3DGP1 과제02

2023-06-16

목 차

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

한국공학대학교

Ⅰ. 과제에 대한 목표

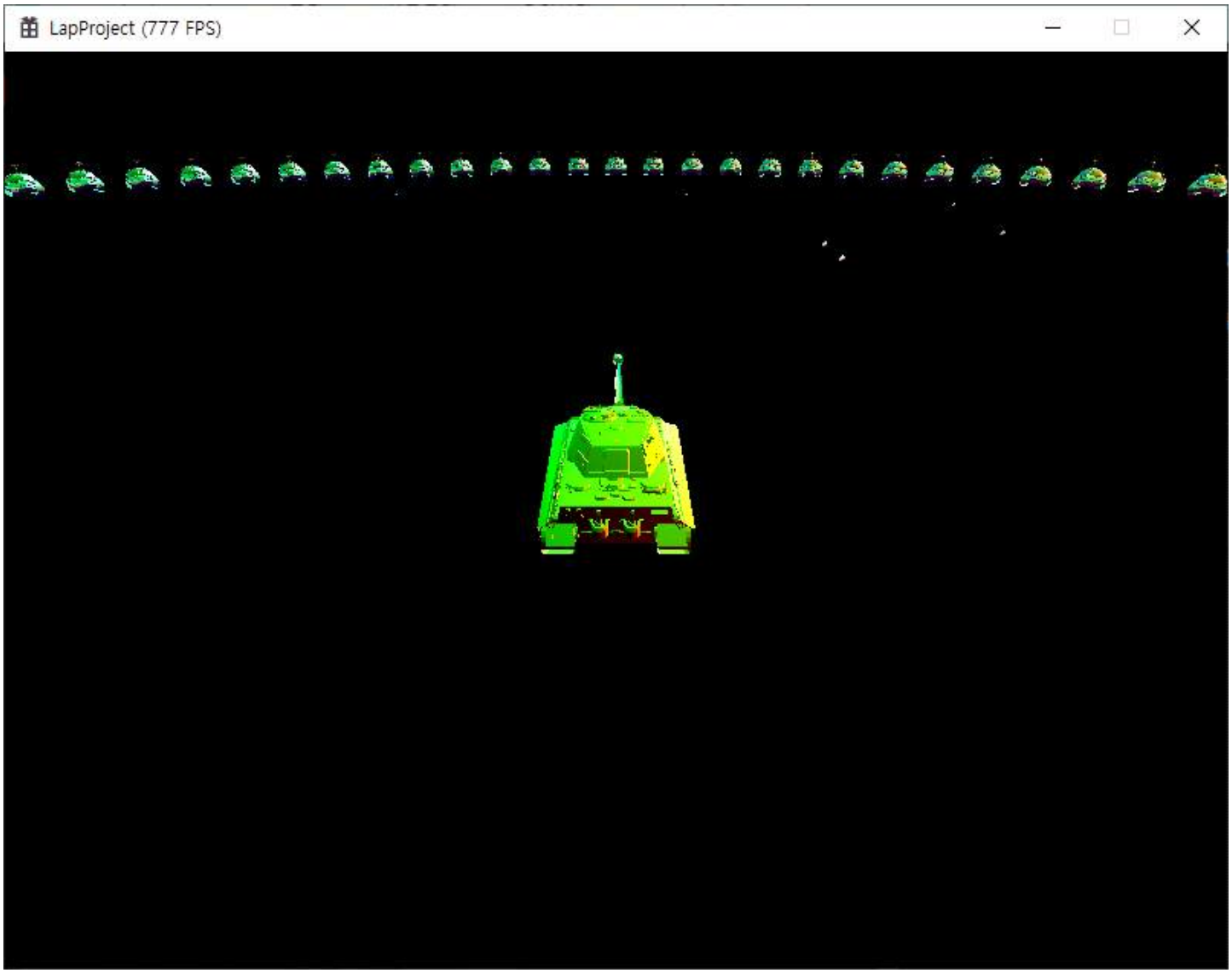
■ 과제 목표

- 3DGP1 과제02는 DirectX12의 디바이스 및 렌더링 파이프라인을 이해하며 모델을 직접 원하는 포맷으로 로드하여 메쉬 인스턴싱을 통한 배치 처리를 생각하며 간단한 게임을 만드는 것을 목표로 하였다.
또한 공간 분할을 통해 쿼드 트리 및 옥트리를 이용한 충돌 처리 최적화를 목표로 하였다.

