3D GameProgramming2 – 1st report

2018180009 김시인

한국공학대학교

TECH UNIVERSITY OF KOREA

실행 환경 : visual studio 2022, Release x64, 최신 C++ 초안의 기능(/std:c__latest)

CPU – AMD Ryzen 7 5800H, RAM 16GB, GPU - RTX 3070

제가 제출한 과제는 팀원(2018182034 전수민)과 함께 만들고 있던 졸업작품을 기반으로 하였습니다.

팀원과 같이 작업한 부분 :

- 디바이스생성을 생성하고 게임 프레임워크를 동작시키는 부분.
- 원시 포인터 대신 스마트 포인터를 이용한 부분
- 오브젝트와 메쉬를 불러오는 부분은 같이 작업하였으나 저만의 방식으로 수정 보완하였습니다.

조작키:

w, a, s, d – 수평이동 및 해당 방향으로 회전

space, ctrl - 상승/하강

마우스 드래그 – 카메라 이동

q – 미사일 발사

k - OOBB렌더링 on/off

1. 게임 오브젝트 (Load, Create)관리

"게임 오브젝트 매니저(GameObjectManager)" 라는 클래스를 만들어 오브젝트를 관리하도록 만들었습니다. 게임 오브젝트 매니저는 파일로부터 계층구조의 게임 오브젝트를 읽어와서 map 컨테이너를 이용하여 파일의 이름과 게임 오브젝트 (스마트)포인터를 매핑합니다.

만약 게임 오브젝트 매니저에게 게임 오브젝트를 요구한다면 아래와 같이 동작합니다:

그 오브젝트를 이미 파일로부터 읽어와서 메모리에 있는지 확인하고 있으면 오브젝트를 Copy하여 전달해주고 없다면 파일로부터 읽어와 메모리에 저장하고 나서 Copy한 오브젝트를 넘겨줍니다.

위와 같이 구현한 이유는 만약 헬기 오브젝트를 생성해야 할 때마다 파일로부터 읽어온다면 메모리에 있는 정보를 그대로 복사하는 것보다 훨씬 느리기 때문에 오브젝트 매니저를 활용하여 관리하는 시스템을 구현하였습니다.

또한 공유하는 자원을 효율적으로 관리할 수 있습니다. 예를 들어 헬기 오브젝트를 파일로부터 읽어온다면 헬기 메쉬의 리소스정보가 새로 만들어 질 텐데 이미 앞에서 같은 헬기를 만들었다면 결과적으로 같은 메쉬를 또 한번 만드는 것이 됩니다. 저는 이 문제를 게임 오브젝트 매니저를 활용하여 오브젝트를 복사하여 넘겨줄 때 메쉬의 리소스정보를 새로 만들지 않고 이미 만들어진 리소스의 포인터만 복사하여 넘겨줌으로써 리소스가 중복되어 load하는 문제 해결하였습니다.

처음에는 c++의 map 컨테이너를 사용하였으나 굳이 정렬될 필요가 없다고 판단하여 속도개선을 위해 unordered_map을 사용하였습니다.

2. 헬리콥터 애니메이션:

GetChild(const string) 함수를 DFS를 자식중에 매개변수로 넘겨온 이름과 같은 자식이 있는지 확인하고 있다면 해당하는 자식에 대한 포인터를 넘기는 함수를 만들었습니다.

GetChild함수를 이용하여 헬기의 메인 날개와 꼬리 날개를 찾아 각각 localY축과 localX축으로 회전하도록 만들었습니다.

3. 카메라와 플레이어 이동:

일반적인 FPS게임의 3인칭 카메라를 구현하고 싶어 플레이어(헬리콥터)의 회전과 카메라의 회전을 분리시켜 구현해 보았습니다. 샘플 프로젝트에서는 마우스를 클릭한 채로 움직이면 헬리콥터가 움직이지만 제가 만든 프로젝트에서는 카메라만 회전하도록 만들었습니다. 그리고 플레이어의 회전은 방향키 w,a,s,d와 카메라가 바라보는 방향을 이용하여 해결했습니다. 자세하게 설명한다면 카메라가 보고 있는 방향벡터를 XZ평면에 투명하여 XZ에 평행한 벡터를 만들고 정규화하여 수평으로 이동할 벡터를 얻고 헬리콥터의 speed값과 경과시간을 곱하여 수평이동을 구현하였습니다.

