[3D 게임 2 과제 01 보고서]

[게임공학과]
최해성(2018180044)

-과제에 대한 목표-

저의 이번 과제에 대한 목표는 다음과 같습니다.

1. HeightMap을 이용해 Terrain 생성 후 텍스쳐 매핑으로 지형을 표현한다.

2. 플레이어는 헬기 모델을 사용하며, 텍스쳐 매핑으로 헬기의 재질을 표현한다.

3. 적은 헬기 모델을 사용하며, 텍스쳐 매핑으로 헬기의 재질을 표현한다.

4. 적은 기본적으로 랜덤하게 주변을 배회하며 지형에 파묻히거나 맵 밖으로 나가지 못하고 플레이어가 근처에 있으면 플레이어를 방향을 바라보며 다가온다. (이때, 플레이어한테는 겹치기 전에 멈추게 한다.)

5. 지형에 나무, 꽃, 풀을 ObjectMap을 이용해 배치하고 빌보드 객체로 표현한다.(이때, 기하쉐이더를 이용해 각 빌보드 객체를 생성한다.)

6. 지형에 물을 텍스쳐를 이용해 표현한다.

7. 플레이어는 총알을 쏠 수 있으며, 총알이 적과 충돌하는 경우 적은 스트라이트 애니메이션으로 폭발하는 듯한 효과를 보여준다.

8. 스카이박스를 적용한다.

-과제 실행 환경-

과제의 실행 환경은 다음과 같습니다.

1. Visual Studio 2019 최신버전

3. Release/x64 모드

4. C++ 언어 표준: 최신 C++ 초안의 기능

-조작법-

W/A/S/D – 플레이어를 앞/왼쪽/뒤/오른쪽으로 이동시킵니다.

Ctrl – 총알을 발사합니다.

Q/E – 플레이어가 위/아래로 이동합니다.

마우스 왼클릭 후 움직이면 플레이어가 회전합니다.

-구현방법-

1. HeightMap을 이용한 Terrain 생성 및 텍스쳐 입히기

Terrain은 CScene에서 생성되고 관리하게 구현했습니다. CScene::BuildObjects()에서 Terrain을 CHeightMapTerrain 클래스를 이용해 객체를 생성하는데, CHeightMapTerrain 클래스는 지형의 높이 정보가 담긴 HeightMap.raw 파일을 읽어 지형 Mesh를 생성하고 지형의 기본색을 나타낼 Base_Texture.dds라는 DDS 텍스쳐파일, 지형의 디테일을 표현할 Detail_Texture_7.dds라는 DDS 텍스쳐 파일, Detail Texture를 어디에 얼만큼 적용할 것인지 표현한 HeightMap(Alpha).dds라는 DDS 텍스쳐파일을 불러옵니다. 또한 CTerrainShader 클래스를 생성해 그래픽스 파이프라인 상태 객체를 생성하고 텍스쳐를 저장할 디스크립터 힙을 생성해 불러온 텍스쳐를 SRV로 생성합니다. 이렇게 생성한 텍스쳐는 렌더링 과정에서 루크 시그니쳐 디스크립터 테이블로 연결하고 픽셀쉐이더에서 연결된 Base_Texture과 Detail_Texture를 uv 좌표로 샘플링하고 HeightMap(Alpha) 텍스쳐에서 샘플링한 알파값을 이용해 선형보간해 색상을 결정하는 방식으로 텍스쳐를 입힌 Terrain을 생성해 렌더링했습니다.

2-0. 플레이어 오브젝트 생성 & 텍스쳐 입히기

플레이어 오브젝트는 CGameFramework::BuildObjects()에서 생성이 되는데 모델은 교수님이 제공해주신 Mi24.bin 파일을 이용해 불러와서 사용했습니다. 이때, Mi24 모델은 2개의 텍스쳐를 사용해 텍스쳐 매핑을 하기 때문에 CModeledTexturedShader라는 이름의 그래픽스 파이프라인 상태 객체를 생성하는 클래스를 생성하고 2개의 SRV를 저장할 디스크립터 힙을 생성해 모델을 불러오는 과정에서 해당 디스크립터 힙에 필요한 텍스쳐를 SRV로 생성합니다. 플레이어 모델은 렌더링과정에서 Texture2D 배열을 사용해 텍스쳐를 연결하고 사용할 텍스쳐들을 UINT 자료형으로 표현한 gnTexturesMask에 따라 픽셀쉐이더에서 텍스쳐를 샘플링해 텍스쳐를 입힌 Mi24 모델로 플레이어를 렌더링하였습니다.

