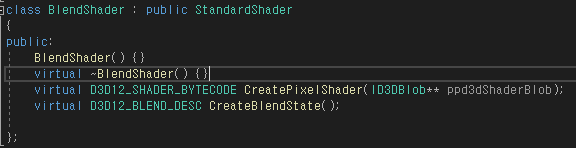
**2018180020 박재우(클라이언트) 13주차 기록**

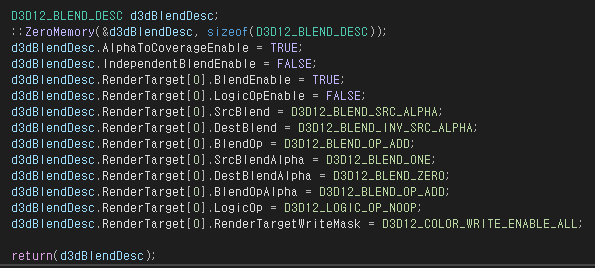
**구조물 띄우기**

유니티에서 추출한 맵 데이터를 이용해 화면에 그리는 작업을 했다.

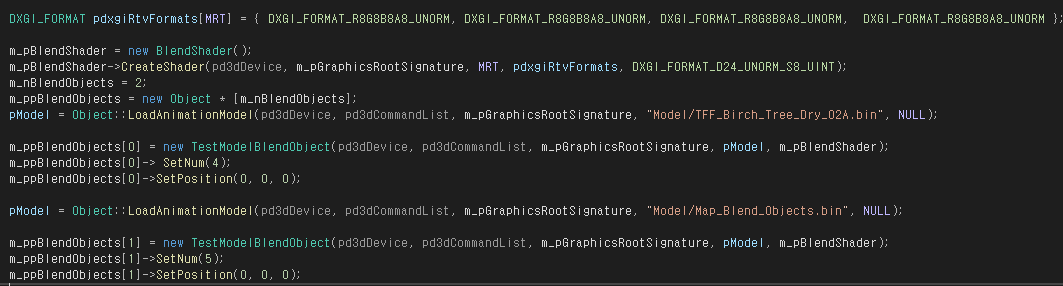
일반 오브젝트들은 상관없지만 나뭇잎 같은 경우 알파블렌딩을 해줄 필요가 있다. 하지만 우리는 디퍼드 렌더링을 이용해 그리고 있기 때문에 알파블렌딩을 사용할 수 없다. 해결 방법으로는 블렌딩이 필요한 오브젝트들은 모든 렌더링이 끝난 후 마지막에 그려주면 된다.



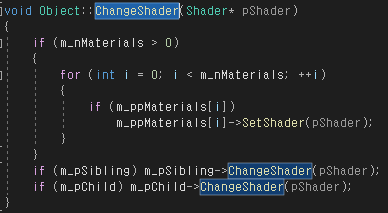
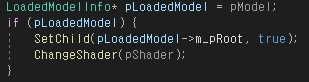
먼저 블렌딩을 위한 쉐이더 객체를 만들어준다. StandardShader에서 PixelShader와 BlendState만 변경되므로 해당 함수들을 선언해준다.



BlendState는 위와 같이 설정해 주어 알파 블렌딩이 가능하게 해준다.



이제 BlendShader를 만들고 오브젝트들을 생성해준다. 오브젝트 파일을 읽는 함수는 LoadAnimationModle 함수로 잘 읽어올 수 있지만 쉐이더는 StandardShader로 만들어진다. 따라서 모든 오브젝트의 쉐이더를 BlendShader로 바꿔줄 필요가 있다.

