프로젝트 현황 보고서

# 프로젝트 요약

|  |  |  |
| --- | --- | --- |
| 보고 날짜 | 프로젝트 이름 | 작성자 |
| 2017년 2월 13일 | 꿀잼팀 졸작 프로젝트 | 강지훈 |

# 상황 요약

1. Animation Data
2. Animater
3. Mesh Texture
4. 파일 쓰기
5. File Based Mesh
6. Animation Tool 정리
7. Framework 개선
8. ResourceManager 개선
9. GBuffer Depth Stencil
10. SSAO
11. SSLR
12. 느낀점
13. 계획

# 내용

일단 1달동안 보고서를 장성하지 못 한점 죄송합니다..

애니메이션 툴 제작과정에서 UI를 제작하는 과정 중 UI에 필요한 정보들을 가져오기 위한 구조를 짜는 것이 어려웠다고 기억하는데 어디가 어떻게 어려웠는지 기억이 잘 나지 않아 그 부분은 생략하고, 이렇게 만들어 졌습니다! 결과 보고와 왜 이런 방법을 선택했는지 중점으로 보고서를 작성하겠습니다.

1. **Animation Data**
2. Animation Data가 Mesh안에 있었던 이유

개발 중에 Animation데이터가 Mesh안 쪽에 위치하고 있었다.

FBX의 Mesh마다 다른 Animation Data를 하나로 통합 하면 장비를 교체하여 다른 장비를 사용하면 다른 Animation Data를 이용하여 하나의 Animation Data로 다시 통합하는 과정을 거쳐야 했고 또 이 과정의 비용이 상당했기에 처음부터 나눠 놓았는데

이 때 관리하는 주체를 Mesh로 하는 것이 합당해 보여 Mesh에 넣어 주었다.

1. Animation Data와 Mesh 분리의 필요성

기존의 코드는 문제가 있다. Animation Data들만 따로 관리하기가 힘들다는 것이다.

Animation Data의 joint정보를 가지고 OBB정보를 컨트롤 해야 하는데 이 때 마다 mesh에 접근해야 한다. 이는 Mesh에 불필요한 기능이 추가되어 코드의 가독성을 떨어뜨리는 결과를 불러 일으킨다.

따라서 Animation Data와 Mesh Data는 분리되는 것이 맞다.

1. Animation Joint Data

Joint Data가 기존에 Animation을 실험하던 mesh에서는 이름이 같은데 정보가 다른 이상한 joint들이 있어서 모든 mesh가 하나의 Joint Data를 사용하지 못하였고,

이로 인해 Mesh마다 Animation Data들이 나눠지게 되었고 그래서 Mesh가 Animation Data를 들고있었지만

지금 우리가 만든 FBX 파일은 이를 하나의 Animation Data로 제작하여도 전혀 문제가 없었다. 그래서 하나의 Animation Data를 모든 Mesh가 돌려 사용하는 모양을 만들었다. 훨씬 보기 간편한 직관적인 구조가 되었다고 자부한다.

1. 한계

기존의 Mesh마다 Animation Data가 존재하는 경우에는 장비 교체가 쉽게 가능하였다. Mesh가 Animation Data를 들고있기에 Mesh만 교체하면 자동으로 Animation Data가 교체된다.

하지만 지금 나의 코드는 Animation Data를 모든 Mesh가 공유하는 형식을 띄고 있기에 관리는 편하지만 교체에 상당한 어려움이 따른다

만약 교체를 하게 된다면 Animation1에는 있는데 Animation2에는 없는 경우와 그 반대의 경우의 Joint들 때문에 Animation이 잘 동작하지 않는다.

이를 위해서는 Animation Tool에서 통합 Animation Data를 만들어야 했지만 아직 이것 까지 하지 못하였다.

1. **Animater**
2. 필요성

Animation Data를 하나만 사용할 것이 아니기 때문에 이를 관리할 객체가 필요했다.

1. 명세서

Animater는 객체마다 하나씩 가지고 있는다.

Joint들의 Data인 Skeleton Data를 관리한다.

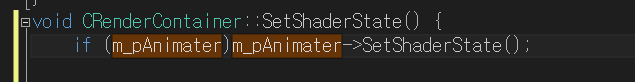
Main AABB를 가지고 있다.

1. 결과

위에서 Animation Data를 Mesh와 분리한 덕분에 Animater를 ResourceManager에서 제작하여 제작한 다음에 Animation정보가 있는 RenderContainer마다 하나씩 set할 수 있었다.

이렇게 함으로 써 관리가 쉬워 졌다고 하는 것이다.

기존에는 이렇게 RenderContainer에서 관리를 하지 못하였다. Mesh를 그릴 때 마다 다른 AnimationData를 set해야 했기에 mesh의 Render과정에서 set data/ Clear data과정이 있었다.



1. 한계

하지만 이 구조는 장비교체 즉 Animation Data의 수정 같은 유연성이 결여된 코드라고 하겠다.

장비 교체를 구현하기 위해 두가지 중 하나를 선택해야 한다.

하나 . Mesh가 변경되면 Animater에서 관리하는 AnimationData까지 교체해주는 번거로운 방법과

둘. 처음 Animation Data를 통합할 때에 하나의 Joint Tree를 제작하여 여러 Mesh에서도 공유가 가능한 Animation 정보를 제작하거나..

첫 번째 방법은 구현하기는 간단하지만 나중에 관리하기가 힘들 것이다.

두번째 방법은 구현하기는 번거롭지만 나중에 관리하는 것이 쉬울 것이다. 따라서 나중에 장비교체가 반드시 필요할 시에 이 두 방법 중에 하나를 골라 구현해 보도록 해야한다.

1. **Mesh Texture**
2. 필요성

우리가 사용하는 FBX에는 Mesh의 Texture정보가 결여되어 있다. 그래서 Mesh에 어떻 Texture가 쓰이는지 정해줄 필요가 있지만 이 Mesh의 순서도 매번 다르기 때문에 자동화 할 수가 없다 따라서 이를 Tool에서 Mesh를 하나하나 보면서 정해주는 것이 필요하다.

