전체를 오른쪽으로 회전
Ctrl : 플레이어 총알 발사
ESCAPE : 게임 종료

Ⅳ. 후기

- 지형을 처음으로 엔진을 사용하지 않고 만들어 봤는데 직접 만든 지형에 충돌처리를 하다 보니 굉장히 흥미로웠다. 또한 쿼드트리를 구성하여 만들었는데 오류도 많았고 아직 잡지 못한 버그도 있었다. 그러나 충돌처리를 매우 짧은 시간으로 줄였다는 것에 굉장히 뿌듯하였다. 다음 목표로는 옥트리를 직접 구성해서 만들어보는 것이고 다른 부분에서도 최적화할 수 있는 것이 있다면 도전해보고 싶다.