2-1. 플레이어 이동/회전 & 지형에 파묻히지 않게 하기

플레이어의 이동은 매 프레임마다 호출되는 GameFramework::ProcessInput()에서 입력키를 검사해 W/A/S/D/Q/E에 따라 플레이어를 이전 프레임이 불린 시간과 현재 프레임이 불린 시간의 차이에 일정 값을 곱해 플레이어의 위치를 이동시켜줍니다. 플레이어의 회전은 윈도우가 클릭된 후 커서가 움직인 정도를 변수에 저장해 변수 값에 따라 플레이어를 회전시켜주었습니다. 플레이어 객체가 생성될 때, CScene 클래스에서 생성된 Terrain 객체의 포인터를 가져와 전달해주어 플레이어가 Terrain 객체에 대한 포인터를 가지게 해 플레이어의 위치를 업데이트할 때, 플레이어의 (x, z) 좌표에 해당하는 Terrain의 높이를 가져와 플레이어가 Terrain의 높이 보다 작은 y 값을 가지고 있으면 y값을 Terrain의 높이로 바꿔주어 지형보다 더 아래로 내려갈 수 없게 해주고, 추가로 최대 높이를 정해 일정 높이보

다 올라갈 수 없게 해주었습니다. x, z 좌표의 경우에도 Terrain의 너비를 가져와 플레이어 객체의 위치가 Terrain을 벗어나게된다면 일정한 CORRECTION 값을 더하거나 빼게 해주어 지형에서 나가지 않게 구현해주었습니다.

3-1. 적 생성 및 텍스쳐 입히기

적 객체들의 경우 CObjectShader에서 120개가 생성되고, 관리하는데 CObjectShader는 CScene::BuildObjects()에서 싱글톤 객체로 호출되어 CreateShader() 함수를 호출해 파이프라인 상태 객체를 생성하고 BuildObjects() 함수를 호출해 실제 적 객체들을 생성합니다.
CObjectShader::BuildObjects()에서는 적 객체를 생성하고 배치하는데 적 객체는 교수님이 제공해주신 SuperCobra.bin, Gunship.bin 모델을 사용하게 했습니다. SuperCobra의 경우 17개의 텍스쳐를 사용하고 Gunship의 경우 2개의 텍스쳐를 사용하기 때문에 디스크립터 힙을 SRV가 19개 들어갈 수 있게 생성해줍니다. 그리고 적 객체들이 하나의 모델을 공유하도록 하기위해서 이 디스크립터 힙을 사용해 SuperCobra 모델 객체를 불러와 std::shared_ptr<CGameObject> 자료형의 pSuperCobraObject라는 이름으로 저장하고, Gunship 모델 객체를 불러와 std::shared_ptr<CGameObject> 자료형의 pGunshipObject라는 이름으로 저장합니다. 120개의 적 객체를 생성하기 위해 m_ppObjects라는 이름을 가진 std::vector<CEnemy*>를 std::vector<>::resize()를 이용해 120개의 CEnemy 클래스를 담을 수 있게 공간을 잡아주고 LabProject 8-0-1에서 객체를 랜덤하게 배치하는 방법을 응용해 Terrain의 범위 안에서 랜덤하게 SuperCobra 모델 혹은 GunShip 모델을 Child를 가지게 구현했습니다. 또한 객객체 생성할 때, CScene에 생성된 Terrain의 포인터를 적 객체를 관리하는 CEnemy 클래스가 가지고 있게 설정해주었습니다.

