3D 게임 프로그래밍 2 – 과제1 보고서

2020184024 이승욱

목차

[**실행 화면 및 조작키** 2](#_Toc210789065)

[**과제 목표** 3](#_Toc210789066)

[**계층 모델 구현** 3](#_Toc210789067)

[**Scene 의 구성** 6](#_Toc210789068)

[**인스턴싱을 위한 과정** 9](#_Toc210789069)

[**Root Signature** 9](#_Toc210789070)

[**RenderManager** 10](#_Toc210789071)

[**결과 분석** 16](#_Toc210789072)

# **실행 화면 및 조작키**

텍스트, 스크린샷, 만화 영화이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

실행 화면입니다. 상단에 표시되는 수치들은 순서대로 프레임 레이트, 그려질 Mesh의 개수, 그려질 인스턴스의 개수, Draw Call 횟수입니다.

프로그램의 조작키는 아래와 같습니다.

|  |  |
| --- | --- |
| 키 | 동작 |
| W, S, A, D, E, Q | Player 이동 (각각 전진, 후진, 왼쪽, 오른쪽, 위, 아래) |
| 마우스 좌클릭 후 드래그 | 카메라 회전 |
| 1, 2, 3, 4, 5, 6 | 모델 변경(각각 Apache, Gunship, SuperCobra, Hummer, M26, Mi24) |
| 방향키 오른쪽, 왼쪽 | 오브젝트 1개씩 추가, 제거 |
| 방향키 위, 아래 | 오브젝트 10개씩 추가, 제거 |
| Home, End | 오브젝트 100개씩 추가, 제거 |
| Page Up, Page Down | 오브젝트 1000개씩 추가, 제거 |

# **과제 목표**

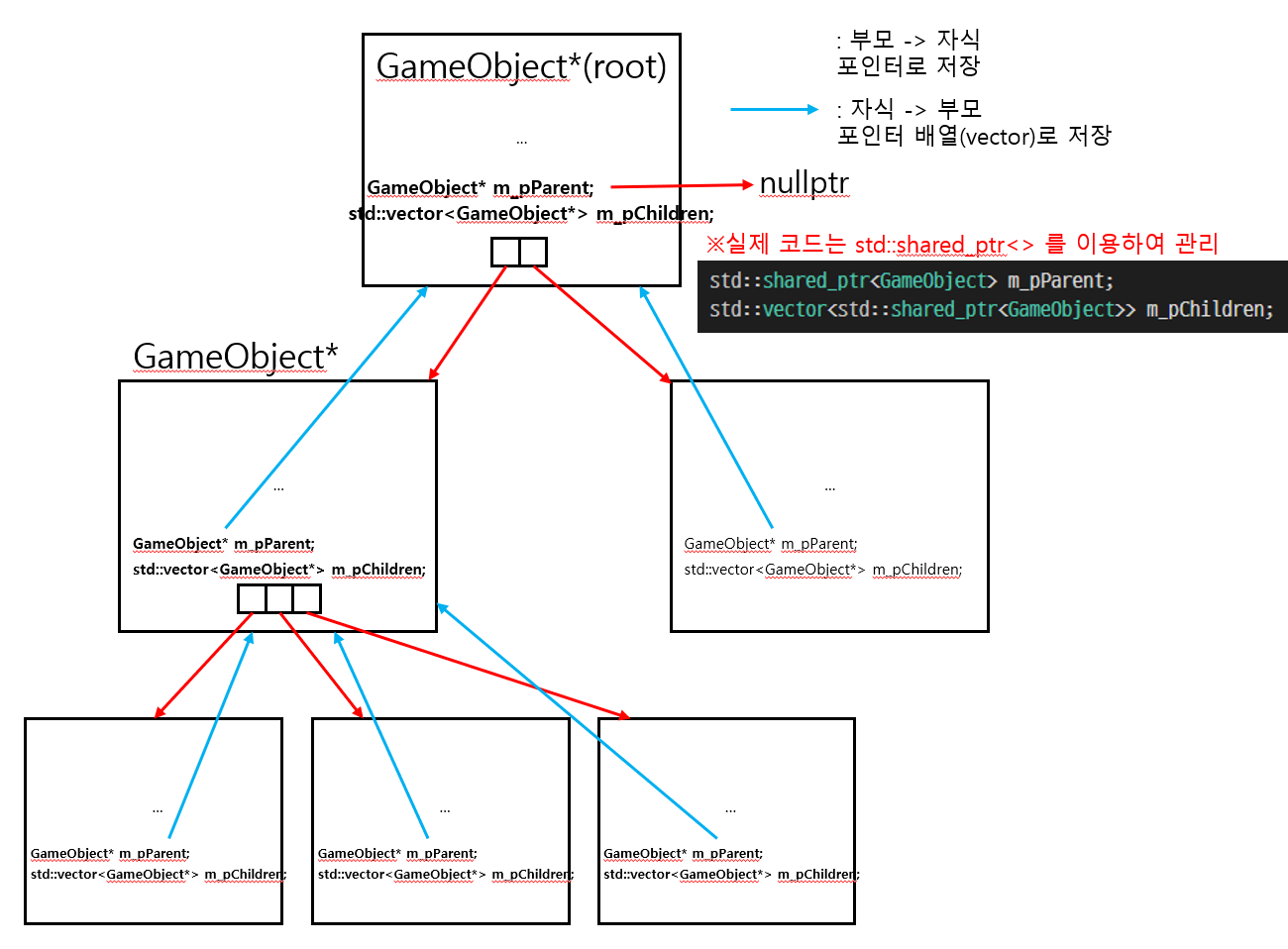
이번 과제1의 목표는 아래와 같습니다.

1. 계층 모델을 그린다.
2. 가능한 프레임레이트 안에서 최대한 많이 그린다. (인스턴싱)

아래부터는 위 목표를 위한 해결 과정을 서술합니다.

## **계층 모델 구현**

기본적인 구조는 LabProject07-9-1 예제코드의 구성을 따라갑니다. 다만 각 계층에서 부모/자식의 참조 방식에 약간의 수정을 하였습니다. 현재 과제 코드에서는 부모/자식 간의 관계를 포인터로 연결시킨 트리의 형태로 구현하였습니다. 구조의 도식화는 아래 그림과 같습니다.



기본적으로 GameObject 객체는 부모에 대한 포인터인 m\_pParent 와 자식에 대한 포인터를 std::vector 를 이용한 m\_pChildren 을 이용하여 구현됩니다. Scene 등 계층 모델 GameObject를 다루는 측에서는 계층 트리의 root 프레임을 참조하도록 합니다. 계층 모델의 root 프레임은 반드시 부모 포인터를 nullptr 을 참조하도록 하여 현재 계층이 root 가 맞는지 구분할 수 있도록 구현되어있습니다.

계층 모델에 대하여 Animate/Update/Render 를 수행할 때에는 Scene 등에서 참조하고 있는 root 프레임으로부터 출발하여 자식들을 저장된 순서대로 재귀적으로 수행됩니다. 따라서 해당 작업들이 수행될 때 계층 프레임들을 깊이우선탐색(DFS) 방식으로 순회하며 작업이 수행됩니다.