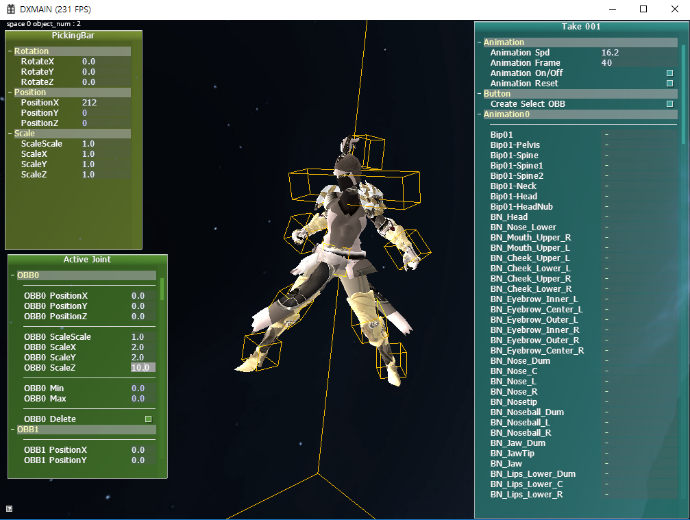
1. 명세서

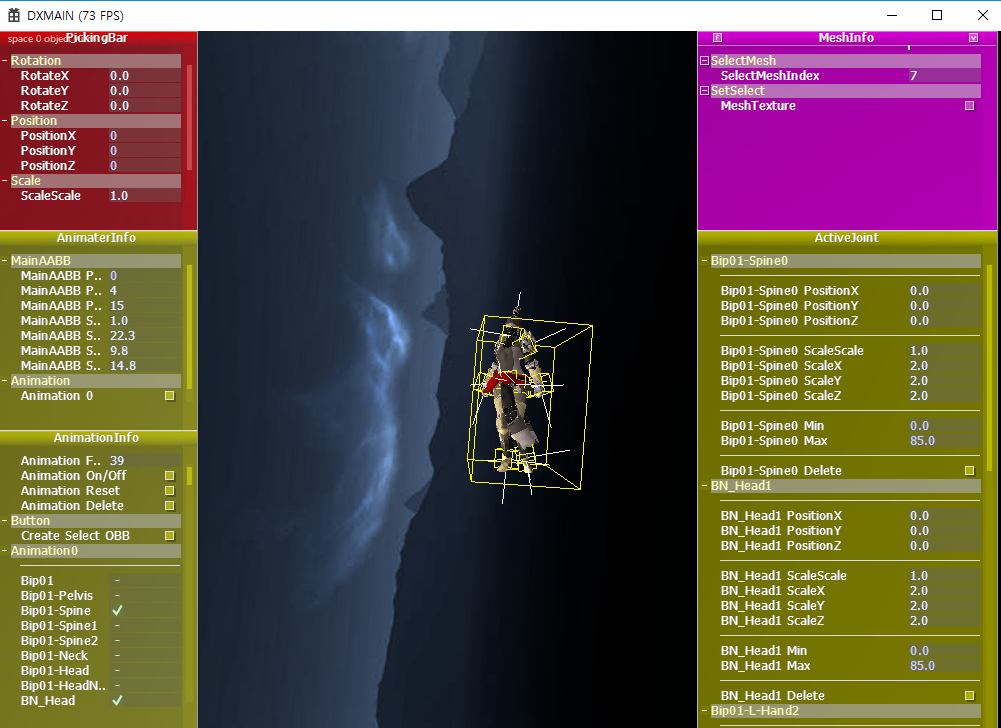
Tool에서 Mesh를 선택 하면 해당 Mesh가 빨간색이 되어야 한다.

선택한 Mesh에 원하는 Texture를 선택하면 해당 Texture가 표시되어야 한다.

마지막에 Mesh마다 정해진 Texture를 경로까지 저장가능해야 한다.

* 중간 결과





1. **파일 쓰기**
2. Mesh 출력

기존에 구조를 고민하던 것은 Animation이었지 Mesh가 아니었다 Mesh는 기존의 모습 그대로 출력하고 읽으면 될 것이다.

0. mesh name

<string name> //resource mgr에 저장할 때 사용 여기에 mesh index도 존재

0-1 mesh index

0-2. texture path

<string path> //파일 이름

1. vertex data

<nVertex> <pPosition> <pNormal> <pUV> <pWeight> <pIndex>

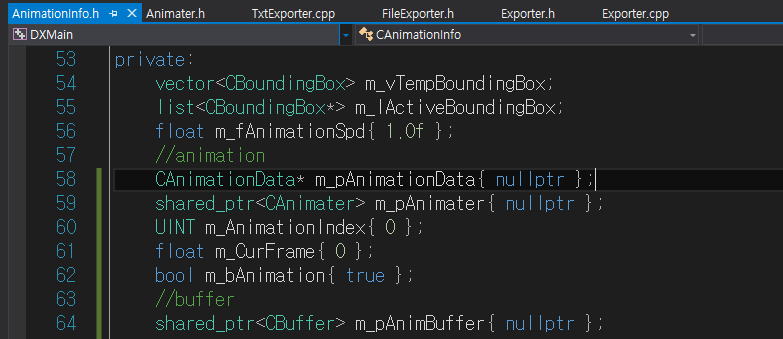
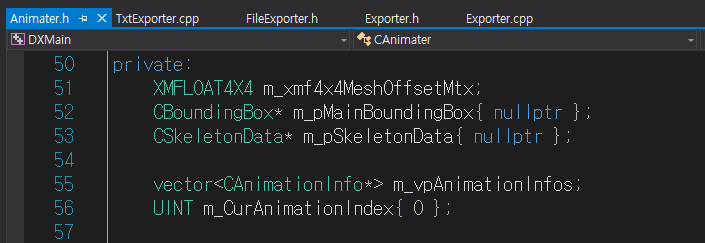
2. index data <nIndex> <pIndex>

1. Animation Data

문제는 Animation Data이다 FBX Animation Data를 저장하기 전에 저장을 쉽게 하기 위하여 확실하게 Data를 나누도록 한다.

JointData를가 원래 Joint의 OffsetMatrix정보와 해당 Joint의 Animation정보 즉 FrameMatrix까지 가지고 있었지만 Animater가 Joint의 OffsetMtx정보 즉 JointData를 가지고 있고 Animation Info객체가 FrameMatrix즉 Animation정보들을 가질 수 있도록 분리하였다.

기존의 FBX파일 전용 객체들인 FBX… 객체들의 FBX를 다 지운다. FBX에서만 사용하는 것이 아니라 모든 영역에서 사용 가능하도록 만들 것이다. FBXImporter의 Joint와 AnimationData의 의미를 변경한다. AnimStackInfo가 JointData를 가지고 있도록 하고 AnimationData가 frame matrix를 들고 있도록 변경하였다.



정리하자면 Animation Data가 CKeyFrame벡터를 가지고 Joint는 그대로 CKeyFrame정보만 뺀다.

이로써 출력할 때의 코드가 정돈된다. Animater는 Joint들의 정보 즉 Skeleton Data를 출력하고

Animation Info는 animation Data즉 Frame Matrix를 출력한다.

* Animater

<MainAABB> <Joint Count> <Joint Offset Matrix>

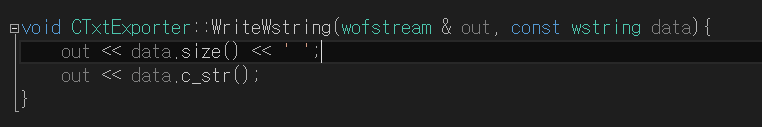
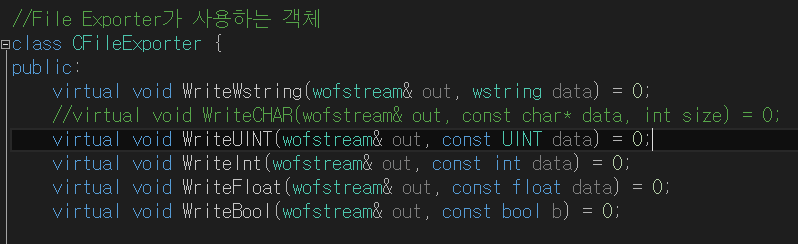
* AnimationInfos

<animation Count> 모든 animation에 대해 <JointOBB> <animation speed> <frameCount> <frameMatrix>

1. Exporter

file을 출력할 객체를 추상화 해서binary/ txt 두가지 방식을 지원 하도록 하자 지금은 눈에 보여야 하니까 txt로 진행하다가 나중에 binary로 변경하도록 한다.

Exporter가 정보를 출력하는 것을 관장하는데 이 때 출력하는 방법을 가상화 해서 FileExporter를제작하였다. WriteUINT, WriteFloat등 함수를 가상화 하여 쓰는 방법에 따라서 txt, binary로 나누려고 하였다. 현재는 txt만을 사용하고 있다. 나중에 느리다면 변경 하겠다.



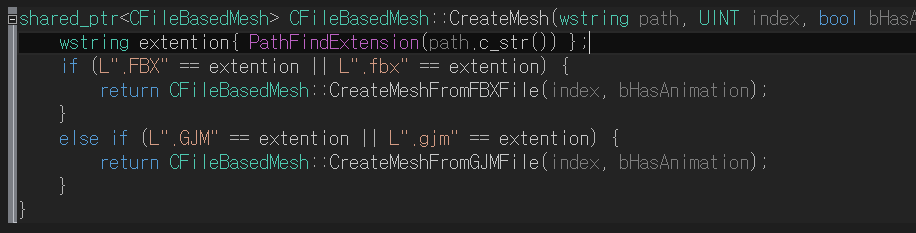
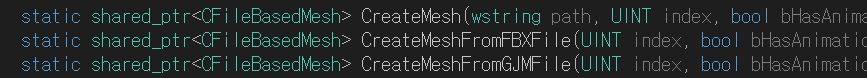
이로 인해 나중에 새로운 출력 방식을 도입 할 때 활로를 미리 연 것과 같다!

1. **File Based Mesh**
2. 방법

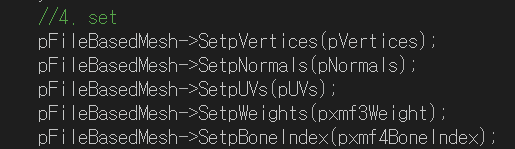
일단 File Based Mesh가 꼭 Animation Data가 존재하는 것이 아니기에 Animation Data가 있는 mesh와 없는 mesh를 나눠야 했지만 그러면 내가 힘들어 진다. 그래서 animtaion 정보와. mesh의 정보를 같이 두었다. 대부분 Animation Data를 가지고 있고 나중에 필요하면 바로 만들 수 있기에 나중에 필요할 때 따로 제작한다.