3-2. 적 이동 및 AI

적 객체는 매프레임마다 CObjectShader::AnimateObjects()에서 적 객체들을 담고있는 std::vector<CEnemy*> 를 for문을 사용해 순회하면서 모든 CEnemy*의 Update() 함수를 호출해 위치를 업데이트해주었습니다. CEnemy::Update()에서는 먼저, UpdateDirection()이라는 함수를 호출해 적 객체의 진행방향을 업데이트해줍니다. 이때, 적 객체가 플레이어와 근접해 객체가 가지고 있는 m_pPlayer라는 포인터에 NULL이 아니라 실제 객체에 대한 포인터를 가지고 있다면 플레이어의 위치에서 현재 자신의 위치를 빼서 플레이어를 향하는 벡터를 구해 진행방향을 설정해주었습니다. 단, 플레이어를 향하는 벡터의 크기가 100.0f 보다 작으면 방향의 크기를 x, y, z를 전부 0.0f로 설정해 일정 이상 플레이어에게 다가가지 못하게 해주었습니다. 플레이어와 근접한 적이 있지 않아 m_pPlayer가 NULL일 경우 먼저, 플레이어의 위치와 현재 자신의 위치를 빼 얻은 방향 벡터의 크기를 구해 300.0f 보다 작으면 플레이어와 충분히 가깝다고 판단해 SetPlayer() 함수를 통해 CEnemy 클래스 내부의 m_pPlayer라는 포인터에 플레이어 객체의 포인터를 설정해줍니다. 만약 플레이어와 충분히 가깝지 않다면 적 객체는 현재 자신의 위치 근처에서 방황하게 됩니다. 방황하는 움직임의 경우

fWanderingTime이라는 현재 객체가 방황하고 있는 시간을 담는 float형 변수를 선언해 Update() 함수가 불릴 때마다 넘어온 프레임시간을 fWanderingTime에 더해줍니다. 이 fWanderingTime이 5.0f보다 크다면(5초가 지나면) 적 객체의 이동방향을 표시하는 XMFLOAT3 변수의 x, y, z 값을 랜덤하게 -1.0f ~ 1.0f을 가지게 하고 Normalize를 해 랜덤한 방향을 가지게 해주는 방식으로 방황하는 AI를 구현하였습니다. UpdateDirection() 함수를 빠져나온 뒤, 객체가 진행방향을 가진다면(적 객체의 방향을 나타내는 XMFLOAT3형 변수 direction의 크기가 0.0f 보다 크다면) 현재 위치에서 진행방향을 더한 위치를 바라보게 설정해주고, 객체가 진행방향을 가지지 않더라도 플레이어를 타겟팅해 m_pPlayer에 플레이어에 대한 포인터를 가지고 있다면 플레이어의 위치를 바라보게 설정해주었습니다.(이때, 객체의 UpVector는 항상 y축 1.0f를 향하게 설정) 이후 진행방향을 나타내는 XMFLOAT3형 변수 direction에 속도 * 프레임시간을 곱해 변위를 구하고 현재 위치에 변위를 더해 최종 위치를 구해준 뒤 지형에 파묻히지 않게 플레이어가 지형에 파묻히지 않게 해준 방법과 동일하게 적용해주었습니다. 이렇게 위치를 최종적으로 구하게 되면 UpdateBoundingBox() 함수를 호출해 객체의 바운딩박스를 갱신 해줍니다.

4. 지형에 기하쉐이더를 이용해 나무, 꽃, 풀 표현

나무, 꽃, 풀 표현 같은 경우에는 ObjectMap 파일, 빌보드 기법, 기하쉐이더를 이용해 구현하였습니다. 나무, 꽃 풀들은 싱글톤으로 정의된 CBillboardObjectsShader 클래스로 생성하고, 렌더링합니다.

(본인 컴퓨터 기준) 나무가 많아도 정상적으로 165FPS가 출력됨을 확인하였습니다.

=> 3번째 실행화면 캡처에서 확인가능

4-1. CreateShader()

CreateShader()에서는 다른 쉐이더 클래스와 같이 파이프라인 상태 객체를 생성합니다. 이때, 기하쉐이더를 사용하기 위해서 D3D12_GRAPHICS_PIPELINE_STATE_DESC::GS에 기하쉐이더를 연결해주고 점을 입력조립기로 보낼 것이기 때문에
D3D12_GRAPHICS_PIPELINE_STATE_DESC::PrimitiveTopologyType을
D3D12_PRIMITIVE_TOPOLOGY_TYPE_POINT로 설정해줍니다. 또한 빌보드 텍스쳐들의 경우 알파값으로 투명한 부분이 표현이 잘 되어있기 때문에 이를 이용하기 위해서 블렌드 상태 객체에서 블렌딩 옵션을 키고, 텍스쳐의 알파값을 이용해 출력병합단계에서 블렌딩이 되도록 설정을 해주었습니다.