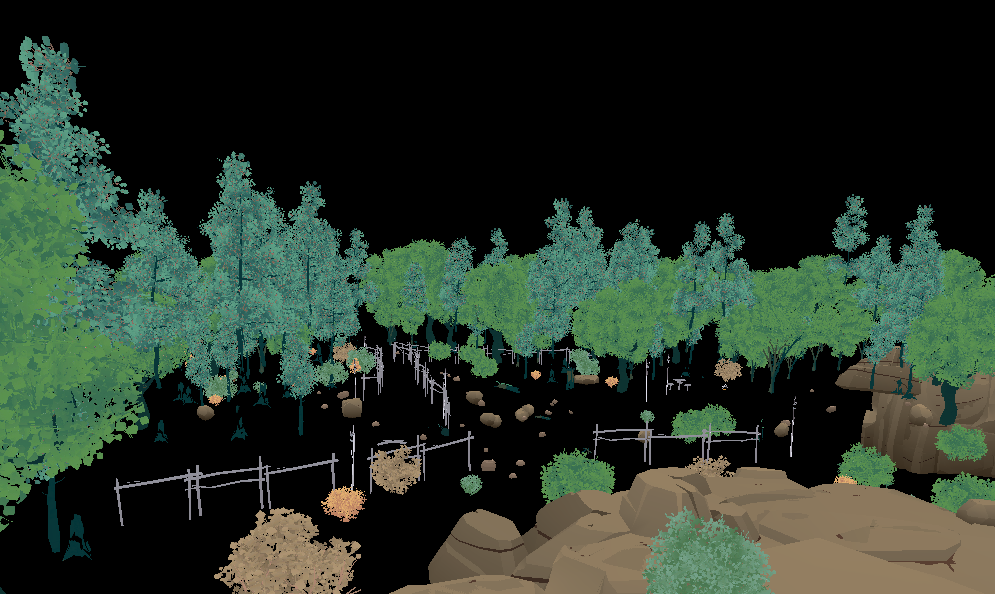


생성자와 ChangeShader함수이다. 모든 쉐이더들을 입력으로 들어온 pShader로 바꿔준다.

이렇게 생성한 오브젝트들은 같은 Render함수에서 그릴 수 없다.



RenderBlend 함수를 만들어서 모든 렌더링 이후에 그리도록 만들었다.



나뭇잎 등등이 알파 블렌딩이 되어 잘 그려 지기는 하나 렉이 심하다. 인스턴싱을 해야 하고 나무 줄기가 초록색이다. 이 문제도 원인을 찾아서 고쳐야 한다.

**맵 추출 CS 파일 제작**

맵을 추출하여 뽑을 때 일반 모델을 뽑는 방법으로 하면 맵의 모든 오브젝트가 하나의 통 Mesh로 뽑히기 때문에 최적화 면에서 굉장히 비 효율적이다. 하나의 Mesh로 그리게 되면 거리에 따라 렌더링을 하지 않는 방법이 통하지 않고 무엇보다 맵의 파일이 매우 커진다. 지금 불러온 맵만 bin파일로 900MB 이상이기 때문에 저 정도의 맵이 4개가 필요한 우리 게임에서는 4GB 정도의 크기가 될 것이다.

새롭게 맵을 추출할 방법은 맵에 배치된 오브젝트가 사용하는 Mesh의 이름과 그 오브젝트가 배치 되어있는 위치 데이터만 저장하도록 한다. 배치된 오브젝트의 모델은 각각 따로 뽑아 두고 우리 게임에서 로딩 할 때 해당되는 모델의 파일을 읽어 생성하여 맵 오브젝트 리스트로 관리한다. 중복되는 모델은 이미 한번 로드를 했으므로 재사용하여 로딩 시간을 많이 단축할 수 있을 것이라 생각된다. 오브젝트 리스트로 각 오브젝트를 따로 처리를 해줄 수 있기 때문에 컬링을 좀 더 쉽게 할 수 있을 것이다.

저장할 파일의 구조를 정해 보자. 저장할 데이터는 Mesh의 이름, 위치이다. Texture, Material등의 데이터는 모델의 파일에 저장이 될 것이므로 Mesh의 이름과 위치 데이터만 있으면 된다.

시작 문구로 <Objects>: 로 시작한다.

<Mesh>: 이후 이름을 적고 <Position>: 이후 위치를 기록한다. 위치는 Scale Rotation을 모두 포함한 4X4 행렬로 표현한다. 이미 앞에서 같은 이름의 Mesh가 저장되었다면 앞에 @를 붙여 중복이라는 사실을 표시한다.

마지막에는 </Objects>: 로 파일에 저장된 모든 오브젝트를 읽었다는 표시를 해 루프를 빠져나갈 수 있도록 한다.

파일의 최종 구조는 다음과 같다.

<Objects>:

<Mesh>: Mesh\_Name1

<Position>: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

<Mesh>: @Mesh\_Name1

<Position>: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

<Mesh>: Mesh\_Name2

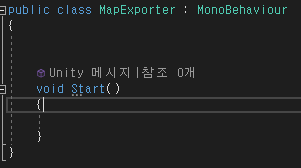
<Position>: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

</Objects>:

<Mesh>: 와 <Position>: 은 오브젝트 개수만큼 반복된다.

