3D 프로그래밍1

과제 02

게임공학과

2013180003 김나단

1. 이번 과제의 목표

이번 과제에서는 따라하기 14번을 바탕으로 과제 1의 내용을 동일하게 구현하는 것을 목표로 잡고 진행하였으며, 추가적으로 인스턴싱을 사용한 파티클 구현을 추가적으로 목표로 잡고 진행하였습니다.

1. 프로그램에 대한 가정

이번 과제는 과제 1과 마찬가지의 과제이므로 저번과 동일하게 플레이어가 앞, 뒤 어느 방향으로도 이동이 가능하며, 이동시에 위화감(적이 나오지 않거나 뒤에서 나오는 등)이 느껴지지 않도록 플레이어의 이동 방향에 따라 장애물 오브젝트가 생성되도록 하였습니다. 플레이어가 벽에 부딪히는 경우에도 저번과 동일하게 벽에서 반사벡터를 얻어 플레이어의 속도를 그 반사벡터에 투영하여 플레이어의 속도에 더하여 벽을 뚫지 않도록 하였습니다.

벽에 대해서는 여러 개의 벽을 사용하는데 저번과는 다르게 색이 칠해져 있기에 점점 색이 변하는 것을 생각하고 두가지 벽 모델을 사용하여 연속적으로 느껴지도록 하였으며, 벽 모델의 개수도 지난번 50개에서 8개로 줄일 수 있었습니다.

가장 큰 차이를 보인 것이 장애물 오브젝트와 파티클인데, 먼저 장애물 오브젝트는 따라하기 14의 오브젝트 쉐이더를 수정하여 사용하였으며, 파티클 또한 따라하기의 쉐이더를 바탕으로 쉐이더를 작성하여 사용하였습니다. 결과적으로 장애물 오브젝트는 80개로, 파티클 오브젝트는 50개에서 200개로 크게 증가하였으며, 그럼에도 프레임 레이트는 Release 모드 기준 400FPS 이상이 나와 다이렉트 파이프라인과 인스턴싱의 영향력을 잘 느낄 수 있는 과제였던 것 같습니다.

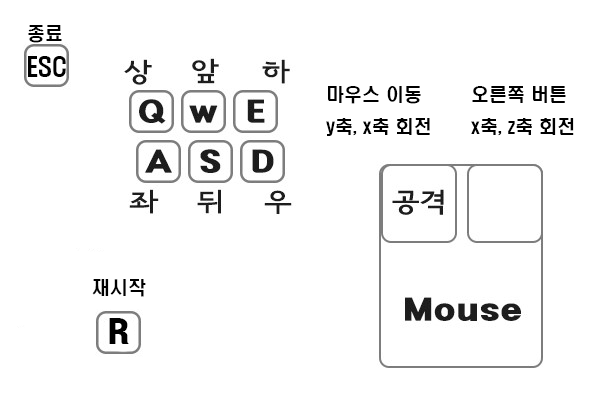
1. 프로그램의 변경 점

먼저, 선으로 표현되던 오브젝트들을 모두 색상이 있는 면으로 표현하게 되었으며,

다이렉트 파이프라인 등을 사용하여 성능이 크게 향상되었습니다.

장애물 오브젝트와 파티클에 대해서는 인스턴싱을 적용하였으며, 총알에도 인스턴싱을 사용하고자 하였으나 실제 인스턴싱을 적용했을 때 잦은 Reset 호출로 인해 프레임 레이트가 오히려 낮아져 인스턴싱 하지 않고 메쉬를 저장해 놓고 이 메쉬를 포인터로 지정하여 사용하였습니다. 또, 총알 메쉬에 대해서 6면체 대신 4면체를 사용하여 파티클과 구별할 수 있도록 하였습니다.

1. 조작 방법



-이전 과제에서 존재했던 부스터를 제거하였는데 이유는 큰 의미가 없어서 입니다.

1. 코드 설명

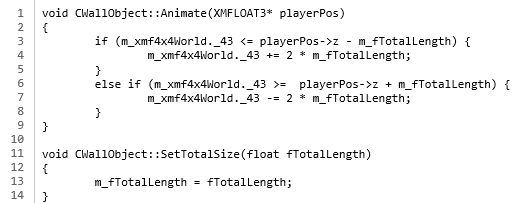
지난번 과제에서 달라진 것만 서술하였습니다.