4-1. BuildObjects()

BuildObjects()에서는 먼저, 나무, 꽃, 풀들을 표현할 텍스쳐들을 pTexture라는 이름의 변수에 불러옵니다. 총 7개의 텍스쳐를 불러오기 때문에 7개의 SRV가 들어갈 수 있는 디스크립터 힙을 만들고 디스크립터 힙에 SRV를 생성해줍니다. 이후 교수님께서 구현하신 것과 같이 ObjectsMap.raw 파일을 읽

어와 픽셀의 색상에 따라 픽셀의 색상이 나무 or 꽃 or 풀을 표현해야하는 색상이면 해당 픽셀 좌표에 해당하는 Terrain의 위치와 Terrain 높이를 구해 XMFLOAT3으로 저장한 뒤 같은 텍스쳐를 사용하는 좌표들끼리 std::vector<XMFLOAT3>로 모아서 저장합니다.(7개의 std::vector<XMFLOAT3>에 나눠서 저장합니다.) 저장한 뒤, 7개의 std::vector<XMFLOAT3>와 어떤 타입의 텍스쳐를 사용하는 지를 표시하는 UINT형 변수 nType으로 CBillBoardPointMesh 클래스를 생성해 std::vector<std::shared_ptr<CMesh>> 컨테이너에 저장합니다.

CBillBoardPointMesh는 std::vector<XMFLOAT3>와 nType을 전달 받아 nType에 따라 텍스쳐에 맞게 생성할 빌보드의 사이즈를 결정하고 std::vector<XMFLOAT3>에 담겨있는 XMFLOAT3 값들로 정점 버퍼를 만들어 저장합니다.

4-2. Render()

빌보드 객체들을 렌더링할 때, 텍스쳐를 한번만 설정해주기 위해성 나무, 꽃, 풀의 7개의 텍스쳐들을 텍스쳐 배열로 t0 ~ t6까지 서술자 테이블로 연결해주고 BuildObjects()에서 생성한 7개의 CBillBoardPointMesh를 몇번째 텍스쳐를 사용하는 메쉬인지 표현하는 UINT형 변수 nType을 Root32BitConstant로 연결해준 뒤 CBillBoardPointMesh에 저장된 정점버퍼를 파이프라인의 입력 슬롯에 바인딩해주고 정점버퍼에 저장된 점의 개수만큼 렌더링을 하게 됩니다. 입력조립기로 넘어간 빌보드 객체들의 위치 정보, 빌보드 사이즈 정보가 담긴 정점 정보를 정점 쉐이더는 그 정보 그대로 기하쉐이더로 넘겨주고 기하쉐이더에서는 넘어온 정점의 정보를 이용해 카메라를 바라보는 빌보드를 표현하는 위치정보와 uv 좌표를 가진 정점 4개를 생성해 픽셀쉐이더로 넘겨줍니다. 마지막으로 픽셀쉐이더에서는 연결되어서 넘어온 nType에 따라 텍스쳐를 선택해 샘플링을 해서 픽셀 색상을 결정해 렌더링합니다.