Ⅱ. 실행 결과 및 구현 내용

■ 초기 화면

- 게임 시작 시 탱크 플레이어가 중앙에 위치하며 적 탱크 객체 100개가 플레이어의 주변에 위치 한다.
주변을 둘러싼 적 탱크의 공격을 피하는 일종의 탄막 게임으로 진행되며 플레이어는 적의 총알을 맞게 되면 일시적으로 파편이 분리되어 형체 없이 돌아다닐 수 있지만 공격은 할 수 없다. 시간이 지나면 다시 리스폰 된다.
탱크는 각각 몸통, 터렛, 포신으로 이루어져 있으며 몸통은 터렛을 자식으로 가지고 있으며 터렛은 포신을 자식으로 가지고 있다. 마우스 회전 이동을 통해 몸통이 아닌 터렛과 포신만 이동하도록 구현하였으며 카메라는 터렛을 따라 움직이게 된다.



■ 컴파일 쉐이더

- 쉐이더의 컴파일은 런타임 컴파일이 아닌 응용프로그램이 미리 컴파일하여 바이너리 형태로 읽을 수 있도록 만들었다. 이 결과 쉐이더의 오류를 컴파일 타임에 해결할 수 있었으며 속도가 훨씬 빨랐다. 또한 정점 쉐이더와 픽셀 쉐이더를 구분하여 서로 다른 .hlsl파일로 관리할 수 있었다.

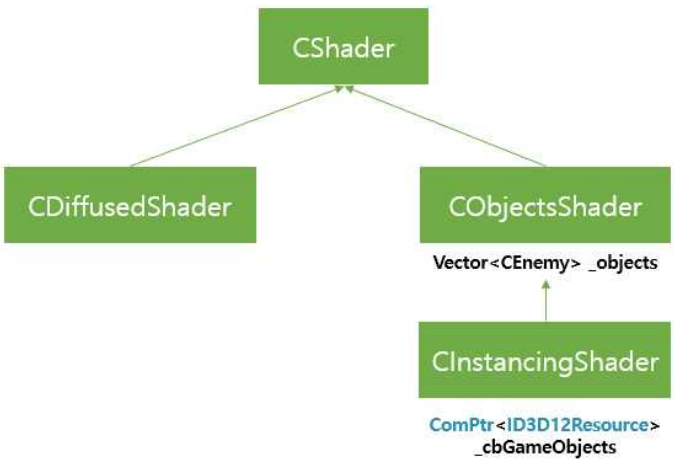
```
D3D12_SHADER_BYTECODE CShader::ReadCompiledShaderFile(WCHAR* fileName, ComPtr<ID3DBlob>& shaderBlob)
{
    std::ifstream in{ fileName, std::ios::binary | std::ios::ate };
    UINT readBytes = (UINT)in.tellg();
    BYTE* byteCode = new BYTE[readBytes];
    in.seekg(0);
    in.read((char*)byteCode, readBytes);

    D3D12_SHADER_BYTECODE d3dByteCode;
    if (shaderBlob) {
        D3DCreateBlob(readBytes, shaderBlob.GetAddressOf());
        memcpy(shaderBlob->GetBufferPointer(), byteCode, readBytes);
        d3dByteCode.BytecodeLength = shaderBlob->GetBufferSize();
        d3dByteCode.pShaderBytecode = shaderBlob->GetBufferPointer();
    }
    else {
        d3dByteCode.BytecodeLength = readBytes;
        d3dByteCode.pShaderBytecode = byteCode;
    }

    return d3dByteCode;
}
```