LabProject07-9-1 예제와 마찬가지로 계층 모델 중에서 헬리콥터의 로터를 회전시키려는 하는 등의 특수한 GameObject 를 위한 계층 프레임의 참조가 필요한 경우, GameObject 를 상속받는 클래스들을 선언하여 활용하였습니다. 이 구조는 예제코드의 구현과 동일하지만 사용되지 않는 클래스들은 코드에서 제외시켰습니다. 예제 코드와 함께 제공된 바이너리 모델 파일들을 전부 사용하고 있습니다. 구현된 특수 GameObject 들의 구조는 아래와 같습니다.

텍스트, 라인, 도표, 그래프이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

상속받은 각 클래스들은 필요한 프레임들을 Initialize() 함수에서 참조하도록 하고, Animate() 함수에서 원하는 애니메이션을 위한 변환 행렬을 업데이트합니다.

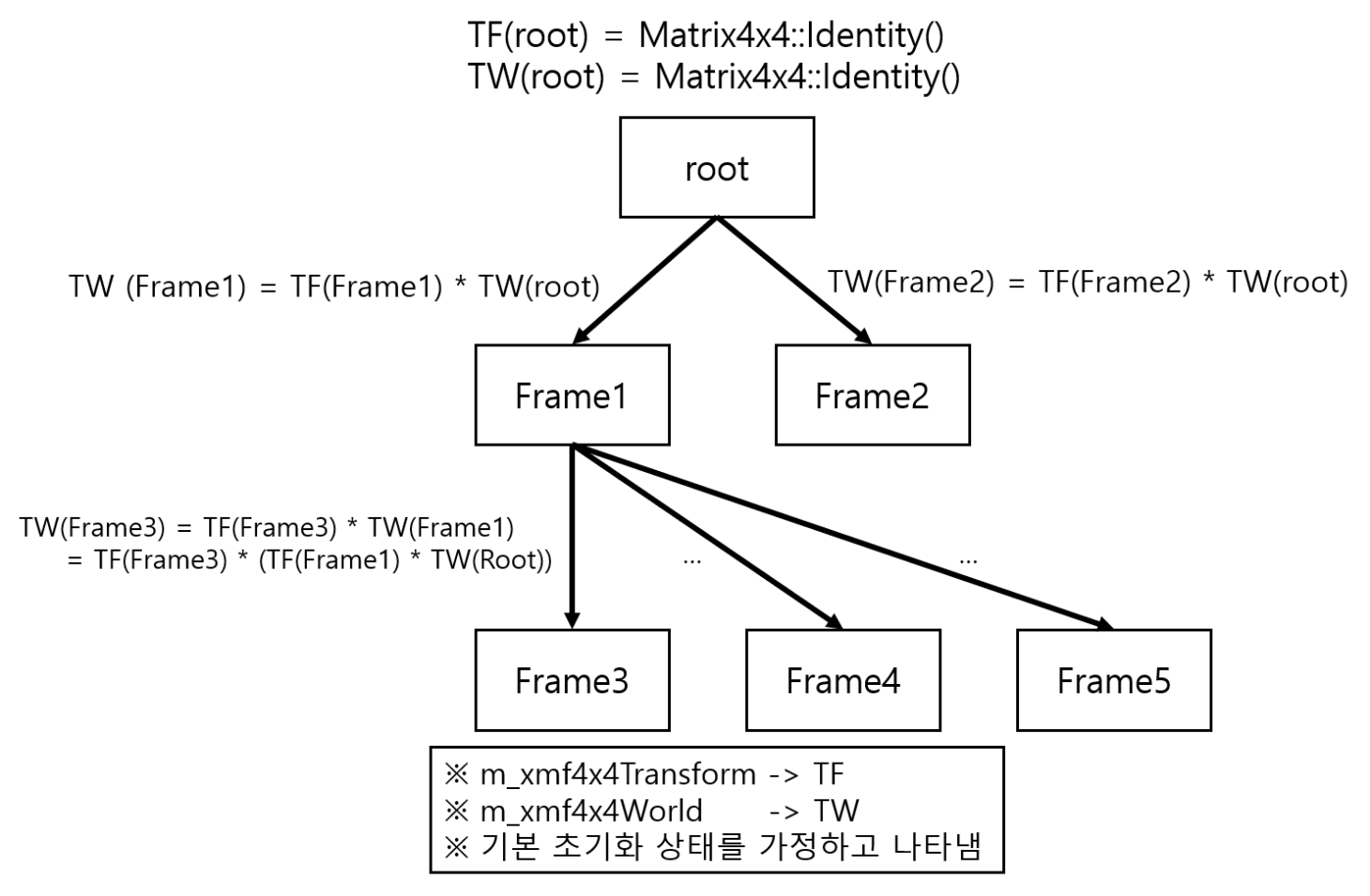
텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

위 코드 예시는 LabProject07-9-1 예제코드에는 구현되어있지 않은 Hummer 모델의 Initialize() 와 Animate() 의 예시입니다. Initialize() 에서는 각 바퀴를 참조하도록 하고 Animate() 에서 매 프레임 바퀴를 회전시키도록 구현하였습니다. FindFrame() 함수는 계층 트리를 순회하며 이름이 일치하는 프레임(프레임)의 참조를 리턴하도록 구현하였습니다. 마찬가지로 DFS 방식으로 계층 트리 순회를 하고, C++의 std::shared\_ptr<>를 이용하여 객체들이 관리되고 있으므로, shared\_from\_this() 를 이용하여 스마트 포인터 내부의 reference count 가 꼬이지 않도록 하고 있습니다. 찾으려는 프레임을 찾지 못한다면 nullptr을 리턴하도록 하였습니다. 예제코드의 동일한 이름의 함수를 현재 구조에 맞게 DFS로 바꾼다고 생각하고 구현하였습니다.

계층 모델을 구현하기 위한 또 다른 중요한 부분은 각 계층을 렌더링하기 위한 계층 변환 행렬의 업데이트 입니다. 우선 모델 바이너리 파일들 안에는 <TransformMatrix> 라는 정보가 담겨져 있습니다. 이 정보는 해당 프레임과 부모 프레임으로부터의 상대적인 변환 행렬이 담겨져있는 정보입니다. 모델을 파일로부터 읽어올 때, 이 행렬을 각 프레임의 m\_xmf4x4Transform 행렬 필드에 저장합니다.

계층 변환을 위해서는 각 노드들의 계층 변환 행렬을 부모로부터 누적하여 계산해야 합니다. 아래는 계층 변환 행렬을 누적하여 각 프레임의 월드 변환 행렬이 구해지는 과정을 그린 그림입니다.