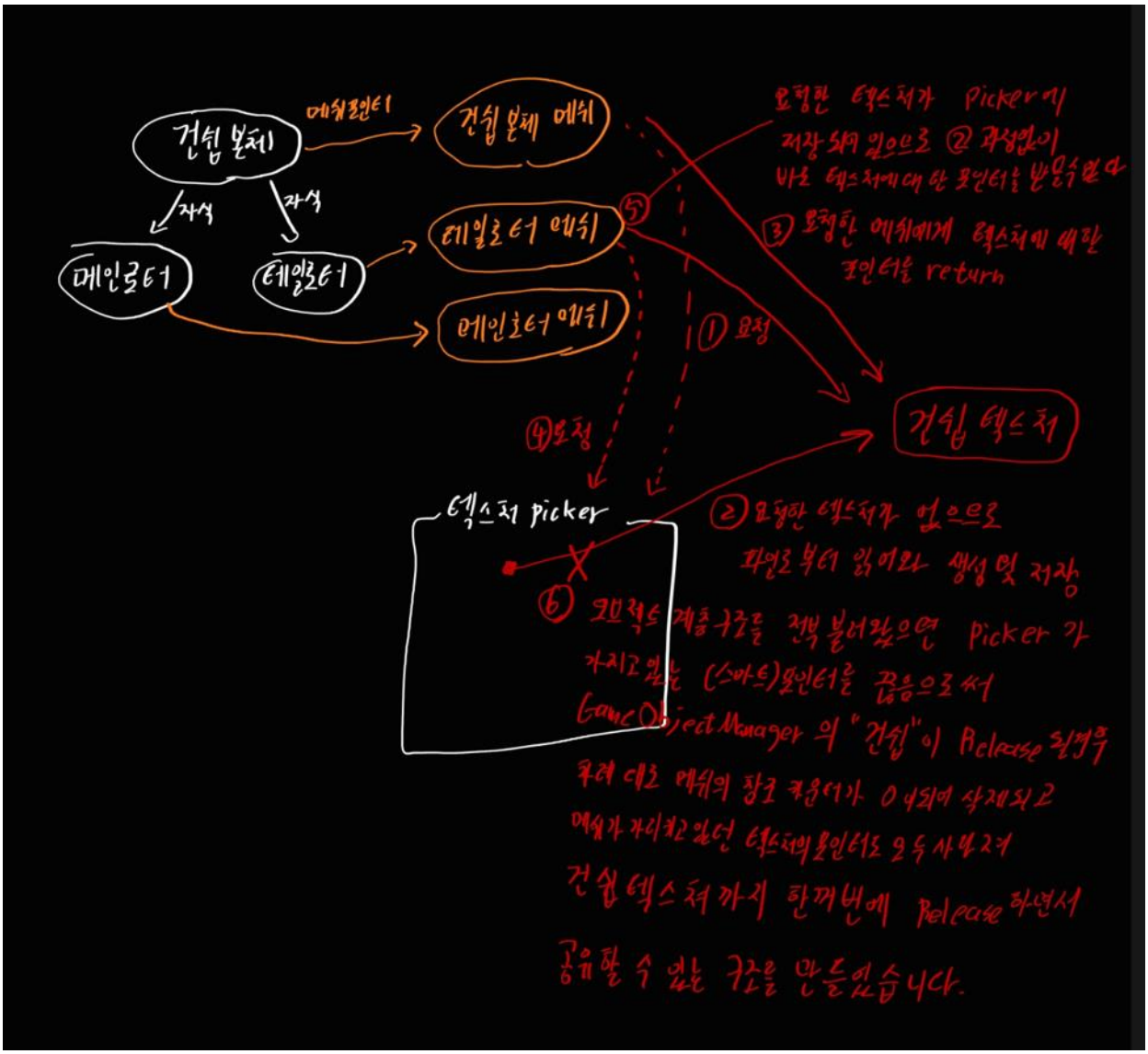
4. 텍스처 :

dds 파일을 읽어와서 사용할 수 있도록 만들었습니다.