1. Load방법

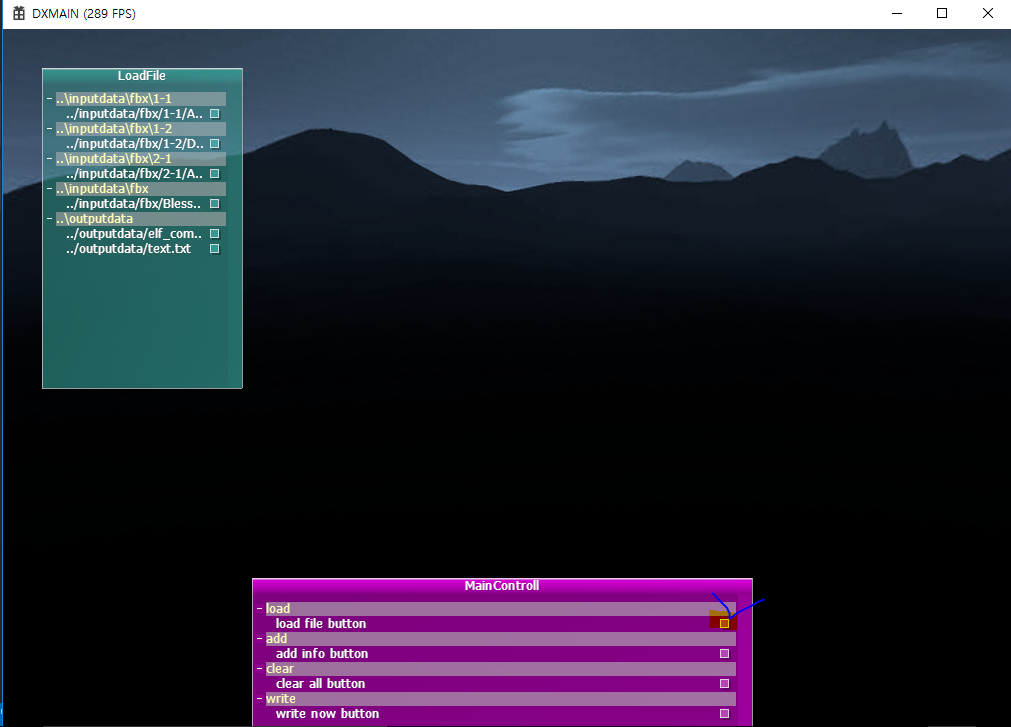
CreateMesh 함수를 static함수로 제작하여 인자의 확장자를 보고 FBX인지 GJM인지 판단한다음 해당하는 Create함수를 호출하도록 한다.



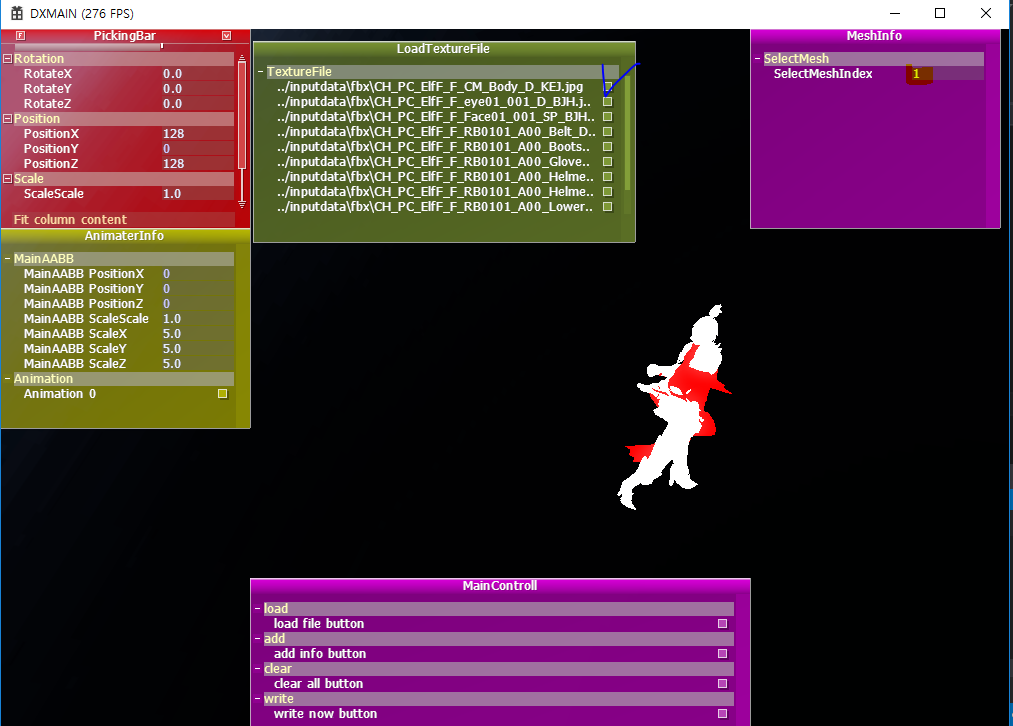
이 때 해당 파일에서 원하는 정보를 얻은 다음 set함수로 모든 정보를 set한 후에 Mesh의 Begin함수로 Vertex Buffer와 Index buffer를 제작한다.



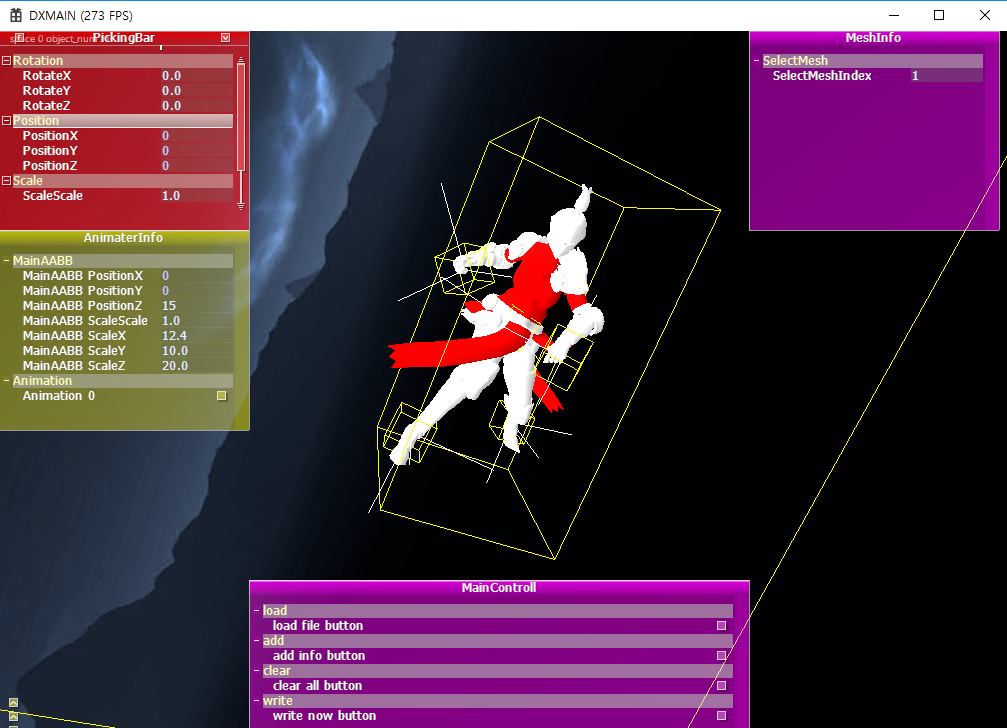
1. **Animation Tool 정리**
2. FBX 파일을 load 할 수 있다.



1. 각 Mesh는 자신의 texture를 가지며 이 Texture를 설정 해 줄 수 있다.