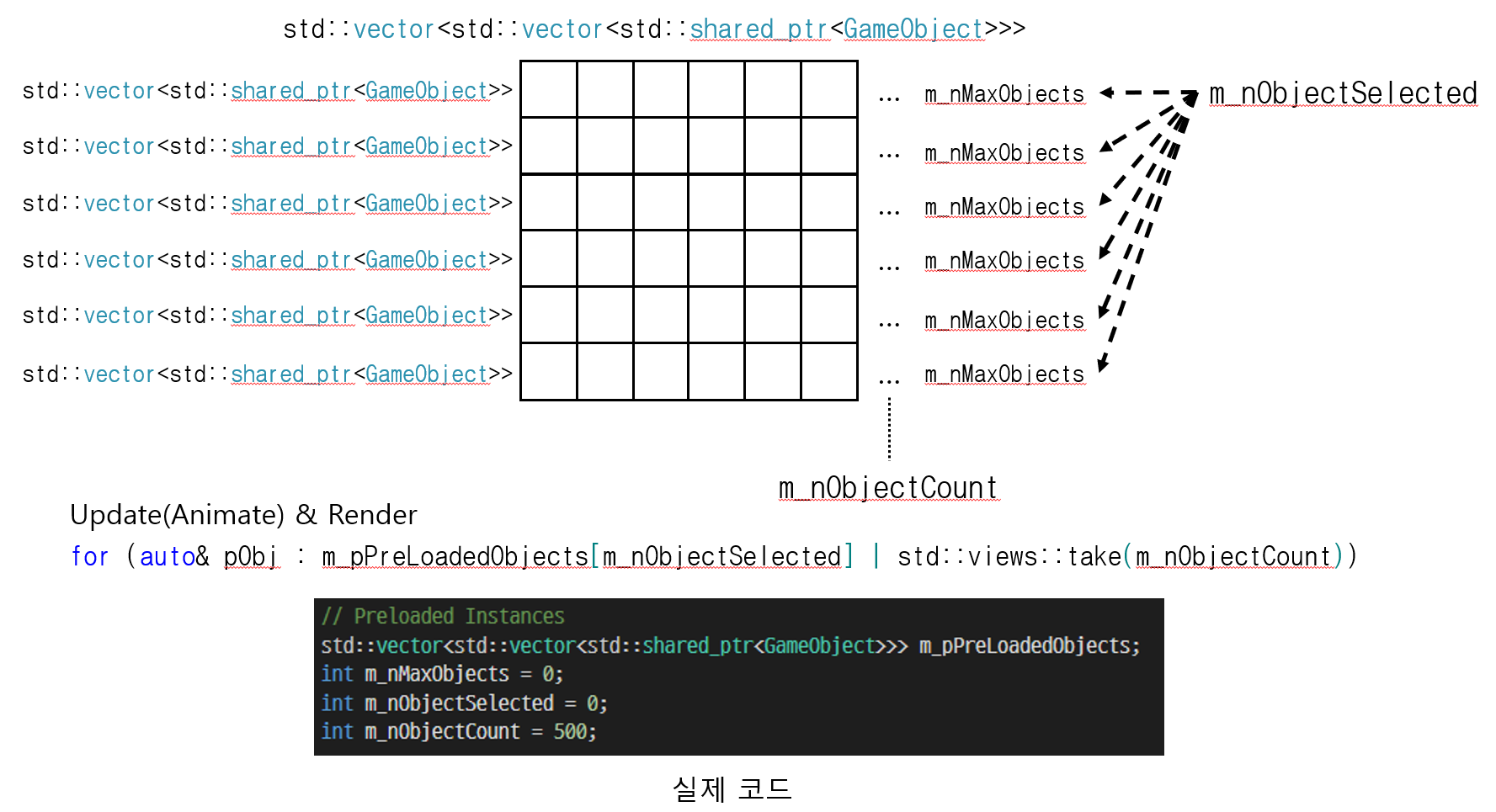
기본적으로 root 프레임의 계층 변환 행렬은 모델 자체의 월드 변환 행렬과 같습니다. 자식 프레임들의 경우 각각의 프레임의 원점을 중심으로 하는 로컬 좌표계를 따라서 메쉬들이 파일로부터 읽어져옵니다. 이들을 월드 공간으로 변환시키기 위해서는 부모로부터 월드 변환 행렬을 계속해서 누적해야만 합니다. 이를 구현하기 위한 함수는 GameObject::UpdateTransform(XMFLOAT4X4\* pxmf4x4Parent) 함수로 구현되어 있습니다. 이 함수도 LabProject07-9-1 의 구현과 동일하지만 현재 구조에 맞는 DFS 방식의 순회를 하면서 변환 정보를 업데이트합니다.

UpdateTransform 함수의 인자로는 부모의 월드변환 행렬을 포인터를 이용한 참조로 받아옵니다. root 프레임의 경우 부모가 없으므로 인자로 nullptr(디폴트인자로 선언됨)을 전달하면 자신의 계층 변환 행렬이 그대로 월드 변환 행렬로 대입됩니다. 자식 프레임들의 경우 부모의 월드 변환 행렬을 전달받아 자신의 계층 변환 행렬과 곱하여 월드 변환 행렬을 구하고 이것을 또 자식의 UpdateTransform() 의 인자로 전달해줍니다. 이렇게 하여 root 프레임의 포인터를 가지고 관리를 하는 Scene 등의 클래스에서는 최상위 부모에 대한 UpdateTransform() 호출 한번으로 전체 프레임들의 월드 변환 행렬 정보를 업데이트 하게 됩니다.

GameObject의 기본적인 이동, 회전등의 변환은 m\_xmf4x4Transform 행렬에 누적됩니다. 누적된 m\_xmf4x4Transform 행렬은 Animate 등의 업데이트 과정이 끝나고 Scene 에서 UpdateTransform() 함수가 호출되어 root 로부터 부모까지 누적하여 곱해진 변환행렬과 곱하여 최종적인 월드 변환 행렬이 구해지고 이 행렬이 m\_xmf4x4World 행렬에 저장됩니다. m\_xmf4x4World 행렬은 나중에 렌더링 시에 인스턴스별 데이터로서 GPU로 전달시킵니다. 이 부분은 렌더링 부분을 다루면서 좀 더 자세히 설명하겠습니다.

## **Scene 의 구성**

Scene 에서 오브젝트를 생성 및 배치하고 그리는 구조는 아래 그림과 같습니다.

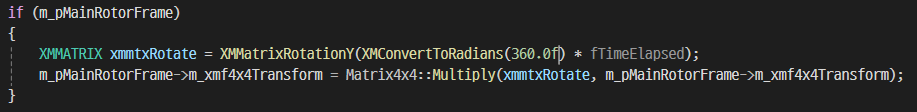


생성되고 배치된 계층 모델 오브젝트들은 2차원 std::vector 배열에 저장됩니다. 2차원 배열에 저장하는 이유는 LabProject07-9-1 예제 코드에 제공된 계층 모델 6개를 미리 읽어와 원하는 개수만큼 미리 저장해놓기 위함입니다. 2차원 배열에 저장하고 나면 m\_nObjectSelected 변수를 인덱스로 하여 그릴 모델을 정하고, m\_nObjectCount 변수를 이용하여 몇 개나 그릴지 조절합니다. 위 변수들을 조정하는 키 입력은 위에서 정리해두었습니다. 나중에 오브젝트를 업데이트하고 렌더링 할 때에는 C++20 의 std::views::take(n) 뷰를 조합하여 원하는 개수만큼만 앞에서부터 가져와 작업을 수행하도록 합니다.