<실제로는 메쉬가 메테리얼을 포인터로 가지고 있고, 메테리얼이 텍스처의 포인터를 가지고 있지만 간략한 설명을 위해 생략하였습니다>

처음 오브젝트, 메쉬, 메테리얼을 불러올 때 텍스처를 같이 불러오도록 만들었습니다. 하지만 똑 같은 텍스처를 여러 번 파일로부터 읽어와 새로운 리소스를 생성하여 비효율적이었습니다.

그래서 같은 그림의 텍스처를 공유하면서 Release를 쉽게 할 수 있는 구조로 만들어주는 TexturePicker라는 클래스를 작성하였습니다. 동작하는 방식은 다음장에서 그림과 함께 설명하겠습니다.



1. 메쉬가 텍스처 Picker에게 텍스처 이름을 매개변수로 주면서 요청을 합니다.
2. 요청을 받은 텍스처 Picker는 매개변수로 받은 이름을 키로 unordered_map<string, Texture> 컨테이너로 관리하고 있는 공간에 요청한 텍스처가 있는지 확인합니다. 요청한 텍스처가 없으면 파일로부터 텍스처를 읽어오고 저장합니다.
3. Return 값으로 읽어온 텍스처의 (스마트)포인터를 넘겨주어 메쉬가 건쉽 테스처를 포인트로 가리키도록 합니다.
4. 테일로터 또한 같은 텍스처를 요청하므로 똑 같은 이름을 매개변수로 주면서 요청합니다.
5. 이번에는 겁쉽 텍스처가 이미 컨테이너 안에 있으므로 파일로부터 읽어올 필요없이(2번 과정 필요없이) 바로 포인터만 넘겨주어 테일로터와 건쉽본체 둘다 같은 텍스처를 공유할 수 있도

록 만들 수 있습니다. 위와 같은 방식을 반복해 같은 텍스처는 공유하면서 모든 텍스처를 불러옵니다.

6. 오브젝트 계층 구조를 전부 불러왔다면 TexturePicker가 가지고 있는 (스마트)포인터를 끊습니다. 이렇게 하면 추후 "건쉬 오브젝트 계층"을 Release하면 가리키고 있는 메쉬의 참조 카운터가 0이 되어 소멸되고 메쉬가 전부 소멸되면 건쉽 텍스처를 가리키던 (스마트)포인터도 전부 사라져 자동으로 소멸되게 됩니다. 결국 게임 오브젝트만 삭제하면 메쉬와 텍스처가 차례차례 소멸되는 구조를 만들었습니다. 만약 texturePicker가 포인터로 계속 가리키고 있다면 참조 카운터가 0이 되지 않기 때문에 소멸되지 않습니다. 그래서 마지막에 꼭 TexturePicker의 포인터를 끊어줘야 합니다.

5. 지형:

지형에도 똑같이 텍스처를 입히고 추가적으로 디테일 텍스처를 추가하였습니다.

지형의 메쉬를 생성할 때 가로, 세로, 높이와 정점이 반복될 넓이를 인자로 받아 좀더 직관적이고 간편하게 메쉬를 만들 수 있도록 하였습니다. 추가적으로 디테일 텍스처가 반복되는 넓이도 인자로 받을 수 있도록 만들었습니다.

6. 빌보드:

```
struct VS_BILLBOARD_INPUT
{
    float3 position : POSITION;
    float2 boardSize : BOARDSIZE;
};
```

기하쉐이더를 이용하여 빌보드를 구현하였습니다.

정점쉐이더로 모델 좌표계기준 빌보드의 중심좌표와 가로, 세로 크기만 보내주면 기하쉐이더에서 전달 받은 가로, 세로크기 만큼의 사각형(삼각형 2개를 TriangleStream)으로 만들어 픽셀쉐이더로 전달하도록 만들었습니다. 또한 빌보드는 항상 카메라를 쳐다봐야 하기 때문에 루트 시그니처 파라메터로 넘어온 카메라의 포지션 값을 이용하여 노멀벡터를 구했습니다.