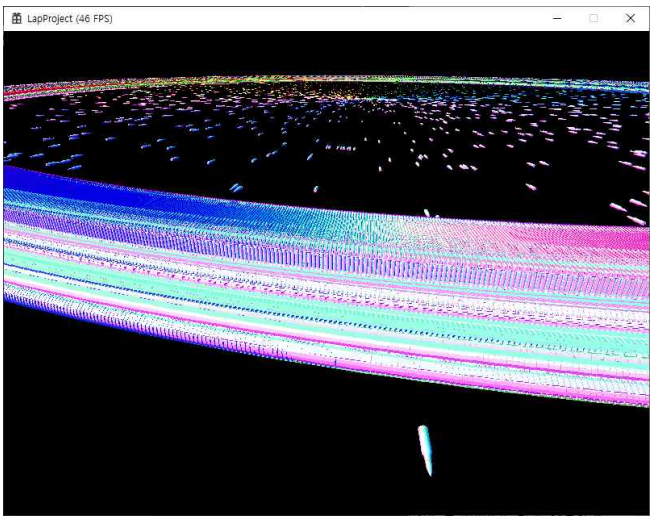
■ 모델 로드

- 유니티를 이용하여 모델을 바이너리 파일 형태로 원하는 포맷 형식에 맞춰 포팅할 수 있었고 LoadMeshFromFile을 통해 바이너리 파일을 쉽게 읽어 .txt파일보다 훨씬 빠른 로드가 가능했다.

■ 프레임 워크

- 쉐이더는 다음과 같이 오브젝트를 가지는 쉐이더와 오브젝트를 가지지 않는 쉐이더로 나눴다. 만약 같은 메쉬를 가지는 객체가 많을 경우 인스턴싱을 위한 인스턴싱 쉐이더를 오브젝트 쉐이더에서 상속받아 인스턴싱을 사용할 수 있도록 설정하였다.





적 탱크를 인스턴싱한 결과 BuildObjects에서 20000개가 넘는 정점을 가진 적 탱크 메쉬를 5000개까지 늘린 결과 프레임 레이트를 60이상으로 유지할 수 있었다. 로드 시간 또한 바이너리 파일로 만들어 순식간에 로드가 되었다. CInstancingShader는 입력 조립기에 변환 행렬 정보와 죽고 흐른 시간의 개별 정보들을 인스턴스로 넘겨주었으며 위치 값과 노말 값, uv값 등이 정점 버퍼로 넘어가게 된다. 인스턴싱을 사용하지 않으며 쉐이더가 아닌 씬이 객체들을 가지고 있을 경우에는 입력 조립기에 위치 값과 노말 값, uv값만 넘어가게 된다.

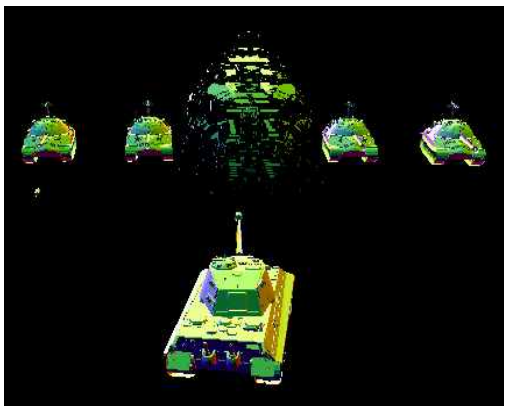
```cpp
D3D12_ROOT_PARAMETER rootParam[3];
rootParam[0].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
rootParam[0].Constants.Num32BitValues = 4;
rootParam[0].Constants.ShaderRegister = 0;
rootParam[0].Constants.RegisterSpace = 0;
rootParam[0].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;

rootParam[1].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
rootParam[1].Constants.Num32BitValues = 17;
rootParam[1].Constants.ShaderRegister = 1;
rootParam[1].Constants.RegisterSpace = 0;
rootParam[1].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;

rootParam[2].ParameterType = D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS;
rootParam[2].Constants.Num32BitValues = 35;
rootParam[2].Constants.ShaderRegister = 2;
rootParam[2].Constants.RegisterSpace = 0;
rootParam[2].ShaderVisibility = D3D12_SHADER_VISIBILITY_ALL;
```

```cpp
cbuffer cbFrameworkInfo : register(b0)
{
    float currentTime;
    float elapsedTime;
    float2 cursorPos;
};

cbuffer cbGameObjectInfo : register(b1)
{
    matrix world : packoffset(c0);
    float timeAfterDeath : packoffset(c4);
};

cbuffer cbCameraInfo : register(b2)
{
    matrix view : packoffset(c0);
    matrix projection : packoffset(c4);
    float3 cameraPos : packoffset(c8);
};
```

- 위와 같이 자주 변하는 값은 루트 상수로 넘겨주었다. 또한 넘겨지는 값이 많지 않아 루트 서술자를 사용하지 않았다.