오브젝트를 배치시키는 방법은 이전 따라하기 예제 코드와 동일합니다. x,y,z 축으로 그릴 개수를 정하면 중심에 1개 + 정한 개수만큼 +/- 방향으로 생성합니다. 예를 들어, 현재 코드에서 x,y,z 축으로 각 10개씩 그리기로 하였다면 각 축을 따라 개씩 그리므로 개씩 배치하여 미리 저장합니다.

기존 따라하기 코드와 약간의 차이점으로는 x->y->z 축을 순회하며 생성하던 기존 코드와 달리 z->y->x 축을 순회하도록 하고, z축의 경우 0에서 시작하여 + 방향으로만 나아가도록 하여 플레이어 시작 지점 (월드의 (0.f, 500.f, -500.f) 위치) 에서 바로 한눈에 들어오도록 생성 순서를 변경하였습니다.

오브젝트들은 기본적으로 Mesh 등이 없이 사실상 비어있는 GameObject 객체를 생성하고 미리 로딩된 계층 모델을 SetChild() 를 이용하여 자식으로 한번에 설정됩니다. 이때 Scene에 생성된 오브젝트들이 하나의 계층 모델 GameObject 객체를 자식으로 공유하게 되면 문제가 생길 수 있습니다. 같은 GameObject 포인터를 여러 객체들이 참조하게 되면 Animate() 시에 동일한 프레임 객체에 대하여 작업을 수행하여 의도치 않은 결과가 초래됩니다. 예를 들어 헬리콥터 객체를 하나 만들고 3개의 오브젝트가 만들어진 하나의 헬리콥터 객체를 참조하게 된 상황을 가정하겠습니다. 기본적으로 헬리콥터의 메인 로터를 1초에 1바퀴 회전시킨다고 가정하면 아래같이 코드를 작성하게 될 것입니다.



m\_pMainRotorFrame 은 Initialize() 에서 해당 프레임을 FindFrame() 함수로 찾아 보관됩니다. 이때 같은 계층 모델 객체를 3개의 오브젝트 객체가 동시에 SetChild() 를 이용하여 자식으로 갖게된다면, 3개의 오브젝트가 가리키는 프레임 객체는 메모리 위치가 동일한 완전히 같은 객체를 참조하게 됩니다. 이렇게 되면 각 오브젝트에 대하여 Animate() 를 수행할 때, 하나의 동일한 프레임 객체를 3번 회전시키게 되므로 의도하던 결과와 다르게 1초에 3바퀴를 회전하는 결과를 불러오게 됩니다.

위 문제를 해결하기 위해서 GameObject::CopyObject() 라는 static 함수를 만들어 사용합니다. 함수의 구현은 아래와 같습니다.



이 함수의 핵심은 만들어진 Mesh 와 Material 은 동일한 객체를 참조하도록 하고, 깊은복사된 별개의GameObject 객체를 만들어 낸다는 부분에 있습니다. 이 부분은 Scene 내의 모든 오브젝트의 개별적인 움직임을 보장하면서, 렌더링을 위한 Mesh 와 Material 은 모두 공유하도록 한다는 점에 있습니다. 위 함수를 사용함으로서 얻는 이점은 다음 페이지에 있는 실행 화면에서 확인 가능합니다.

텍스트, 스크린샷, 항공기, 야외이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

계층 모델 GameObject 객체들은 Scene 의 BuildObject 도입부에서 한번에 읽어옵니다. 이때 읽어오는 과정에서 ResourceManager 라는 리소스 관리 클래스에 보관하도록 합니다. ResourceManager 는 GameFramework 에서 static 으로 선언하여 생성되는 클래스로 그 구조는 아래와 같습니다.

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

ResourceManager 를 이용하여 원본 객체는 1개만 보관하도록 합니다. 필요할 때에는 GetGameObject() 혹은 CopyGameObject() 함수에 원하는 오브젝트의 Key 값을 넘겨주면, 각각 원본/복사본을 리턴합니다. Key 값의 경우 읽어들인 파일 이름을 이용합니다. C++ 17 에서 추가된 <filesystem> 을 이용하면 아래 한 줄의 코드를 이용하여 파일 이름만 가져오는 것이 가능합니다.