그렇게 구현한 빌보드용 쉐이더를 이용하여 나무 1000그루를 물이 없는 지형에 랜덤하게 생성되도록 만들었습니다.

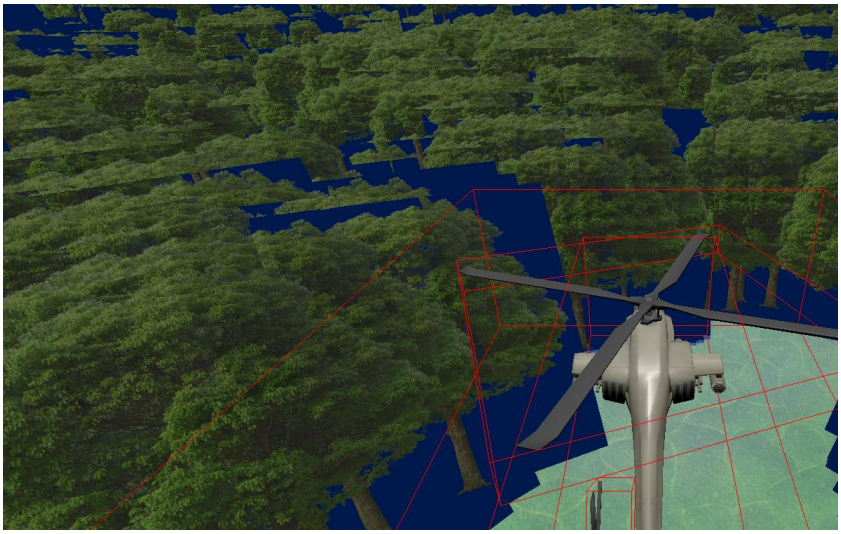
```
extern random_device rd;
```

```
uniform_real_distribution<float> urd(0, 2000.f);
```

```
//높이 맵 이미지에서 (x, z) 위치의 픽셀 값에 기반한 지형의 높이를 반환한다.
float GetHeight(float _terrainX, float _terrainZ) const;
```

랜덤으로 생성하기 위해서 랜덤 디바이스를 생성하고 실수 균등 분포 객체를 생성하여 x, z위치를 정하고, 해당 x, z위치의 터레인 높이를 높이맵 이미지로부터 구해와서 y값을 구하도록 만들었습니다.

위의 그림에 표시한 것과 같이 나무의 가장자리 부분은 투명하기 때문에 헬기와 같은 일반적인 오브젝트를 그릴 때처럼 하게 된다면 투명해서 그려지면 안되는 부분인 데도 불구하고 깊이가 write가 되어 아래의 그림과 같이 투명한 부분에 의해 짤려서 그려지게 됩니다.



그것을 해결하기 위해 빌보드들을 멀리있는 것부터 가까운 것 순으로 정렬을 하고 빌보드를 그릴 때 깊이 검사만하고 깊이 값은 쓰지 않도록 DepthWriteMask를 D3D12_DEPTH_WRITE_MASK_ZERO 로 설정해 주었습니다. 그렇게 했을 때 아래와 그림과 같이 정상적으로 그려졌습니다.

여기서 졸업작품 팀원이 discard를 사용하면 굳이 정렬을 하지 않아도 된다고 알려주어 새롭운 방식으로 수정하였습니다. 실제로 텍스처가 반투명없이 완전투명(알파값 0)과 불투명(알파값 1)만 존재한다면 알파값이 0일 때만 픽셀쉐이더에서 discard를 통해 다음 파이프라인 단계로 보내지 않으면 깊이 값을 쓰지도 전에 버려지므로 정렬을 하지 않고도 정상적으로 그려지도록 할 수 있었습니다.

정렬을 하지 않아도 되기 때문에 성능이 더욱 향상되었습니다. 정렬을 하는 방식일때는 30’000그루에 60FPS였지만 discard를 이용하여 투명한 부분을 버리는 방식을 이용했을 때 55’000그루까지 늘려도 60FPS를 유지하였습니다. 2배 정도의 성능향상이 있었습니다.