5. 스카이 박스

스카이박스는 CScene에서 생성, 렌더링됩니다. 스카이박스를 표현하는 CSkyBox 클래스는 생성자에서 스카이박스를 구성할 6개의 면에 해당하는 텍스쳐들을 불러오고 스카이박스를 렌더링을 하는데 사용할 CSkyBoxShader를 생성해 그래픽스 파이프라인 상태 객체, 디스크립터 힙, SRV를 만들어줍니다. 단, 스카이박스를 렌더링하는데 사용하는 그래픽스 파이프라인 상태 객체를 생성할 때, 깊이 검사를 하지 않게 설정을 해줍니다. 또한 스카이박스 텍스쳐를 보여줄 카메라를 충분히 덮는 큐브모양을 구성할 6개의 사각형 메쉬를 만들어 줍니다. 스카스카이박 렌더링될 때, 카메라의 위치 정보를 가져와 xmf3CameraPos라는 XMFLOAT3 변수에 저장하고 스카이박스의 위치를 카메라의 위치와 같게 설정을 해주어 항상 카메라를 스카이박스가 덮고 있는 형태를 띄게 만들어줍니다. 이를 제외하고는 렌더링하는 부분에서 다른 객체들과 같이 텍스쳐를 쉐이더에 서술자 테이블을 이용해 연결해주고 정점버퍼를 연결해주는 등의 과정을 거쳐 렌더링을 하게 됩니다.

6. 물 생성

물은 CScene에서 생성, 렌더링됩니다. 물을 표현하는 CRippleWater 클래스는 생성자에서 Terrain과 같은 크기의 XZ 평면의 257 * 257개의 점을 가진 사각형 메쉬를 적당한 위치에 생성하고 물을 표현할 베이스 텍스쳐로 Water_Base_Texture_0.dds와 물의 일렁이는 느낌을 표현할 Water_Detail_texture_0.dds를 불러와서 저장하고 물을 렌더링할 때 사용할 그래픽스 파이프라인 상태 객체를 생성할 CRippleWaterShader라는 클래스를 생성합니다. CRippleWaterShader는 블렌딩을 알파값을 이용해 할 수 있게 설정한 뒤 두 개의 SRV를 저장할 디스크립터 힙을 생성하고 불러온 텍스쳐로 SRV를 설정합니다. CRippleWaterShader의 정점 쉐이더에서는 CGameFramework에서 렌더링을 할 때, 설정한 타이머부터 얻어온 현재시간, 정점 x 좌표, 정점 z 좌표를 이용해서 정점의 y 좌표에 보정을 줘서 y축으로 굴곡이 있는 형태를 만들었고 색상도 y축 좌표가 높을수록 색상이 진하게 표현이 되도록 색상에도 보정을 해주었습니다. CRippleWaterShader의 픽셀 쉐이더에서는 물 베이스 텍스쳐와 디테일 텍스쳐를 3:7로 블렌딩 해 최종 색상을 결정해주었고 블렌딩 상태 객체를 통해 지형과 물이 블렌딩 될 수 있도록 색상의 알파값을 0.55로 설정해줍니다. 이때, 디테일 텍스쳐의 uv 좌표에서 u 좌표는 시간에 따라 변화를 줘서 텍스쳐가 움직이는 듯한 효과를 주었습니다.

=> 4번째 실행화면 캡처에서 물 아래의 지형이 블렌딩 되어서 보이는 것을 확인할 수 있습니다.

7-1. 총알 생성 및 발사

총알은 CTerrainFlyingPlayer 클래스의 생성자가 불릴 때, PrepareShooting()이라는 함수로 최대 100개의 총알을 생성할 수 있도록 총알 객체를 관리하는 CBullet 객체들을 std::vector 컨테이너에 y축으로 회전하게 담습니다. 이때 사전에 생성한 총알을 표현할 육면체의 텍스쳐 매핑을 한 큐브 메쉬를 가지는 하나의 m_pBullet 이름의 총알 객체를 자식으로 갖게 하는 방법으로 총알 모델을 모든 CBullet 객체가 공유하게 해주었습니다. 프로그램이 실행되고 Ctrl 키를 누르면 std::find_if를 사용해 100개의 총알 객체들을 담고 있는 m_pBullets에서 활성화가 되어있지 않은 총알을 찾아 그러한 총알이 있다면 해당 총알 객체를 활성화 시키고, 위치를 플레이어의 위치로 설정해주며 이동방향을 플레이어가 바라보는 방향으로 설정해줍니다. 생성된 총알들은 매 프레임 CTerrainFlyingPlayer::Animate() 함수가 호출 될 때, 활성화가 되어있는 총알들을 설정된 방향으로 프레임 타임 * 속도만큼 이동시켜주면서 총 이동한 거리가 1000.0f보다 커지면 다시 비활성화해 발사 가능한 상태가 되도록 설정해주었습니다. 총알의 위치를 업데이트를 한 뒤에서는 UpdateBoundingBox()로 바운딩박스를 현재 변환 행렬을 가지고 업데이트 해줍니다. 렌더링은 플레이어와 같은 쉐이더를 사용해서 했습니다.