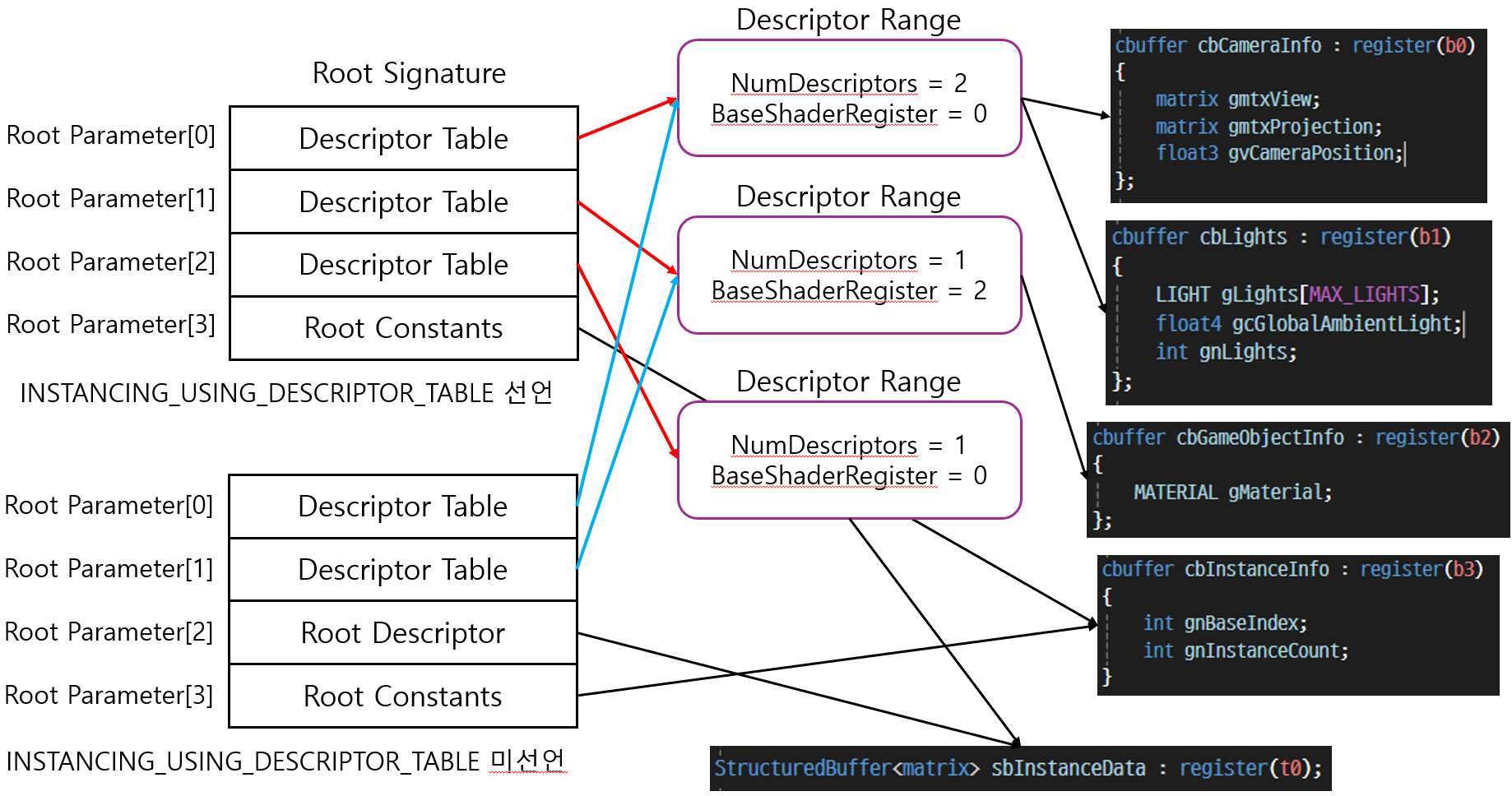
## **인스턴싱을 위한 과정**

아래 단락부터는 인스턴싱을 위한 클래스, 자료구조, 알고리즘을 설명합니다.

### **Root Signature**

Root Signature는 예제코드의 구조와 마찬가지로 Scene 에서 생성하고 관리합니다. Root Signature는 컴파일 문맥에 따라서 2가지로 구성됩니다. stdafx.h 에서 INSTANCING\_USING\_DESCRIPTOR\_TABLE 이라는 매크로를 #define 하면 인스턴싱을 위한 월드 행렬 정보를 담은 Structured Buffer 를 Descriptor Table 을 이용해서 GPU에 바인딩하고, 매크로를 선언하지 않으면 Root Descriptor 로 GPU에 바인딩합니다. 두 방법의 차이는 나중에 결과분석때 살펴보도록 하겠습니다.

두가지 Root Signature는 위 차이점만 존재하고 나머지는 동일합니다. 기본적으로 4개의 Root Parameter 로 구성되고 각 파라미터와 파라미터 타입은 아래 그림과 같습니다.



두가지 Root Signature 모두 4개의 루트 파라미터로 구성됩니다. 0번에서는 Descriptor Table 을 이용하여 Scene 별로 한번씩만 갱신이 필요한 카메라 정보와 조명 정보를 넘겨주고 1번에서는 오브젝트가 가진 Material 정보를 넘겨주도록 합니다.

2번은 위에서 설명한 바와 같이 매크로를 이용한 컴파일 분기에 따라서 인스턴싱 정보가 담긴 Structured Buffer 를 Descriptor Table 로 넘기거나 SetGraphicsRootShaderResourceView() 를 이용하여 Root Descriptor로 넘기도록 하고, 4번은 각 인스턴스가 행렬 Structured Buffer 에 접근하기 위해서 필요한 Index의 시작위치를 넘겨줍니다. 3번 파라미터의 경우 뒤에서 조금 더 자세하게 설명하겠습니다.

### **RenderManager**

이번 과제의 핵심인 RenderManager 는 렌더링할 오브젝트들을 Mesh 와 Material 정보를 이용하여 분류하고 인스턴싱 횟수를 카운팅하며, 최종적인 Draw 까지 담당하는 클래스입니다. 이 클래스 객체는 이전 ResourceManager 와 마찬가지로 GameFramework 에서 static 을 이용한 전역 객체로 선언되고 생성되어 사용합니다. 아래는 RenderManager 와 내부에서 사용하는 자료형들의 코드입니다.

텍스트, 스크린샷이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

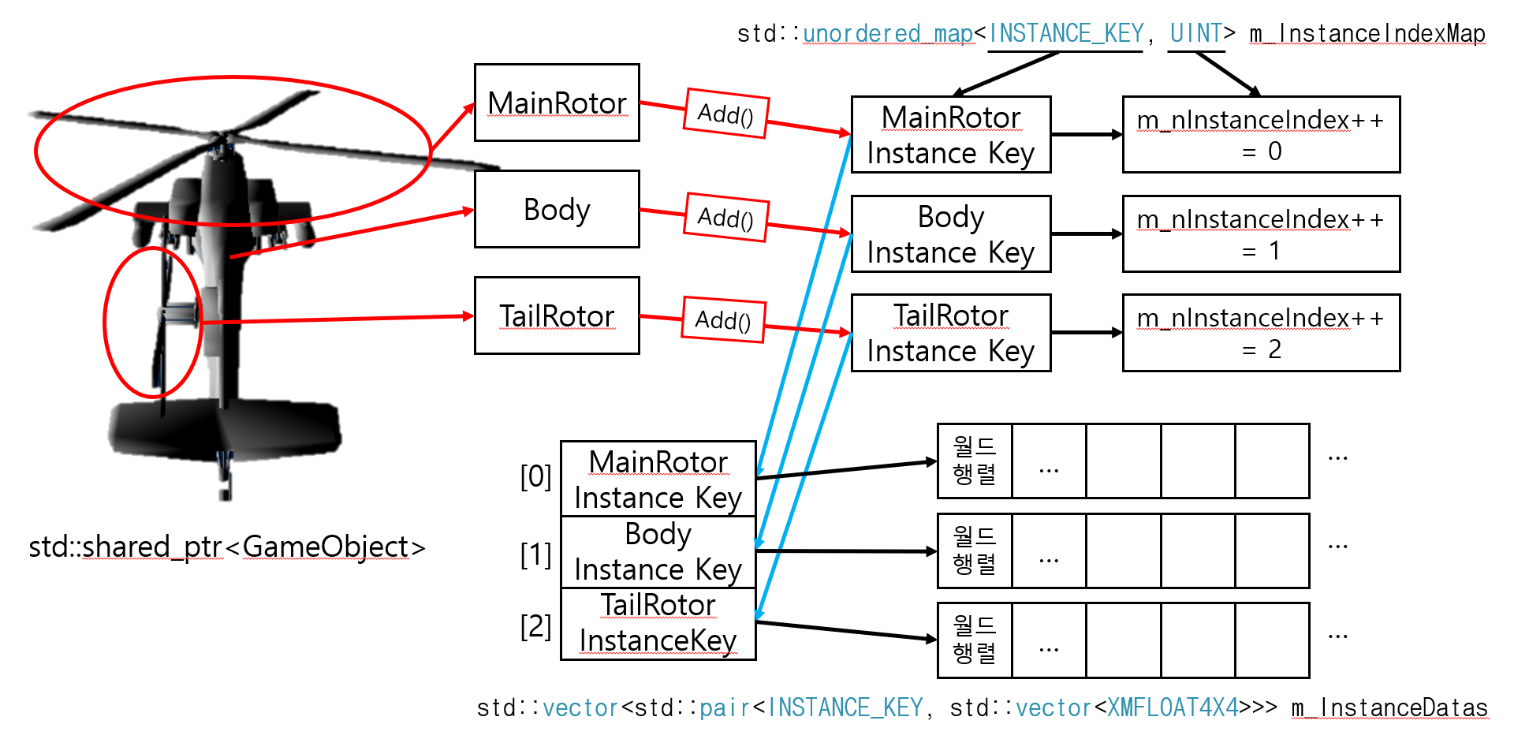
텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

기본적으로 Scene 안의 GameObject 들은 Scene::Render() 함수에서 각 객체가 바로 렌더링되지 않고 그리는데 필요한 리소스들만 RenderManager가 참조하도록 합니다. 그려질 모든 대상에 대하여 RenderManager에 보관하는 과정을 거치고 나면, RenderManger는 참조하고있는 리소스와 카운팅된 인스턴스 개수를 이용하여 동일한 Mesh 들을 인스턴싱하여 그리도록 합니다.