1. Camera

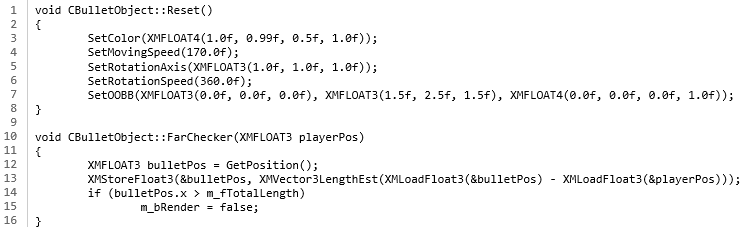
먼저, 플레이어가 비행기 이기 때문에 SpaceShipCamera를 이용하였습니다. 속도감을 위하여 TimeLag값을 부여하였으며, 3인칭 카메라와 동일하게 update를 추가하여 TimeLag가 적용되도록 하였으며, 기존 Rotate함수는 제거하였습니다.

1. GameObject

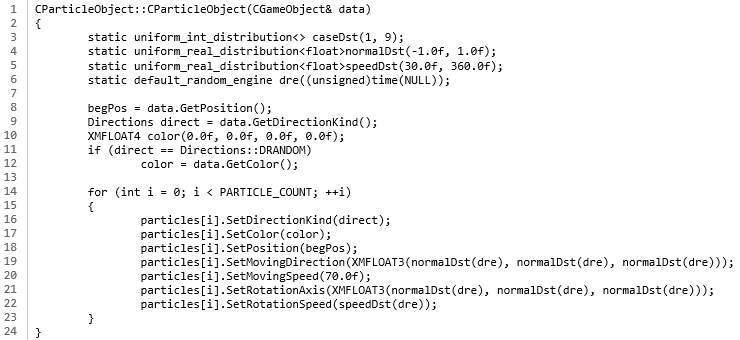
RotatingObject를 상속받는 BulletObject와 ParticleObject를 각각 정의 하였으며, GameObject를 상속받는 WallObject를 정의하여 사용하였습니다.



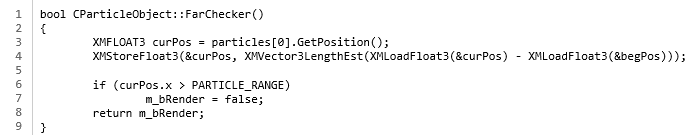
WallObject의 경우 벽의 Animate에서 플레이어의 위치에 따라 벽의 위치를 재 설정하기 위하여 사용하였으며 벽의 총 길이의 절반인 totalLength를 저장하여 벽의 생성 위치를 설정하였습니다.



Bullet의 경우 Reset 함수를 만들어 총알의 기본 설정을 쉽게 처리 할 수 있도록 하였으며, FarChecker를 매 Animate 마다 실행하여 플레이어와의 거리가 멀어질 경우 List에서 제거하였습니다.



파티클 생성시 생성 위치, 색상 등을 결정하기 위하여 게임 오브젝트로 오브젝트를 받아와 데이터를 저장하였습니다. 그리고 터지기 시작한 위치를 저장하여 FarCheck 함수를 호출하여 파티클 제거 조건을 부여하였습니다.

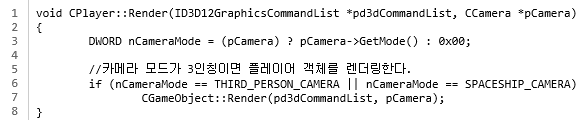


1. Mesh

벽, 총알에서 사용하기 위한 CWallMeshDiffused와 CBulletMeshDiffused를 추가로 정의하였습니다.

1. Player

플레이어와 벽의 충돌처리를 하기위한 CollisionWalls 함수를 추가하였습니다. 내부적인 충돌 검사는 기존 과제와 동일합니다.