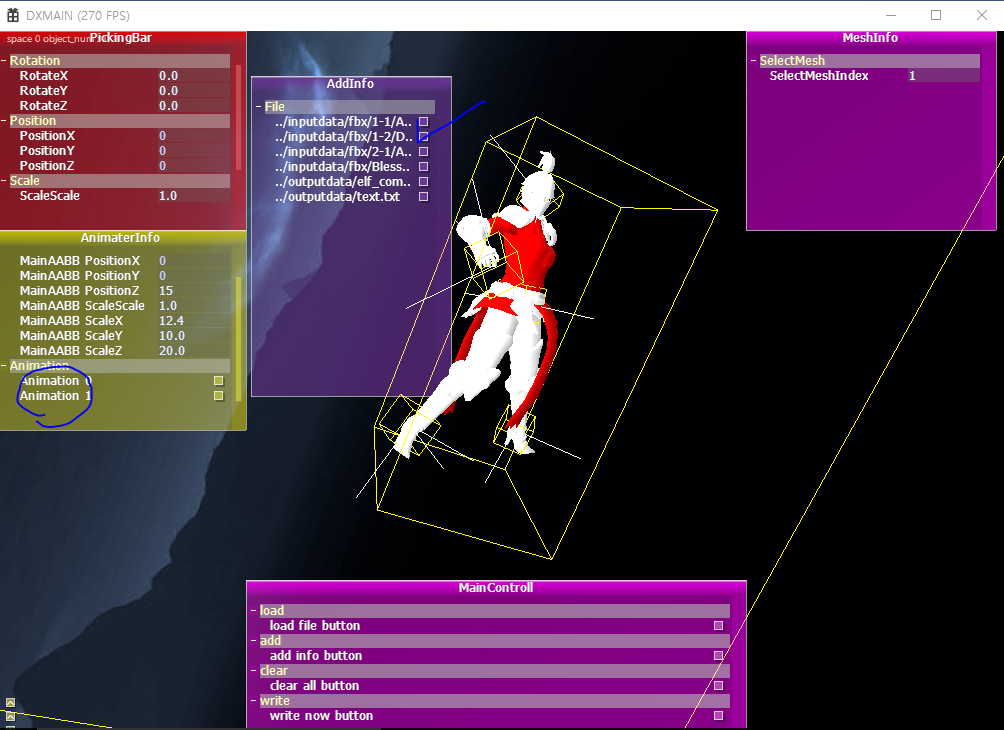
1. Animater는 main AABB를 가지며 크기나 위치를 편집 가능하다.



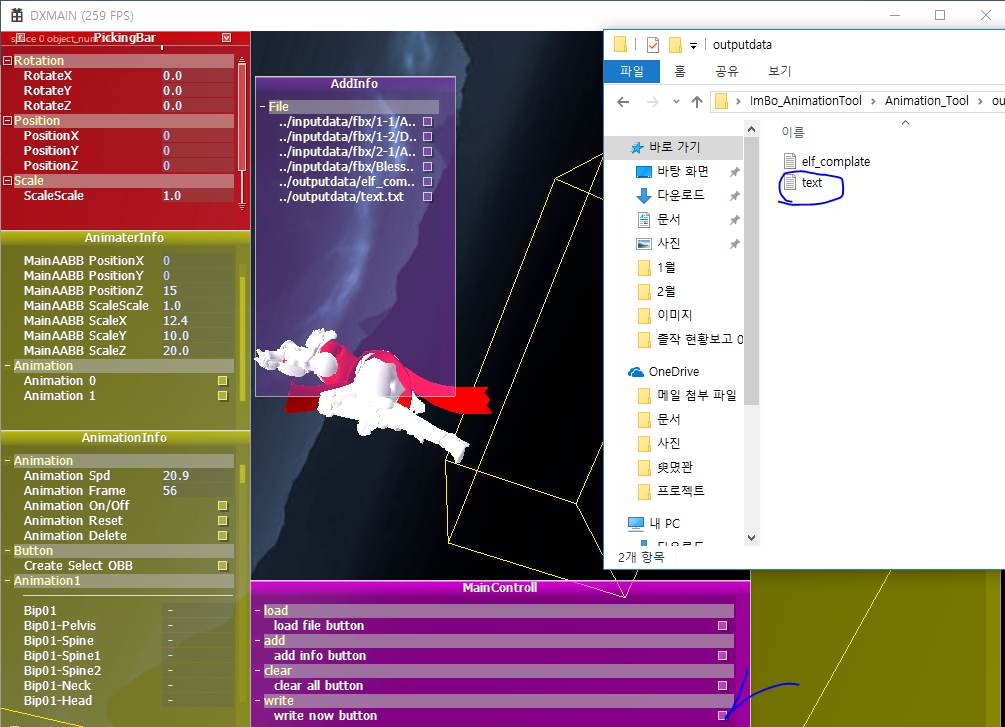
1. 각 animation info 는 joint OBB를 가지면 각 OBB를 편집 가능하다



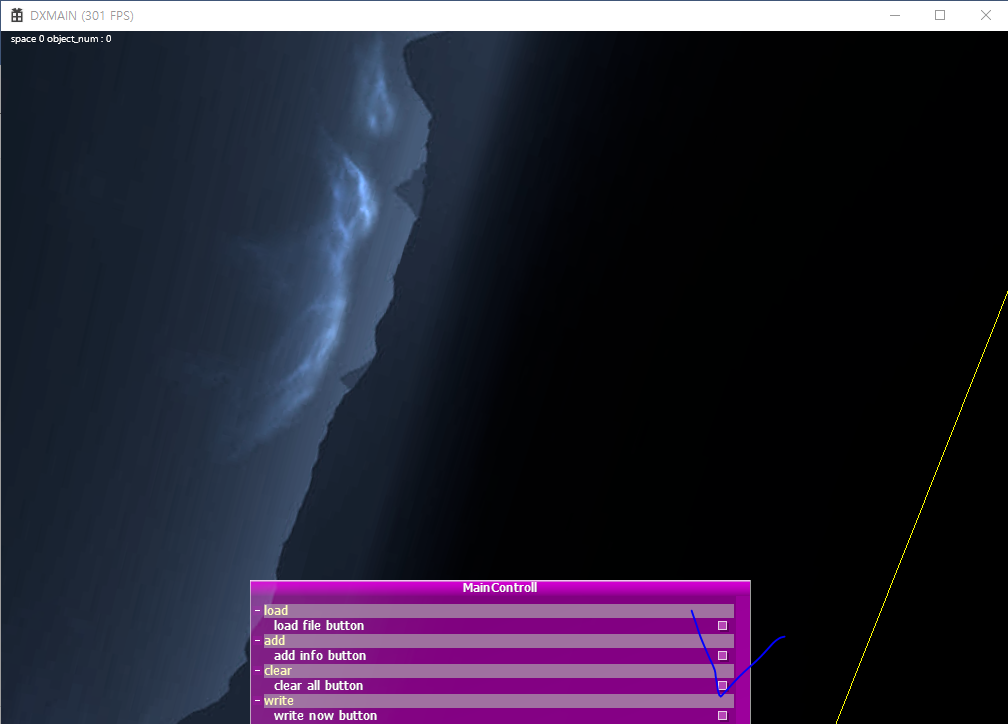
1. Animation data를 여러 개 합칠 수 있다.



1. 이 모든 데이터를 Write할 수 있다.



1. 데이터가 마음에 들지 않으면 delete/ clear 해 줄 수 있다.



1. 이 저장한 data를 다시 Load할 수 있다.



1. **Framework 개선**
2. 필요성

Framework의 기능이 너무 많다. 코드를 보면 엄청나게 길어졌는데 이를 나열해 보면

1. Device, DeviceContext제작
2. Scene관리
3. Render

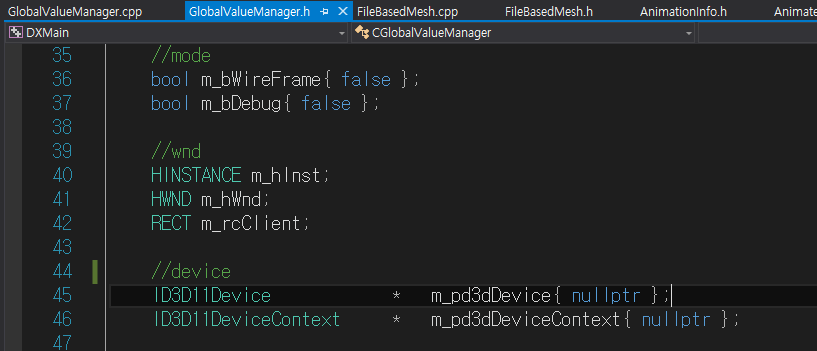
이렇게 3가지다. 이 각 기능 중에서 Scene관리 빼고 나머지를 나누는 작업을 진행하였다

또한 각 객체들의 초기화 방식의 일관성이 없는 부분과

ResourceManager가 너무 사용하기 불편하여 개선이 필요성을 절실히 느끼고 있어 Framework총 정리가 필요했다.