RenderManager가 객체를 보관하는 방식은 다음과 같습니다. 우선 RenderManager 에 객체를 집어넣게 되면(Add()) Mesh와 Material 의 배열(현재는 std::vector)을 가져와 고유한 INSTANCE\_KEY 구조체를 만듭니다. 이 구조체로 먼저 m\_InstanceIndexMap 이라는 해시맵 컨테이너에 해당 INSTANCE\_KEY가 있는지 확인합니다. 만약 없다면 m\_InstnaceIndexMap 에 key 를 추가하고 m\_nInstanceIndex 값을 이용하여 std::vector 컨테이너인 m\_InstanceDatas 에서 사용할 인덱스를 결정해줍니다. 그러고 나면 이 컨테이너는 INSTANCE\_KEY 와 월드 변환 행렬 배열의 쌍으로 이루어집니다. m\_InstanceIndexMap 에서 현재 추가하려는 INSTANCE\_KEY 가 vector 의 몇번째 인덱스에 보관되어 있는지 알 수 있고 해당 인덱스에 접근하여 월드 변환 행렬을 추가합니다.

아래 그림(다음 페이지) 에서는 위에서 설명한 Add() 함수의 동작을 나타내고 있는 그림과 실제 코드입니다.



텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

Scene 에 있는 그려질 모든 GameObject 에 대하여 RenderManager에 집어넣는 작업이 완료되고 나면 그려야 할 차례입니다. 이 작업은 RenderManager::Render() 함수에서 일어나게 됩니다.

RenderManager 안에는 GPU에 바인딩 할 수 있도록 Flags 를 SHADER\_VISIBLE 로 하여 생성된 Descriptor Heap 이 있습니다. 한 프레임에서 Descriptor Table 을 이용해 넘어가게 될 모든 리소스들은 이 Descriptor Heap 에 기록됩니다. Descriptor Heap은 매 프레임 마다 아래처럼 구성됩니다.

텍스트, 도표, 폰트, 라인이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

그림처럼 DescriptorHeap 에 한프레임에 필요한 정보를 많이 담기 위해서는 NumDescriptors 개수가 충분해야 합니다. 그러나 런타임에 그릴 모델을 바꿀 수 있는 현재 상황에서 필요한 Descriptor 의 개수를 정확하게 알기는 어렵습니다. 또한 모델이 바뀔 때 마다 필요한 Descriptor 의 개수도 달라지므로 충분한 개수의 Descriptor 를 미리 만들어 두고 사용하기로 결정하였습니다.

GPU에 Descriptor Heap 으로 넘겨줄 정보를 들고 있는 객체들은 내부적으로 ConstantBuffer 를 가지고 있습니다. ConstantBuffer 클래스는 상수 버퍼를 사용하기 쉽도록 한번 wrapping 한 클래스입니다.

각 ConstantBuffer 들의 Constant Buffer View 를 담을 Descriptor Heap 의 Flag 를 SHADER\_VISIBLE 하게 만들지 않습니다. 그러므로 파이프라인에 Descriptor 를 바로 바인딩 할 수 없고, RenderManager 에 있는 SHADER\_VISIBLE 한 Descriptor Heap 의 원하는 위치에 CopyDescriptorSimple() 함수를 이용하여 복사하여 전달하도록 하였습니다. 복사에서 오는 시간 손실이 있을 것임에도 불구하고 이 방법을 사용한 이유는 파이프라인에는 같은 종류의 Descriptor Heap 을 여러 개 걸 수 없기 때문에 하나의 Descriptor Heap 안에 한 프레임에 그릴 리소스들을 전부 담아주면서 좀 더 예측 가능한 형태로 구성하여 리소스 바인딩에 문제가 생길 시에 해결하기 좀 더 용이하게 하기 위한 이유도 있습니다.

다음 페이지부터는 실제 코드와 함께 렌더링을 위한 리소스 바인딩 과정을 설명합니다.

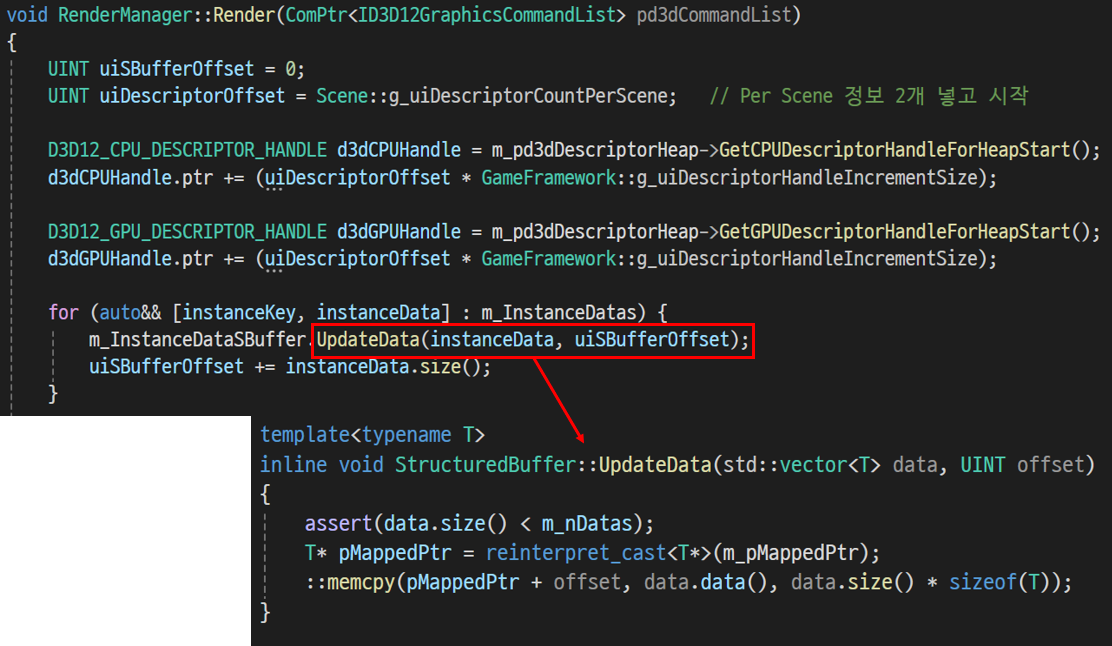


설명하기 앞서 코드에 있는 RENDER 매크로는 GameFramework::g\_pRenderManager 를 매크로로 만든 것으로 전역 static 객체인 RenderManager 를 조금 더 편하게 접근할 수 있도록 만들어 두었습니다. 매크로 선언은 GameFramework.h 의 GameFramework 클래스 선언 아래부분에서 찾을 수 있습니다.