■ 쉐이더 프로그래밍

- 간단한 쉐이더를 활용한 프로그래밍이다. 적 또는 플레이어가 탄환에 맞게 맞은 시점으로부터의 흐른 시간을 인스턴싱을 통하여 외부로부터 값을 전달받아 적의 노말 값에 곱해준다. 각 버텍스의 정점을 노말 방향으로 death time만큼 이동시켜 탱크가 터지는 효과를 주었다.



```cpp
float deathTime = input.timeAfterDeath;

float3 newPosition;
newPosition.xz = input.normal.xz * 1.5f * deathTime;
newPosition.y = input.normal.y * 3.f * deathTime - deathTime * deathTime * 0.5;

float t = input.position.y + newPosition.y;

newPosition.x = clamp(newPosition.x, -10.f, 10.f);
newPosition.z = clamp(newPosition.z, -10.f, 10.f);

output.positionW = mul(float4(input.position + newPosition, 1.0f), input.transform).xyz;

if(output.positionW.y < 0)
    output.positionW = 0.f;
```

■ 자료구조

```cpp
std::shared_ptr<COctreeNode> CScene::BuildOctree(Vec3 center, float halfWidth, int depthLimit)
{
    if (depthLimit < 0)
        return nullptr;

    std::shared_ptr<COctreeNode> octNode = std::make_shared<COctreeNode>();
    BoundingBox boundingBox = octNode->GetBoundingBox();
    octNode->SetCenter(center);
    octNode->SetRadius(halfWidth);

    Vec3 offset;
    Vec3 childCenter;
    float step = halfWidth * 0.5f;

    for (UINT i = 0; i < CHILD_NODE_COUNT; ++i) {
        offset.x = (i & 1) ? step : -step;
        offset.y = (i & 4) ? step : -step;
        offset.z = (i & 2) ? step : -step;
        childCenter.x = offset.x + center.x;
        childCenter.y = offset.y + center.y;
        childCenter.z = offset.z + center.z;
        octNode->AddChildNode(BuildOctree(childCenter, step, depthLimit - 1));
    }

    return octNode;
}
```

- 객체의 충돌처리 최적화를 위하여 간단하게 옥트리 형태를 구현했지만 적이 많지 않은 점과 지형이 없다는 점에서 실제로 게임에 적용하지는 않았다. 그러나 추후에 객체가 많아진다면 쿼드 트리 혹은 옥트리를 고려해야 할 것이다.

Ⅲ. 게임적 요소 및 조작법

■ 게임적 요소

- 플레이어는 사방에서 날라오는 적의 공격을 피해야 한다. 총알을 통해 먼 거리의 적을 물리칠 수 있으며 이 때 적의 파편이 터져 나오는 쉐이더 효과를 통해서 타격감을 주었다. 일종의 탄막 게임으로 자신의 컨트롤의 한계를 체험할 수 있다. 플레이어의 회전을 마우스와 키보드로 나눠 마우스는 터렛과 포신을 빠른 속도로 회전시키며, 키보드는 바디 전체를 느린 속도로 움직이도록 구현하여 게임의 흥미를 추가하였다.

■ 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

- WinApi만 사용하였을 때와 DirectX12를 사용하였을 때의 차이는 확연하게 갈렸다. 객체의 인스턴싱과 GPU를 적극적으로 활용할 수 있어 게임의 프레임 레이트를 높일 수 있었다. 가장 좋았던 점은 GPU와 직접 소프트웨어를 통해 통신할 수 있다는 점이었다. 그저 그래픽 카드를 응용 프로그램들이 정해진 대로만 사용하다가 직접 사용해보니 새로운 경험이었고, 배치처리와 GPU를 활용한 쉐이더 최적화를 더 배우고 싶어졌다. 또한 DirectX12의 가장 중점이라고 하는 멀티 쓰레드 프로그래밍을 추가하고 싶었다. DirectX12에는 CPU와 GPU를 프로그래머가 직접 동기화 할 수 있도록 설계 되었기 때문에 가능한 가장 빠른 프레임 레이트를 끌어 올릴 수 있도록 제작하고 싶었다. 직접 모델을 로드하여 나의 게임에 적용 시켜보니 매우 재미있었지만 부족한 점으로 루트 테이블과 서술자 등을 적극적으로 사용해야 겠다는 생각이 들었다.