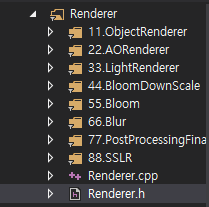
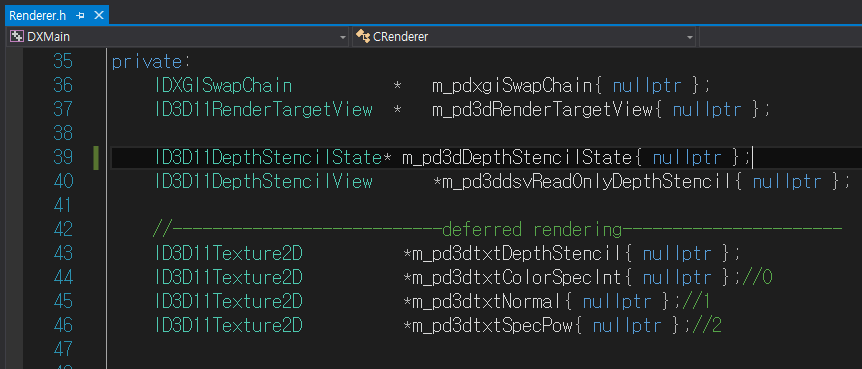
1. GlobalvalueManager

* Device와 DeviceContext를 이곳으로 옮겼다. 뿐만 아니라 DebugMode같은 코드 곳곳에서 사용 가능할 것 같은 전역 변수들을 전부 모아서 관리하는 싱글톤이다.



1. CRenderer제작

* Framework의 render기능을 전부 가지고 온다.



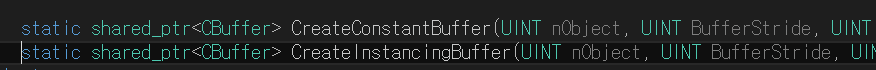
* Framework가 깔끔하게 정리 되었다.



1. **ResourceManager에서 각객체의 create를 만든다.**
2. Buffer

* 방법 : Buffer가 constantbuffer/ instancingbuffer/ globalbuffer등이 존재했지만 전부 constantbuffer혹은 instancingbuffer이면서 생성방법만 다른 것이었다. 따라서 Create함수를 constantbuffer와 instancingbuffer를 제작해 모든 buffer를 사용하는 곳에서 사용하도록 한다.

C:\Users\강지훈\AppData\Local\Microsoft\Windows\INetCacheContent.Word\de buffer.png



1. Render Shader

* 방법 : CreateRenderShader함수를 제작하여 함수 인자로 inputLayout Flag와 Bind Flag를 받아 제작한다.

C:\Users\강지훈\AppData\Local\Microsoft\Windows\INetCacheContent.Word\df rendershader input.png

* Input layout flag

POSITION NORMAL TEXCOORD TANGENT BITANGENT BONEWEIGHT BONEINDEX INSWORLDMTX INSPOS INSQUATERNION INSFLOAT\_C\_A INSFLOAT\_C\_B INSFLOAT\_C\_C INSFLOAT\_B\_A INSFLOAT\_B\_B INSFLOAT\_B\_C

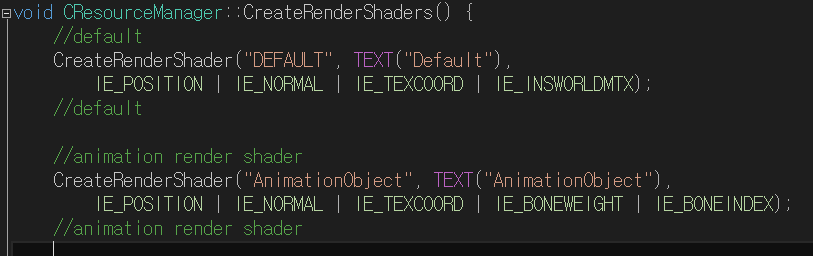
INSFLOAT\_ABC들은 FLOAT123들이다. 또 그 옆에 존재하는 \_ABC들은 첫번째 두번째 세번째 원소라는 표시이다. 정확한 이름보다는 이렇게 의미를 부여하는 것이 좀더 확장성이 있어 이런 변수명을 선택하였다.

* Shader 이름 정하기

VSname PSname같이 이름을 만들어 shader를 각각 제작한다.

* 결과

resourceManager에서 renderShader를 제작하는 곳이 깔끔해 졌다.

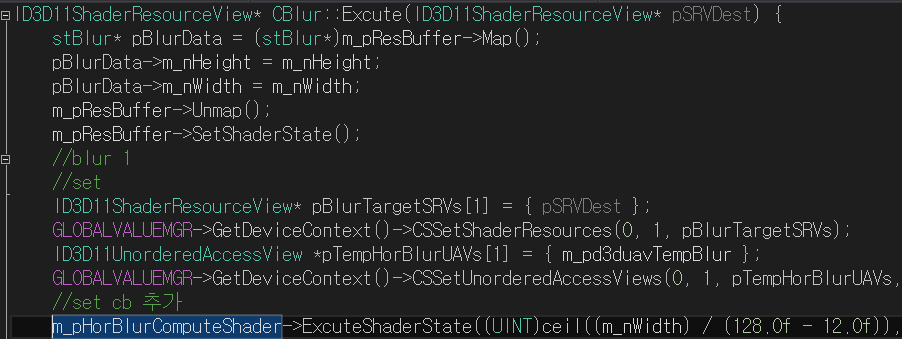
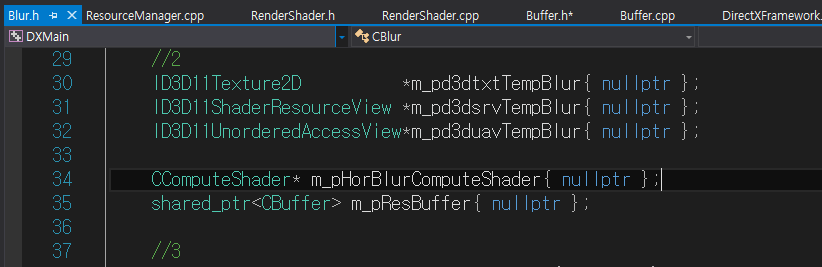


1. ComputeShader

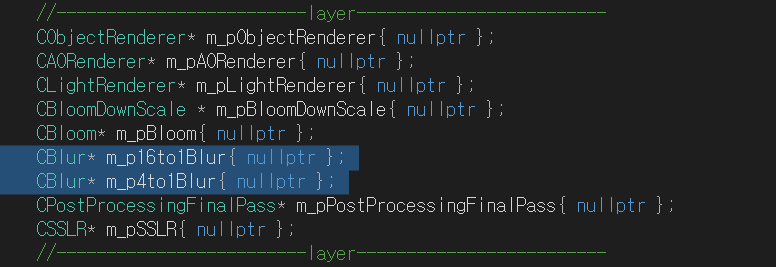
* 기존의 ComputeShader

기존의 ComputeShader는 자신이 맡은 역할을 다 끝내는 객체였다. 만약 blur를 한다면 사용할 data는 다 set되어 있다는 가정하에 computeShader를 돌리고 다 돌린 후에 결과물을 set하는 객체였다 이렇게 되면 computeshader객체의 재활용성이 떨어진다

* 방법 : ComputeShader는 computeShader만 동작 시키는 것을 기능으로 잡고 그에 사용하는 Resourcce Set Clean은 그 밖에서 관리하도록 했다.