가장 먼저 리소스 바인딩이 이루어지는 부분인 Scene::Render() 부분부터 봐야합니다. 이 함수에서는 오브젝트들을 RenderManager 로 밀어넣는 역할도 하지만 RenderManager 안에 있는 Descriptor Heap 을 가져와 Scene 의 정보들을 복사하고 파이프라인에 먼저 Set 해주게 됩니다.

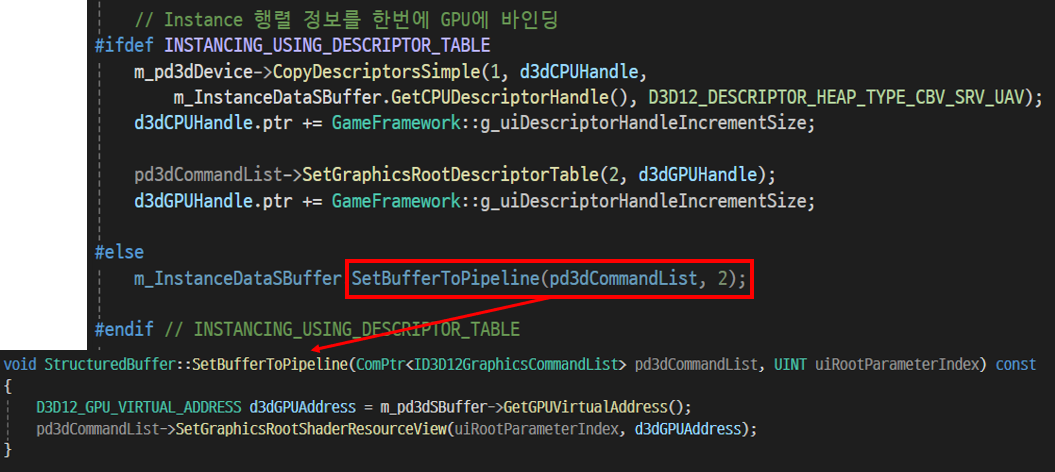
이후 Descriptor Heap 의 원하는 위치에 복사가 이루어지게 하기 위해서 CPU 핸들을 가져와 ptr 에 Descriptor 의 크기를 더해 위치를 이동시키며 복사하였습니다. 2개의 Descriptor 를 위 페이지에 있는 구성도 처럼 Heap 의 앞부분에 복사시켜 주고 Root Parameter 0번에 바인딩 시켜주어 조명 정보와 카메라 정보를 GPU에 바인딩 되도록 하였습니다.

Scene 의 정보들에 대한 바인딩이 끝나고 나서 Scene::Render 가 리턴하면 GameFramework::Render() 에서 RENDER->Render() 함수를 호출하여 본격적인 렌더링을 위한 작업을 시작합니다. 이 함수는 총 3단계로 나누어 설명하겠습니다.



RenderManager::Render() 의 함수 초입에서 가장 먼저 하는 일은 GPU에 바인딩 될 Descriptor Heap 의 CPU/GPU 시작 핸들을 가져와 Scene에서 이미 복사한 개수만큼 포인터를 이동시키고 시작합니다. 현재 Scene 에서 2개의 Descriptor 를 복사하였으므로 모두 2칸씩 이동하여 3번째 칸부터 원하는 Descriptor 가 복사될 수 있도록 합니다. 이때 CPU 핸들은 Descriptor 복사에 사용되고 GPU 핸들은 Descriptor Table 을 GPU에 바인딩할 때 사용됩니다.

그 다음으로는 m\_InstanceDatas 에 보관된 행렬들을 순서대로 Structured Buffer 에 기록합니다. 기록하는 함수는 아래에서 조금 더 세부적으로 다루도록 하겠습니다. 현재 사용중인 함수를 이용하면 std::vector 로 된 리소스들을 memcpy 함수를 이용하여 ID3D12Resource 에 Map 된 포인터에다가 한번에 복사합니다.



다음은 인스턴스 정보가 기록된 Structured Buffer 를 GPU에 바인딩합니다. 앞서 Root Signature 설명에서도 언급되었던 INSTANCING\_USING\_DESCRIPTOR\_TABLE 매크로의 선언 여부에 따라서 바인딩 방법이 달라지게 됩니다.

우선 Descriptor Table 로 넘길 때는 이전 Scene 데이터에 넘길때와 마찬가지로 RenderManager 의 Descriptor Heap 에다가 CopyDescriptorSimple() 함수를 이용하여 Descriptor 를 복사한 뒤 리소스를 바인딩합니다. Scene 정보와 달리 리소스가 Descriptor Heap 의 중간에 위치하고 있으므로 GPU 핸들의 포인터를 복사된 위치로 이동시켜 GPU에 바인딩합니다.

Root Descriptor 로 넘길때는 SetGraphicsRootShaderResourceView() 함수를 이용하여 원하는 Root Parameter 자리에 바로 리소스를 바인딩합니다. 위 코드에서 SetBufferToPipeline() 함수에서 위 동작을 수행합니다.



텍스트, 라인, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

Render 의 마지막 단계입니다. 보관해둔 INSTANCE\_KEY 를 순회하면서 Material을 Descriptor Heap 에 복사하고 복사된 위치를 Set 한 다음 인스턴싱할 개수만큼 Mesh 를 그려주도록 합니다.

이전 월드 행렬들을 Structured Buffer 에 복사하여 Set 하는 과정에서 인스턴스 구분 없이 보관된 순서대로 순차적으로 행렬을 복사하여 저장하였기 때문에 그리기 전에 해당 인스턴스의 행렬이 Structured Buffer 의 몇번째 인덱스에 해당하는지도 함께 전달해주어야 합니다. 이는 0부터 시작하여 각 인스턴스의 개수를 더해가면서 계산할 수 있고, 계산된 값을 Root Parameter 3번에 있는 Root Constant 상수 버퍼에 바인딩해줍니다. 정점 쉐이더 에서는 SV\_InstanceID 로 넘어온 값과 Root Constant 로 넘어온 값을 더하여 해당 인스턴스의 월드 행렬이 보관된 위치를 알 수 있습니다.

위 까지의 내용이 RenderManager 를 이용하여 인스턴싱을 수행하는 과정입니다. 다음으로는 인스턴싱을 통해 예제 코드와 제공된 계층 모델들을 얼마나 그릴 수 있었는지 확인해본 결과를 설명합니다.