[실행환경은 2장에 있습니다]

위의 사진은 성능을 테스트하기 위해 나무를 엄청 많이 생성한 것으로 과제에 올리는 결과물에는 1’000개의 나무만 생성하였습니다.

호수:

특정 높이에 xz평면에 평행한 커다란 사각형 오브젝트를 그렸습니다.

```
pd3dSamplerDescs[0].AddressU = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
pd3dSamplerDescs[0].AddressV = D3D12_TEXTURE_ADDRESS_MODE_WRAP;
```

호수를 그리기 위해서 물 이미지의 텍스처가 반복해서 그려지도록 위의 사진과 같이 옵션을 준 샘플러를 이용하여 렌더링을 수행하였습니다.

```
blendDesc.RenderTarget[0].SrcBlend = D3D12_BLEND_SRC_ALPHA;
blendDesc.RenderTarget[0].DestBlend = D3D12_BLEND_INV_SRC_ALPHA;
```

그리고 물이 맑아 속이 비춰 보이는 것을 표현하기 위해 Blend State를 위와 같이 설정하여

물의 색상 * 물의 알파값 + 쓰여진 색상 * (1 – 물의 알파값) 이 되도록 하였습니다.

충돌체크:

저는 충돌체크를 어떻게 하면 최소화하기 위해 두가지 방식을 적용하였습니다.

첫번째는 오브젝트들을 충돌처리가 비슷한 오브젝트끼리 묶어서 관리하는 것입니다.

A와 B라는 오브젝트들이 각각 500개씩 있고 (1)한 컨테이너에 전부 담겨있는 경우와 (1)서로 다른 컨테이너에 담겨있는 경우를 비교해 보겠습니다.

A오브젝트와 B오브젝트의 충돌체크를 한다고 했을 때 (1)의 경우 A입장에서 컨테이너를 순회하면서 B오브젝트인지 확인하고 충돌체크를 해야 합니다. 이것 모든 A가 수행해야 하므로 1000*1000번 순회해야 하지만, (2)의 경우 다른 컨테이너에 각각500씩 들어있으므로 500*500번 만에 해결 가능합니다. A와 A의 충돌까지 체크한다고 해도 마찬가지로 (2)의 경우가 더 효율적입니다. 이렇게 비슷한 충돌을 처리하는 오브젝트끼리 컨테이너로 묶어서 관리하면 불필요한 오브젝트에 접근을 하지 않아도 되기 때문에 속도가 향상됩니다.

* Object Oriented Bounding Box를 OOBB라고 칭하겠습니다.

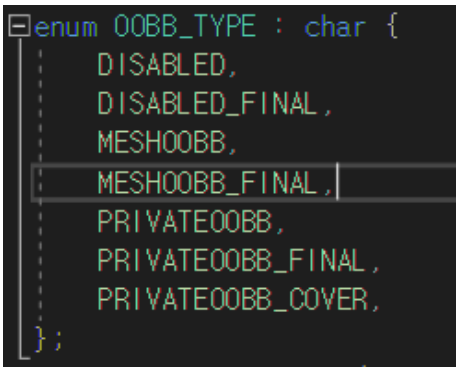
처음 충돌 체크하는 함수를 만들 때 나를 포함한 나의 자식들과 상대를 포함한 상대의 자식들 전부 OOBB가 겹치는지 확인하여 하나라도 충돌할 경우 true를 반환하도록 만들었습니다.

그렇게 만들 경우 충돌검사횟수를 생각해 보았습니다. 만약 자식이 9개(자신을 포함하면 10개의 오브

젝트로 구성된 계층구조)있는 오브젝트A가 있다고 할 경우 A와 A의 충돌을 체크할 때 최악의 경우 10*10번의 충돌체크를 해야 합니다.

만약 오브젝트A가 100개 존재한다면 (10 * 10 * 99개) * (100 / 2)= **495000** 번의 OOBB충돌체크를 하게 됩니다.

저는 OOBB충돌체크 횟수를 줄이기 위해 OOBB_TYPE을 아래와 같이 나누고 타입에 따라 다른 방식으로 충돌처리를 하도록 만들었습니다.