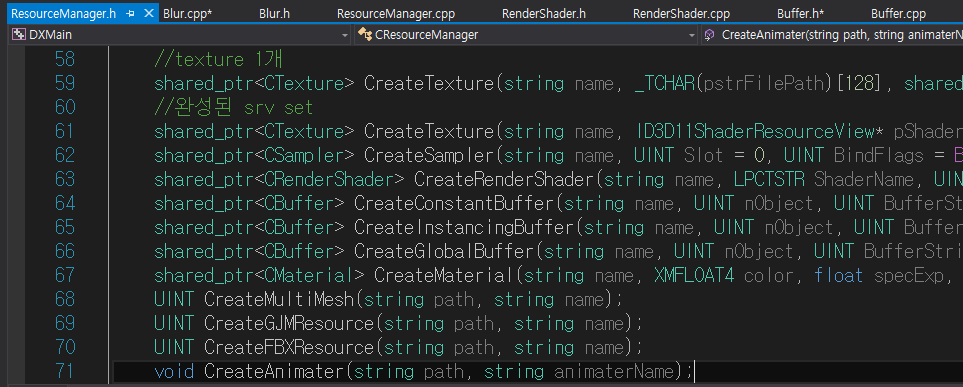


* 결과 : 이렇게 하니 Blur를 좀더 재활용 할 수 있었다.

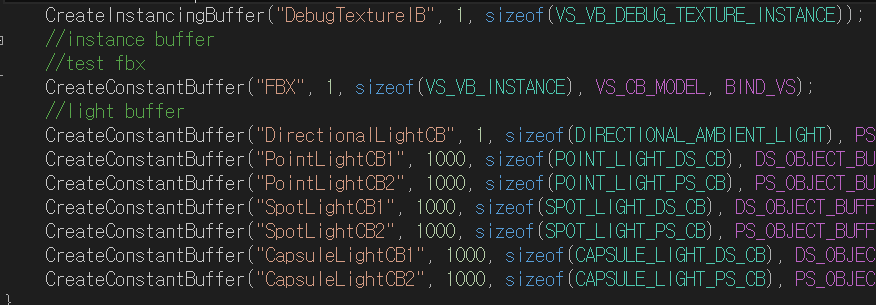


1. Etc Object

* 방법 : 기타 다른 객체들도 Create함수를 만들고 ResourceManager에서도 Create.. 함수를 제작해 객체 제작과 ResourceManager에서 관리하는 map에 추가하는 것까지 멤버 함수를 제작한다.



* 결과 : 코드를 더욱 깔끔하게 제작한다.



1. **Gbuffer에 depth stencil buffer를 사용하도록 한다.**
2. 필요성 :

* 기존에 사용하던 방식은 Gbuffer에 World Position을 render해서 Light 계산에 사용하는 것이었다. 이 방법의 문제는 2가지 이다.

1. Gbuffer가 하나 더 필요하다.
2. 깊이 값을 필요로 하는 후처리인 DOF 즉 거리심도를 하기가 힘들어 진다.

* 그래서 차선책이 GBuffer에 선형 깊이 값을 render 하는 것이다. 이렇게 함으로 써 깊이를 사용하는 후처리에도 사용이 가능해 졌다. 하지만 이 방법에도 문제가 있다.

1. 일단 gBuffer에 render 한다
2. 깊이 값을 얻기 위해 모든 정점에 대해서 view변환, porojection변환을 따로 해 줘야 한다.
3. 조명 계산에 필요한 world position을 구하기 위해 view inverse변환을 시켜줘야 한다.

* 마지막 방법은 render pipe line을 돌면 자동으로 나오는 결과물인 depth stencil buffer을 그대로 사용하는 방법이다. 이렇게 하면 render 할 gbuffer가 하나 줄어든다. 이 방법의 문제는

1. 조명 계산을 하기위한 world position 을 구하려면 projection 변환 view inverse 변환을 해야한다.

* 방법 1 :

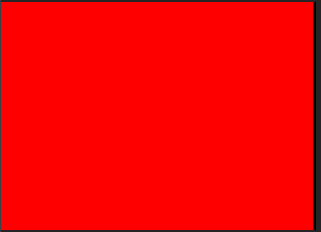
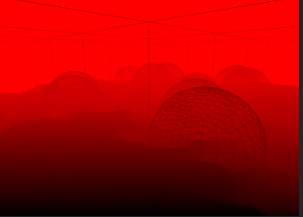
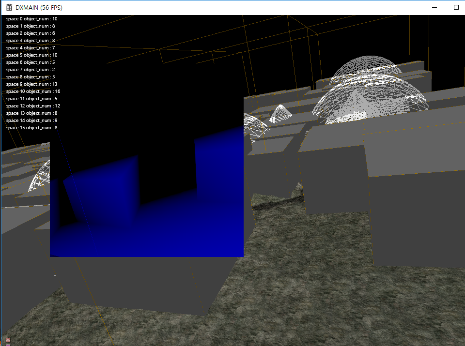
방법 1은 gBuffer에 Render하는 정보량도 많을 뿐더러 depth값이 필요하면 추가 계산까지 해줘야 한다. 그럼으로 방법 2, 3 중 하나를 선택해야 하는데

* 비교 :

처음에는 방법 2를 사용하려고 하였다. 이유는 render하는 정보가 중복으로 하나 더 생기긴 하지만 shader에서 계산하는 량이 적어서 였다. 하지만 이 방법은 scene에 object 정점 수가 많아지면 문제가 발생할 수 있다. 약 1000만개 즈음의 정점을 render할 경우

방법 2는 정점이 많아지면 많아질수록 계산양이 늘어나는 반면에 방법 3은 그렇지 않다 대신에 기본적으로 계산양이 조금 더 있을 뿐이다. 그 계산양은 projection 변환으로 \*두 번 + 두 번 정도로 거의 문제가 될 정도의 연산양이 아니라고 판단되었다.

* 결론 : 방법 3을 사용한다. 방법 2와 방법3은 50보 100보의 녀석들이지만 방법 3은 아무리 객체가 많아도 후처리, 조명계산을 할 때에 연산양이 방법2 보다 크게 증가하지 않는다는 것이 그 이유이다.

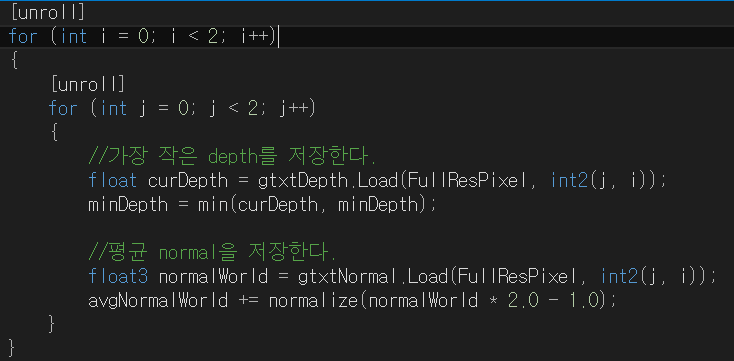
  

1. 문제 : 깊이버퍼를 이용하였더니 서로 깊이 먼 거리의 조명들이 서로 깊이 싸움을 하는지 번쩍거리는 것이 보였다. 가까움 거리의 조명들은 번쩍거리는 문제가 발생하지 않아 지금 비정상적으로 큰 Map이 아니라면 큰 문제는 되지 않을 것이라고 판단된다.
2. **SSAO**
3. 방법 :

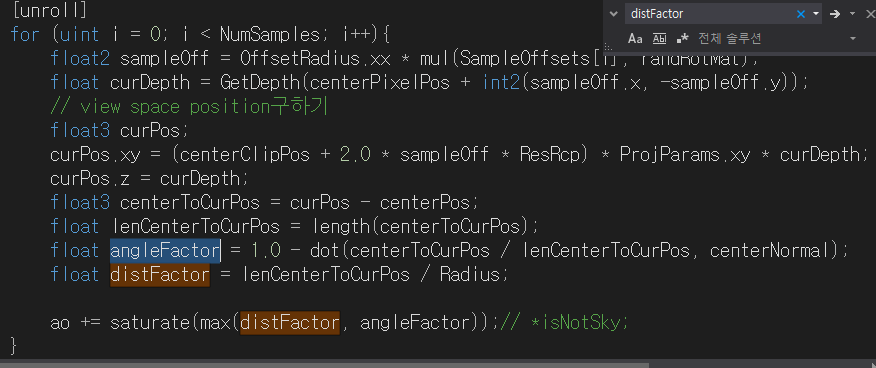
* 간단 하다. 일단 필요한 모든 것을 화면공간으로 변경한다 즉 투영 좌표계이다. 필요한 정보는 두가지이다. 하나. 깊이/ 둘. Normal