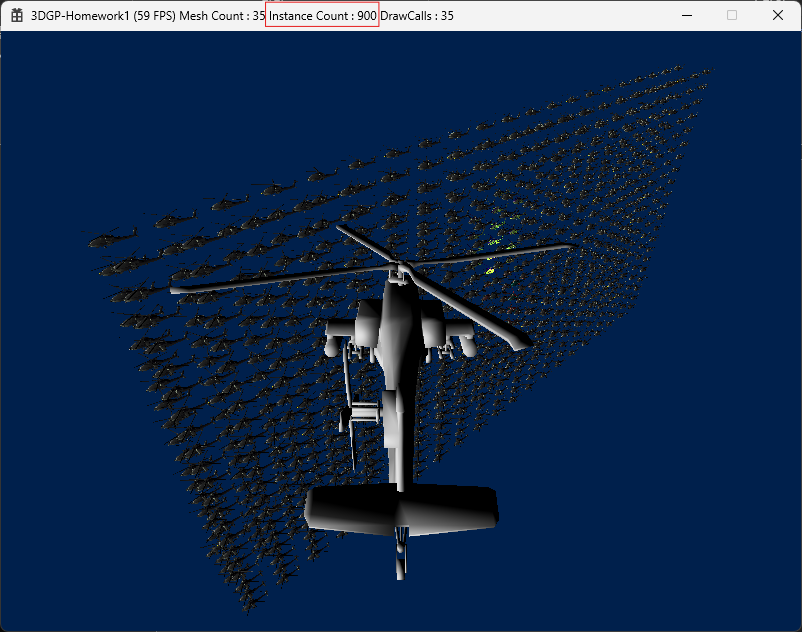
# **결과 분석**

우선 테스트 환경 PC 사양입니다.

* CPU : 13th Gen Intel Core I9-13900H
* GPU : NVIDIA GeForce RTX 4070 Laptop GPU

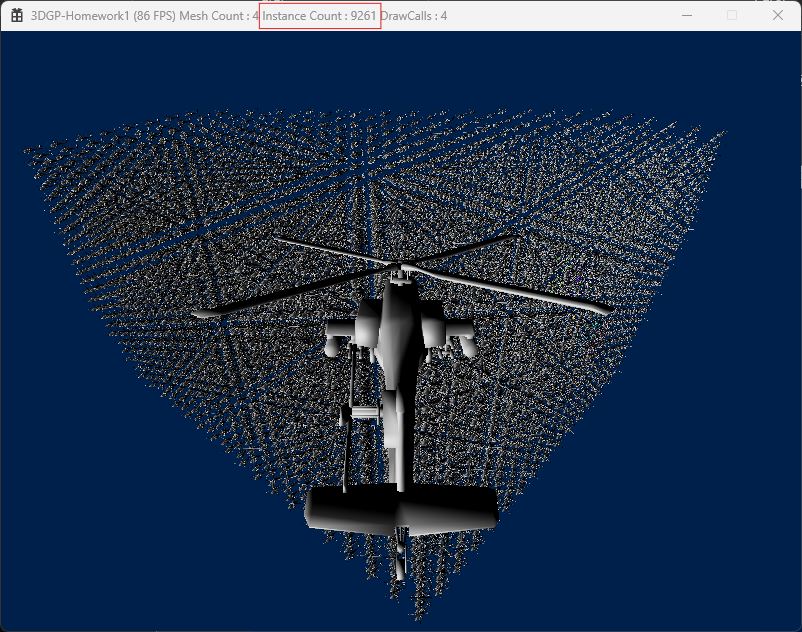
테스트는 가능한 클리핑 되는 정점이 없도록 화면 안에 모든 오브젝트들이 최대한 들어올 수 있도록 한 뒤 테스트하였습니다. 또한 오브젝트들을 단순 그리기만 하는 것이 아닌 애니메이션을 위한 업데이트 과정도 인스턴스 갯수만큼 거치기 때문에 이점도 함께 고려해야 합니다. 이 부분은 현실적으로 게임이 동작할때도 모든 인스턴스가 공유되는 하나의 오브젝트의 업데이트에 의존하지 않고 각각이 독립적으로 움직이는 상황이 더욱 많을 것이라고 생각해 좀 더 현실적인 시나리오로 진행하기로 하였습니다.

목표 프레임 레이트는 60FPS 입니다.



첫번째는 Apache 입니다. 이 모델의 경우 900개를 넘어가기 시작하였을때부터 60프레임 이하로 조금씩 내려가기 시작하였습니다. 이어 930개를 넘어가면 60프레임 달성이 어려워져 안정적인 60프레임 유지에는 900개 정도가 적당하였습니다.

이 모델은 제공된 모델 중에서 가장 프레임 레이트의 하락이 가파르게 오는 모델이었습니다. 그 이유를 추측해보자면 모델을 구성하는 몇몇 Mesh 들의 정점수가 다른 모델들에 비해 매우 많은 것이 문제인 것으로 보입니다. 다른 모델의 경우 인덱스 버퍼의 크기가 10000개를 넘는 Mesh가 많아야 한두개 정도 있는 반면에 Apache 의 경우 많게는 90000개가 넘는 인덱스 버퍼로 구성된 Mesh가 있어 상대적으로 렌더링에 걸리는 시간이 긴 것 같다고 생각합니다.



두번째는 Gunship입니다. 이 모델은 플레이어의 모델과도 동일하여 Draw Call 수를 조금 더 줄이는 것이 가능했습니다. 또한 4개의 Mesh 만으로 이루어진데다 정점 수도 적고, SubMesh 도 없는 단순한 모델이기 때문에 생성시켜놓은 9261개를 모두 그려도 60FPS 를 훨씬 상회하는 프레임레이트를 보여주었습니다.

안테나, 스크린샷, 야외, 하늘이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

세번째는 SuperCobra입니다. 이 모델은 대략 2100~2200 개의 인스턴스 사이에서 60FPS 정도의 프레임 레이트를 보여주었습니다. 몇 개의 Mesh 들이 2개 이상의 SubMesh로 구성되어 있지만 정점 수가 Apache 보다는 많지는 않아 2배 이상 많은 모델을 렌더링 가능했습니다.

스크린샷, 텍스트, 안테나, 야외이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

네번째는 Hummer입니다. 이 모델은 1200개를 넘어가면 프레임레이트가 가파르게 떨어지는 모습을 보였습니다. 제공된 모델 중에서 가장 Mesh의 수가 많고, 몇 개는 SubMesh로 나누어져 있어 하나의 모델을 위해 가장 많은 Draw Call 이 호출되지만 정점 수가 Apache 보다는 많지 않아 조금 더 많이 그릴 수 있었다고 생각합니다.

스크린샷, 안테나, 야외이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

다섯번째는 M26 입니다. 이 모델은 SuperCobra 와 유사하게 2100 ~ 2200 개의 인스턴스 사이에서 60FPS 정도의 프레임 레이트를 보여주었습니다. Mesh 들의 특징도 SuperCobra 와 유사했습니다.

스크린샷, 텍스트, 야외, 안테나이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

마지막은 Mi24 입니다. 이 모델의 경우 9261개를 전부 다 그리도록 하였을 때 60FPS 유지는 약간 어렵지만 대략 55FPS 정도는 유지를 하는 모델이었습니다. 이는 Mesh의 개수가 그리 많지 않으면서 정점 수도 다른 모델들에 비하면 상당히 적기 때문에 가능했다고 생각합니다.