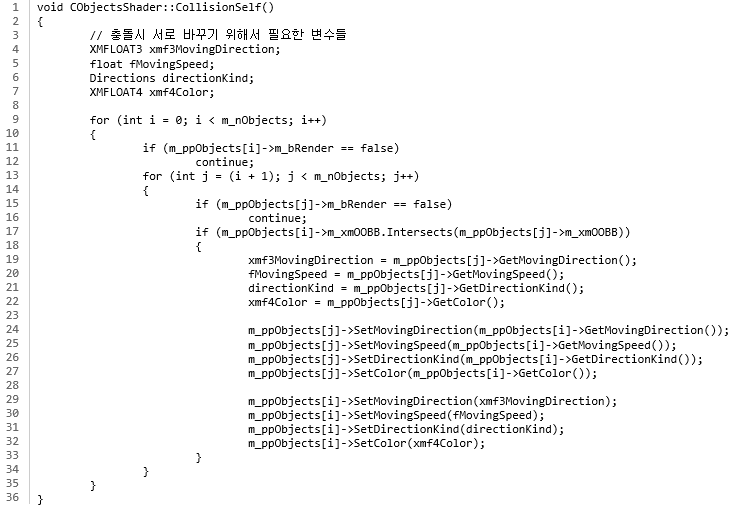
플레이어 Render 조건에 SpaceShipCamera를 추가하여 비행기 객체가 보이도록 하였습니다.

1. Shader

ObjectShader 클래스에 플레이어와 벽의 충돌을 제외한 충돌 처리를 추가하여 충돌처리를 하도록 하였으며, 쉐이더가 충돌처리를 해야 할 대상 중 가장 많은 데이터(장애물 오브젝트)를 가지고 있기 때문에 쉐이더에서 충돌처리를 하게 되었습니다.

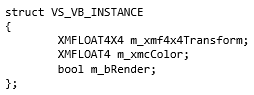
총 4종류의 충돌관련 함수를 추가하였으며, 함수 명은 CollisionWalls, CollisionPlayer, CollisionBullets, CollisionSelf이며, 순서대로 장애물과 벽, 장애물과 플레이어, 장애물과 총알, 장애물과 장애물의 충돌처리를 담당하고 있습니다.

기본적인 충돌 처리 방법은 기존 과제의 충돌처리와 동일합니다.

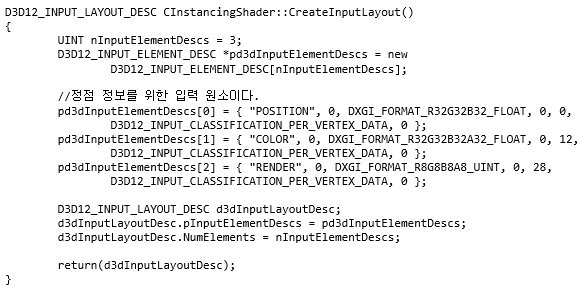


장애물과 장애물의 충돌처리는 기존에 충돌한 물체를 저장해 두고 다시 루프를 돌면서 데이터를 교환하는 것에서 현재 충돌을 확인하면 그 즉시 두 물체의 데이터를 교환하는 방식으로 변경하였습니다. 결과적으로 기존의 Collider 변수를 사용하지 않게 되었습니다.

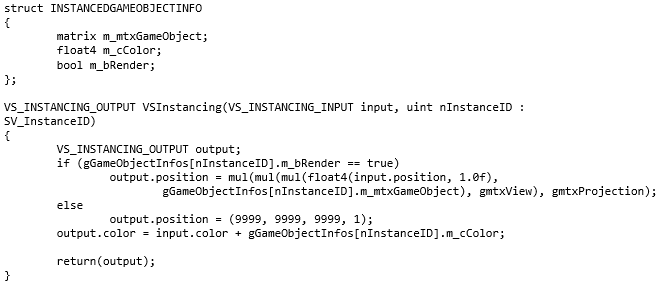
충돌 이후 그려지면 안되는 장애물 오브젝트를 파악하기 위하여 InstancingShader를 수정하여 Bool 변수를 Shader.hlsl 의 VSInstancing 으로 넘겨줄 수 있게 하였습니다.



먼저, HLSL에 넘겨줄 구조체를 수정한 뒤,



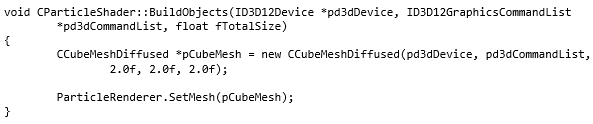
CreateInputLayout에서 InputElementDescs를 늘리고 값을 넘겨줄 수 있도록 메모리를 적용하였습니다.