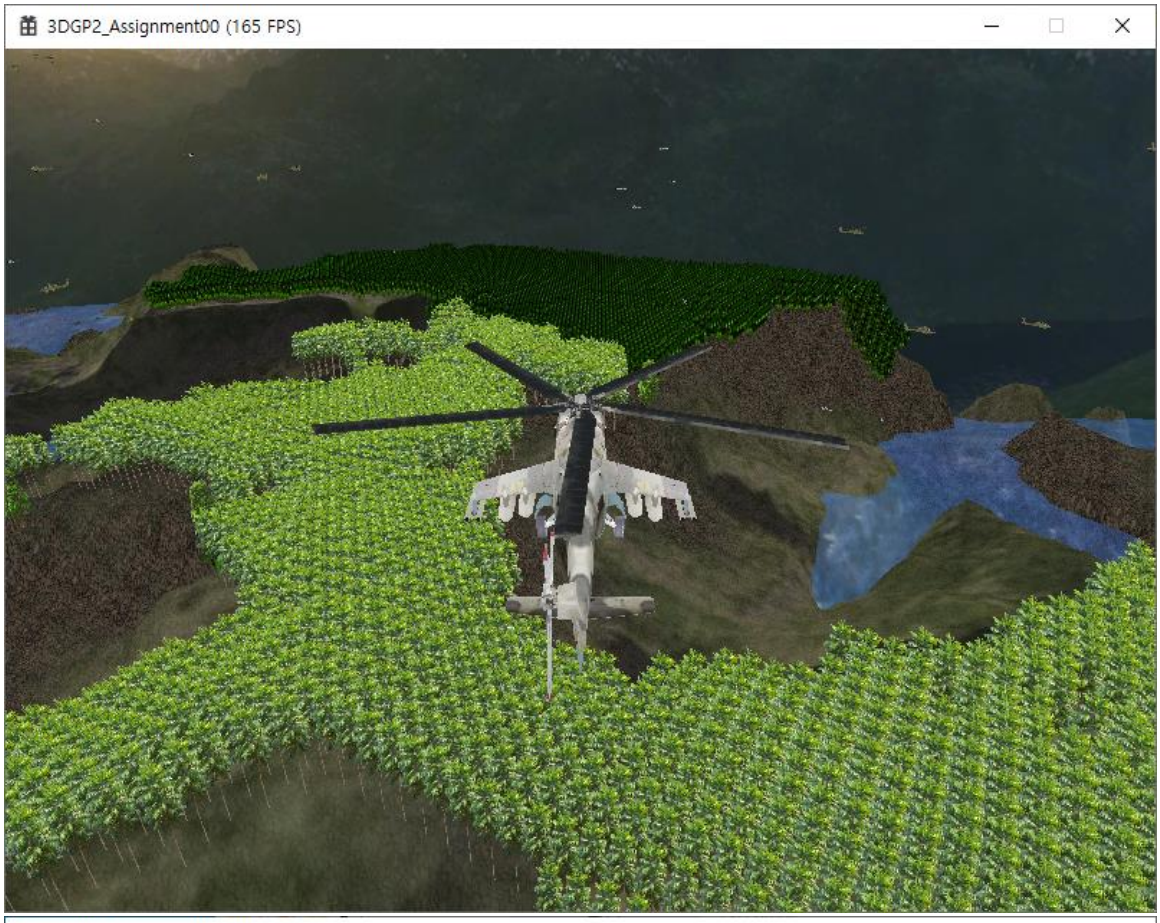
7-2. 오브젝트 충돌, 스프라이트 효과

오브젝트의 충돌은 CScene에서는 매 프레임마다 CheckCollision() 함수가 호출되면 적 객체들에 관한 정보가 있는 CObjectShader를 싱글톤으로 호출해 for문으로 전체를 순회하며