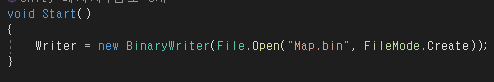
파일을 읽을 때 <Mesh>: 에서 이름을 읽어 모델의 데이터를 리스트로 저장하여 중복이 있을 때를 대비한다. 앞에 @가 붙은 이름은 모델의 리스트를 확인해 같은 이름의 모델을 SetChild로 새로 생성한 오브젝트에 붙인다. 파일을 읽는 작업이 모두 끝나면 모델을 저장한 리스트는 delete로 지워준다. LoadedModelInfo 클래스엔 포인터 변수가 저장되어있어 실제로 필요한 데이터는 지워지지 않을 것이다.

만든 cs 파일을 사용하기 위해 클래스를 생성한다.

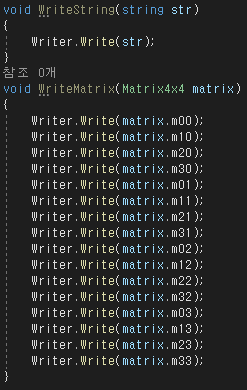


이 cs 파일을 컴포넌트로 오브젝트에 넣고 실행하면 Start() 함수가 실행된다. Start함수에서 파일을 열고 쓰기를 시작한다.

Binary 파일을 만들기 위해 BinaryWriter라는 클래스를 사용한다. 해당 객체를 멤버 변수로 만들어주고 Start함수에서 파일을 생성해서 연다.

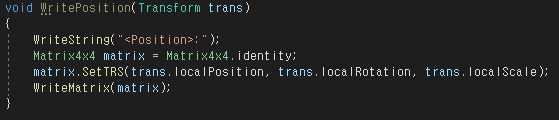


데이터는 String, 4X4Matrix 두가지 데이터만 저장된다. 앞으로의 편의를 위해 이 두가지 데이터를 파일에 쓰는 함수를 만든다.

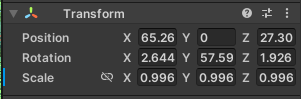


BinaryWriter의 Write 함수는 인자로 들어온 값의 자료형에 맞게 파일에 쓴다. Matrix는 float 데이터 16개로 이루어져 있고 각각의 값을 파일에 써준다.

다음은 <Position>: 부분을 저장하는 함수를 만든다. <Position>: 를 적고 행렬을 저장한다. Transform Rotate Scale 값으로 이루어진 행렬을 만들고 위에서 만든 WriteMatrix 함수를 이용한다.



인자로 들어오는 Transform 값은 유니티 오브젝트의 컴포넌트이다.



Transform 컴포넌트를 이용해 행렬을 만들어 줄 수 있다.

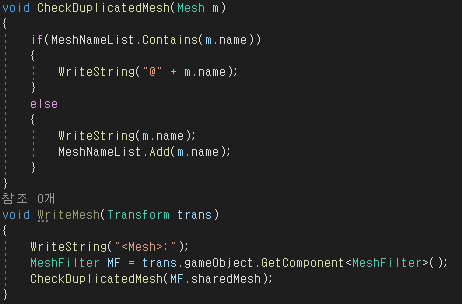
다음은 <Mesh>: 부분을 저장하는 함수를 만든다.

<Mesh>: 다음으로 Mesh의 이름을 받아와야 하는데 Mesh의 이름은 MeshFilter라는 컴포넌트에서 얻을 수 있다. MeshFilter에 접근하기 위해

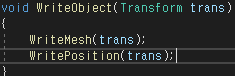
Transform.gameobject.GetComponente<>()를 이용한다.

Mesh의 이름이 중복이 되면 @를 붙여서 저장해야 하기 때문에 List로 이름들을 저장하여 중복을 확인한다.

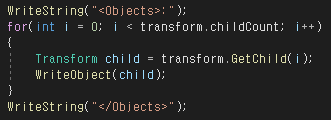




<Mesh>:를 적고 MeshFilter를 이용해 Mesh의 중복여부를 CheckDuplicatedMesh 함수로 확인해 파일에 적어준다.



Mesh와 Position을 적는 함수를 호출해줄 WirteObject함수를 만들었고 이 함수에 Transform 컴포넌트를 넘겨준다.



Start함수에서 <Objects>:를 적어준다. 현재 오브젝트는 모든 오브젝트의 최상위 객체이기 때문에 Mesh도 없고 원점에 박혀 있다. 따라서 현재 오브젝트의 모든 Child를 WriteObject로 저장한다. Transform.childCount를 이용해 for문을 돌리고 GetChild로 Child의 Transform 컴포넌트를 WriteObject함수로 넘겨준다. for문이 끝나면 </Objects>로 파일이 끝났음을 알려준다. Writer.Close로 파일을 닫아주면 맵을 로드하는 작업은 끝이 난다.