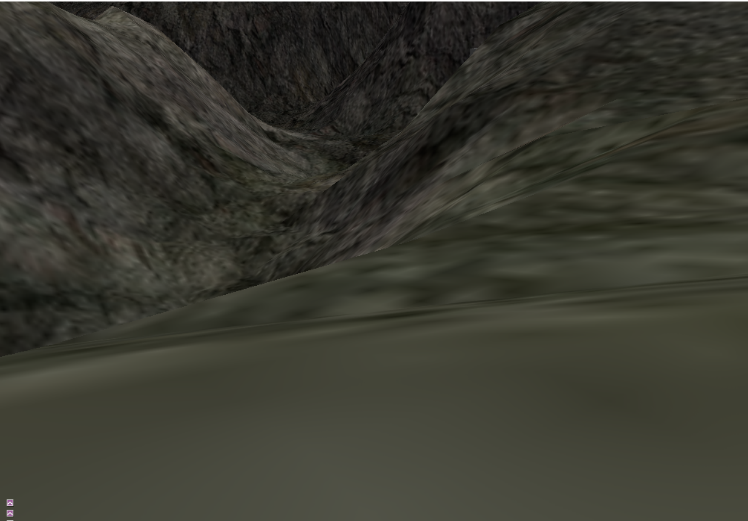
* 이 때 화면 전체를 계산하면 연산양이 많아져 퀄리티와 속도의 중간인 1/4 크기의 텍스쳐를 제작해 SSAO를 계산할 것이다. 해당 과정을 compute shader에서 진행한다.



* 이 정보를 가지고 해당 픽셀의 주위의 문작위 픽셀 8개를 뽑아 현재 좌표와 깊이 가 차이가 적게 나거나. 각도의 차이가 90도 이하면 차폐효과가 있다고 판단하고 기록한다. 이 과정을 cs에서 진행한다.



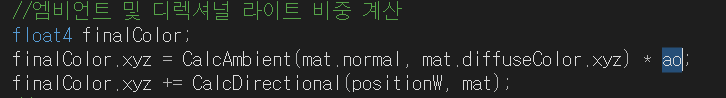
* 결과 :



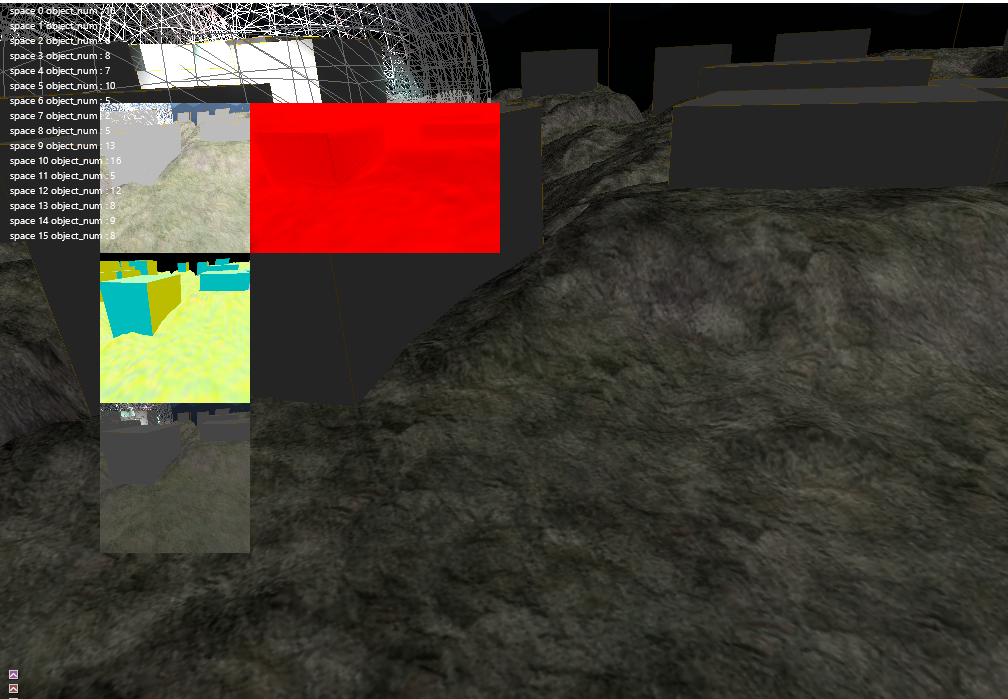


1. Ambient Light에 적용

* 만든 SSAO이미지를 ambient light를 계산한 후 마지막에 곱해준다. 그럼 차폐 효과가 적용된 것이다.



* 결과 :

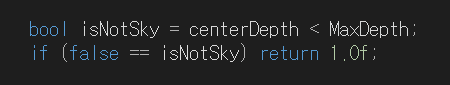


1. 문제

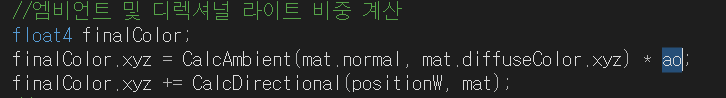
* 차폐가 많이 일어나는 곳의 영역이 넓다면, 넓은 영역의 차폐가 순식간에 일어났다가 사라질 때 화면이 깜박거린다. 이 문제를 해결하기 위해선 적응처럼 기존의 ao값과 현재 ao값을 교대해 가면서 선형적으로 밝아지거나 어두워 지도록 해야 한다..

1. **SSLR**
2. 차폐 이미지로 하늘과 불투명한 물체 구분

* 차폐 이미지에서 깊이 값을 가지고 구분을 한 녀석들 에게만 해당된다. 차폐 이미지를 제작하면서 깊이 값이 1이라면 즉 가장 먼 곳, 스카이 박스라면 그냥 1을 리턴 하도록 제작한다.

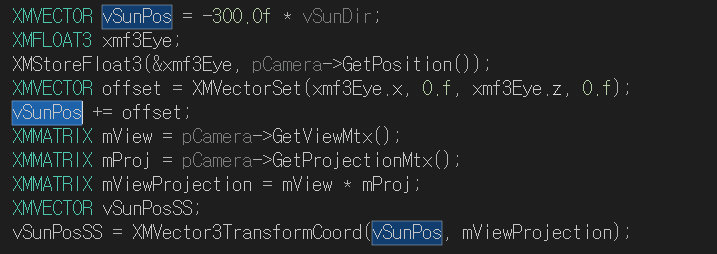


* 그 후에 SSLR에 사용한 Occlusion이미지를 제작할 때에 1이면 하늘이라고 생각하고 1을 return한다.

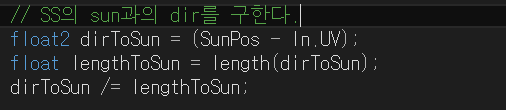


1. 광선 이미지 제작

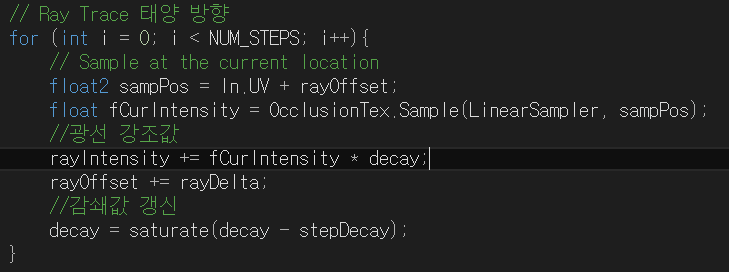
* 일단 태양의 위치가 필요하다 또한 태양의 방향도 필요하다. 이를 이용해 화면에 태양이 위치하는지 위치한다면 어느 곳에 위치하는지 알아낸다.