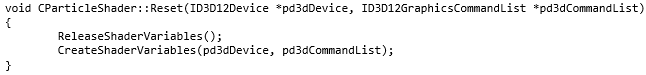
이후 Shader.hlsl 에 있는 구조체도 수정하고 VSInstancing에서 받아온 Boolean 값이 False인경우( 화면에 보이면 안되는 경우) 그 객체의 그리는 위치를 카메라의 범위 밖으로 옮겨 화면에 보이지 않도록 하였습니다.

처음에는 색상을 없애도록 하였으나 그냥 검은색 혹은 흰색으로 그려지고 투명해 지지는 않아 그냥 안보이도록 옮기게 되었습니다.

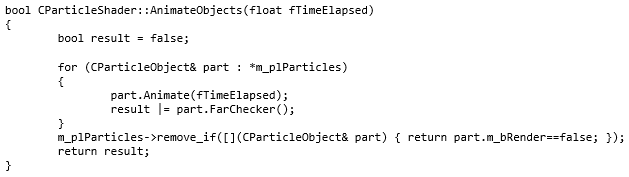
Particle의 인스턴싱을 위하여 CInstancingShader를 상속받는CParticleShader를 추가하였습니다. 저는 파티클을 그리기 위하여 List를 사용하였으며, 그로 인해 전체 크기가 변경되므로 BuildObject에서 버퍼를 생성하지 않고, List의 크기가 변경 될 때 마다 버퍼를 제거했다가 삭제하는 방식으로 진행하였습니다.



그렇기에 BuildObject 에서는 인스턴싱을 위한 객체를 하나만 초기화 합니다.



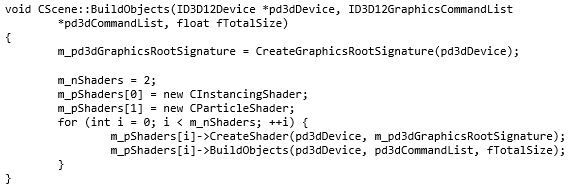
버퍼를 지우고 생성하는 것을 묶은 Reset 함수를 만들어 사용하였습니다.



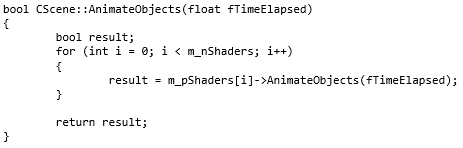
Reset 회수를 줄이기 위하여 Reset이 필요한 경우 True를 리턴하여 Scene에서 이 Boolean 값을 받아 True인 경우 Reset을 호출하도록 하였습니다.

1. Scene

ParticleShader를 추가하기 위하여 pShaders를 CInstancingShader의 포인터를 저장하는 배열로 변경하고 개수를 2개로 늘렸습니다.

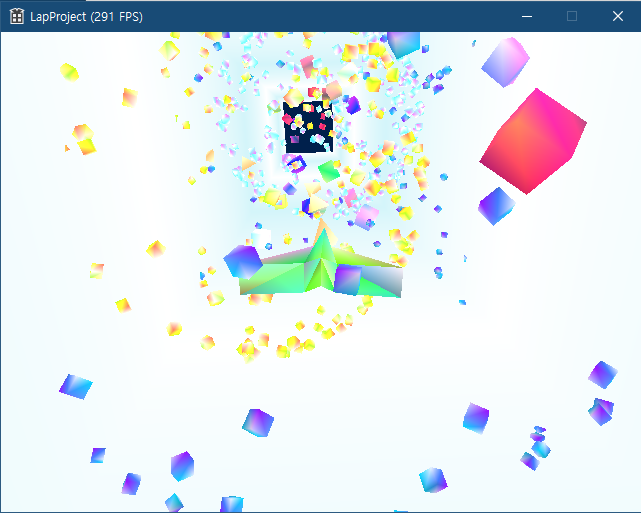


BuildObject를 위와 같이 수정하여 Shaders[0]과 [1]에 각각 다른 쉐이더를 저장하였습니다.



Shader의 Animate하여 나온 Boolean 값을 외부로 전달합니다.

Reset을 위해서 pd3dDevice와 pd3dCommandList가 필요한데 이 변수들은 GameFramework에 저장되어 있기에 이렇게 처리하였습니다.



프로그램 실행 화면