DISABLED – 충돌체크를 하지 않겠다. MESHOOBB – mesh에 있는 OOBB를 월드변환해서 사용하겠다.

PRIVATEOOBB – 따로 OOBB를 만들어서 그것을 사용하겠다.

_Final이 붙어 있는 타입의 경우 자식의 OOBB충돌을 하지 않습니다. 자식 오브젝트까지 충돌체크를 할 필요가 없을 경우 Final이 있는 타입을 설정해 줄 경우 불필요한 충돌체크를 줄일 수 있습니다.

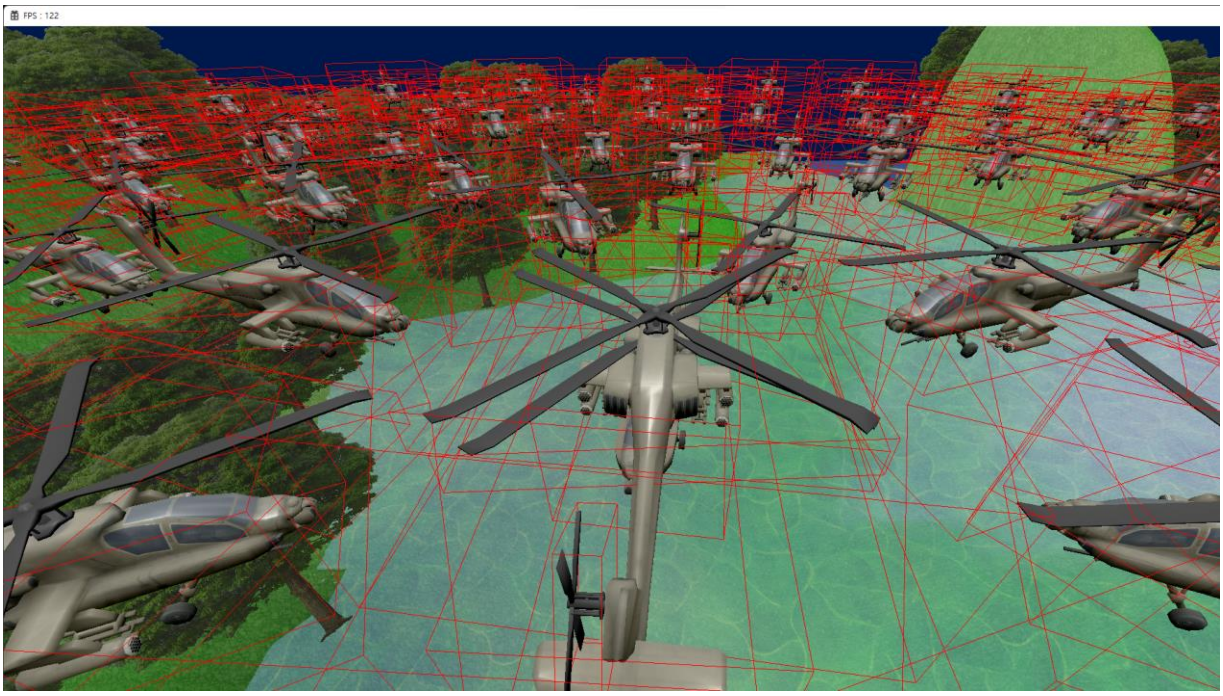
그리고 가장핵심은 _COVER가 붙어 있는 타입입니다. COVER는 자식객체의 OOBB를 대표해서 한번 충돌체크를 해보는 것입니다. 즉, 만약 충돌체크를 하지 않으면 자식들도 모두 충돌하지 않는 것으로 보는 것입니다. 이것이 효과적인 이유는 대부분의 오브젝트와 충돌을 하지 않기 때문입니다. 헬기가 100대씩 있다고 하면 보통 많아야 3~4대와 충돌하고 나머지 96~97대는 충돌하지 않기 때문에 그 96~97대에 대해서 빠르게 충돌하지 않는다는 것을 판단할 수 있어 속도에 큰 이점이 있습니다.