CEnemy::CheckCollision()을 CTerrainFlyingPlayer::getBullets() 함수를 통해 가져온 총알들이 담긴 std::vector를 인자로 호출해줍니다. 적 객체들을 관리하는 CEnemy 클래스는 생성자에서 m_xmOOBB라는 이름의 BoundingOrientedBox를 Center를 {0.0f, 0.0f, 0.0f}, Extents를 {5.0f, 5.0f, 5.0f}로 생성하고 앞서 말했듯이 매 프레임마다 Update() 함수가 불릴 때, 이 바운딩 박스를 적 객체 변환행렬을 적용해 업데이트해줍니다. 그러면 CEnemy::CheckCollision()에서 인자로 넘어온 총알들의 std::vector를 for문으로 돌면서 총알들이 활성화 되어있으면 적 객체의 바운딩박스와 총알의 바운딩박스를 BoundingOrientedBox::Intersects()를 통해 충돌을 검사합니다. 총알의 바운딩 박스의 경우 CBullet 클래스의 생성자에서 m_xmOOBB라는 이름의 BoundingOrientedBox를 Center를 {0.0f, 0.0f, 0.0f}, Extents를 {3.5f, 3.5f, 3.5f}로 생성해 매 프레임마다 업데이트해줍니다. 적 객체와 충돌이 확인되면 적 객체의 활성화 여부를 나타내는 m_bEnable을 false로 설정해주어 더 이상 렌더링이 되지 않게 하고, 스프라이트 효과를 보여주는 CMultiSpriteObjectsShader 클래스를 싱글톤으로 호출해 AddObject() 함수로 현재 적 객체 위치를 추가시켜 현재 적 객체 위치에서 폭발하는 듯한 스프라이트 이미지를 출력하게 해주었습니다.

CMultiSpriteObjectsShader 클래스는 폭발하는 스프라이트 애니메이션을 렌더링하기 위해 정의된 클래스로 그래픽스 파이프라인 상태 객체에 대한 설정은 나무, 풀, 꽃을 표현하는 CBillBoardObjectsShader 클래스와 같습니다. CMultiSpriteObjectShader는 CScene의 BuildObjects()에서 CreateShader()와 BuildObjects() 함수가 호출되는데 CreateShader()는 그래픽스 파이프라인 상태 객체를 설정하고 생성하는 함수이며 BuildObjects()에서는 스프라이트 애니메이션을 표현할 Explode_8x8.dds라는 이름을 가진 텍스쳐 파일을 불러와서 디스크립터 힙에 SRV로 저장하고 이 텍스쳐를 입힐 CTexturedRectMesh라는 이름의 사각형 메쉬를 생성합니다.
CMultiSpriteObjectsShader::AddObject()에서는 XMFLOAT3의 형태로 스프라이트 애니메이션이 출력될 위치를 전달받아 해당 위치에 CMultiSpriteObject라는 스프라이트 애니메이션을 출력하는 객체를 생성해 m_ppObject라는 이름을 가진 std::list<CMultiSpriteObject*> 컨테이너에 추가합니다.
CMultiSpriteObject 클래스는 스프라이트 애니메이션을 업데이트하는 객체를 생성하기위해 정의한 클래스로 CMultiSpriteObject::Animate()에서 매 프레임마다 출력해야할 스프라이트를 텍스쳐 변환행렬로 표현해주면 이를 정점 쉐이더에서 루트시그니쳐로 전달받아 uv 좌표에 대해 변환을 해주어 스프라이트 애니메이션이 연속적으로 출력이 되게 해줍니다. 만약 폭발하는 스프라이트 애니메이션을 충분히 출력했다면 FullAnimated라는 bool형 변수를 true로 바꿔주고 이는 CMultiSpriteObjectsShader::AnimateObjects()에서 매 프레임마다 검사해 FullAnimated라면 m_ppObjects에서 std::list<>::erase()를 통해 삭제시켜줍니다. FullAnimated가 아니라면 출력해야하는 상태이므로 CMultiSpriteObjectsShader::Render()에서 스프라이트 애니메이션을 출력하는 메쉬의 방향을 카메라를 바라보게 설정해주고 Render() 함수를 호출해줍니다.

실행결과


3DGP2_Assignment00 (165 FPS)


3DGP2_Assignment00 (165 FPS)