전 장의 예시와 같이 A오브젝트가 100대가 있고 각 오브젝트가 5개의 다른 오브젝트와 충돌한다고 했을 때 ((10 * 10 * 5개) + (1 * 94개)) * (100 / 2) = **29700** 번의 OOBB충돌체크를 하게 됩니다.

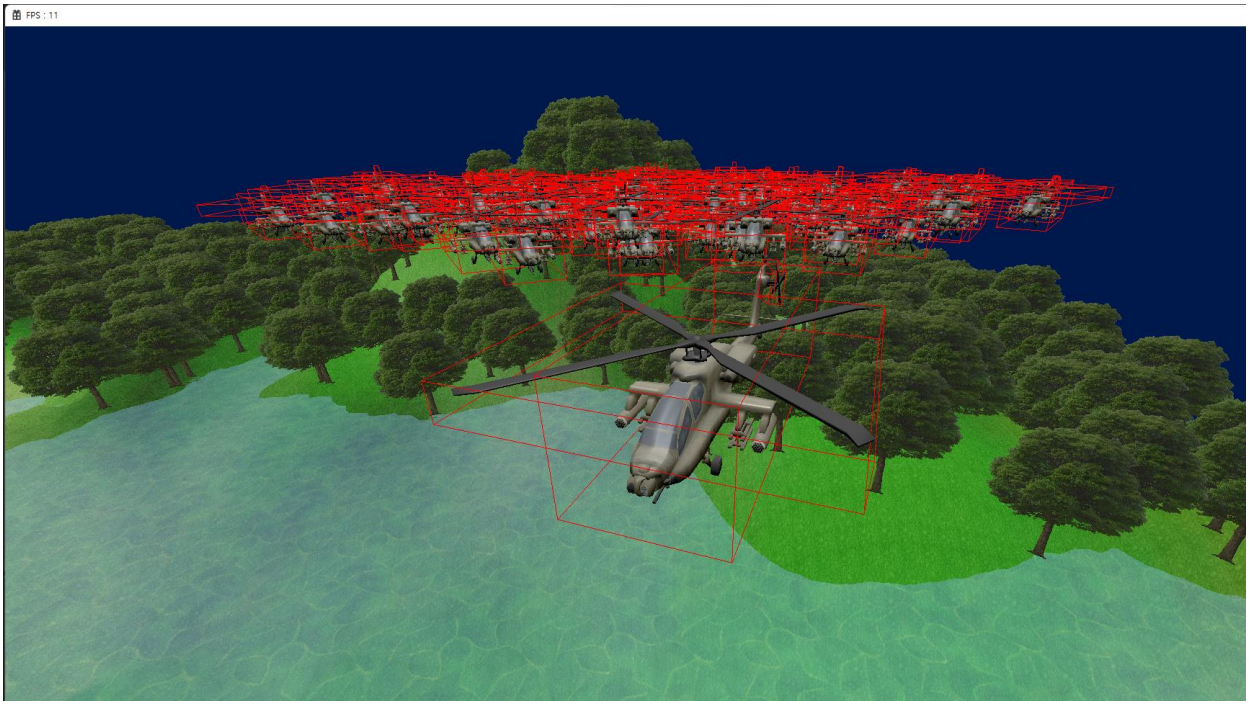
해당 예시로 비교하였을 때 충돌체크의 횟수가 6/100 이나 줄어들게 됩니다. 오브젝트의 개수가 늘어날수록 그 효과는 더욱 클 것입니다.

똑같이 헬기 100대를 생성하고 PRIVATEOOBB_COVER를 만들어준 경우와 그렇지 않은 경우를 비교해 보았습니다.

헬기에 PRIVATEOOBB_COVER를 해줬을 경우 : FPS 120~130



헬기에 PRIVATEOOBB_COVER를 하지 않았을 경우 : FPS 10



공격 : Q키를 누르면 헬기가 바라보고 있는 방향으로 미사일을 발사합니다.

미사일이 적에게 닿으면 적은 사라지게 됩니다.

OOBB 렌더링: 충돌박스를 눈으로 확인할 수 있도록 선으로 렌더링을 하였습니다. K키를 눌러서 on/off 할 수 있습니다.

<실행화면>