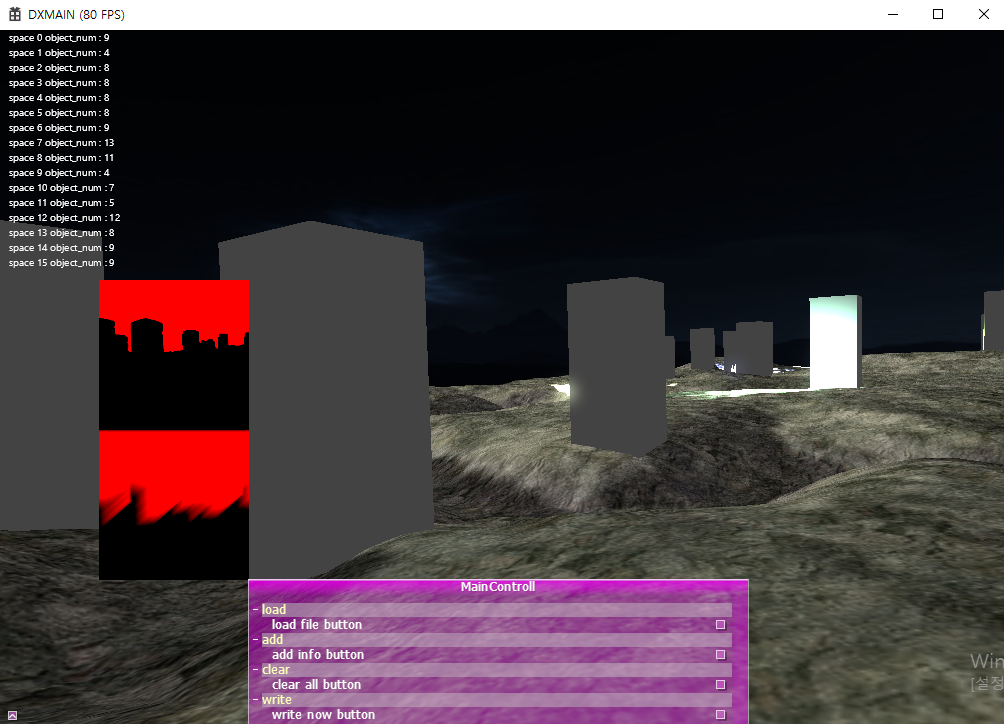
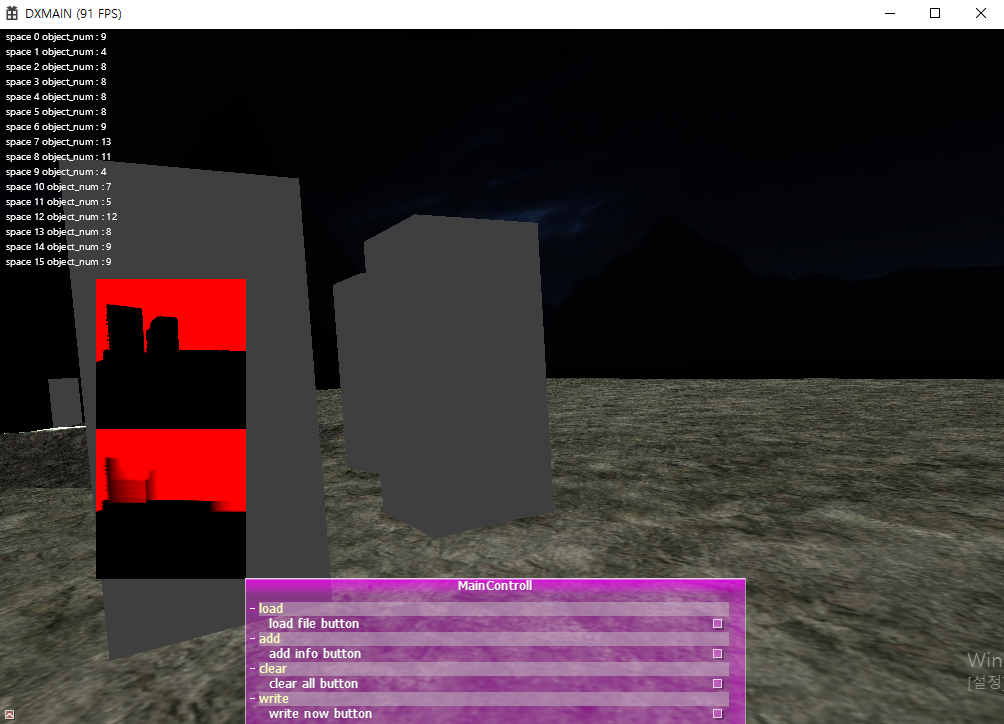
* 그럼 현재 픽셀의 화면상의 태양의 위치 쪽으로 이동하는 방향 벡터를 구할 수 있다.



* 이 방향 벡터를 n번으로 나눈 후 해당하는 픽셀들과 Occlusion이미지와 비교해 장애물이면 계산하지 않고 장애물이 아니면Ray Intensity값을 더한다. 이를 n으로 나누면 해당 픽셀의 RayIntensity값을 얻을 수 있다.

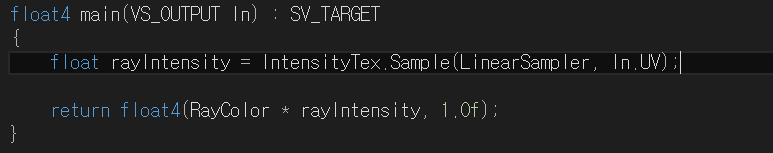


* 결과 :

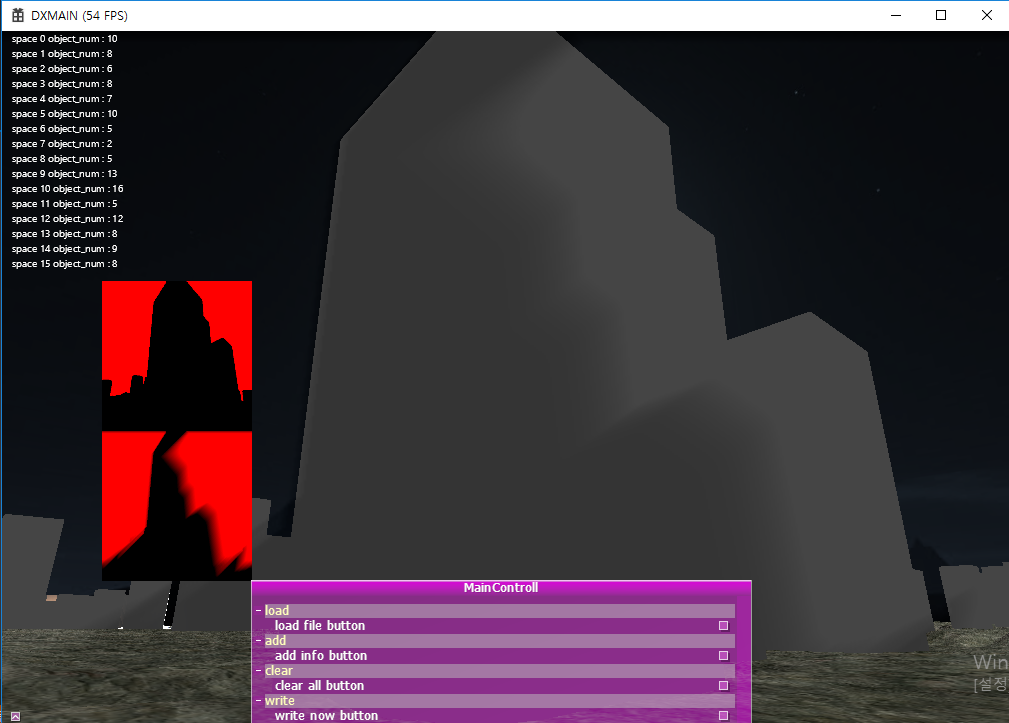


1. 광선 이미지와 Light RTV와 합성

* 위에서 얻은 RayIntensity값을 Light를 Render한 RTV에 가산 Blending하여 그리면 Light Shaft가 완성된다.



* 결과 :



1. **느낀점**

HLSL기능을 추가하는데 원리를 이해한지 어언 1달이 넘어가서 제작하는데 그리 오랜 시간이 걸리지 않았지만.. Framework가 새로운 기능이 추가되면서 기존의 틀을 부수고 다시 짜야 하는 경우가 파다하여 힘들었다. 특히 Animation이 그러했는데 아직도 개선해야 하는 점이 보여 큰일이다.

Animation정보를 제작하는데 있어서 가장 힘들었던 점은 구조의 제작이다. 어떻게 하면 좀더 편하게 개발할 수 있을 까 하는 쪽의 생각을 많이 하다 보니 구조도 계속 변경되어 머리가 너무 아팠지만 만족스러운 결과물이 나와서 너무 기쁘다. 아직 손봐야 할 곳이 넘쳐나지만 더욱 노력해서 좋은 flamework를 제작하고 싶다.

SSLR은 아직 활용하기에는 예쁜 장면을 어떻게 표현해야 할지 변수를 조정해 봐야 한다. 나중에 SSLR컨트롤 ui를 제작 했을 때 연구가 필요하다.

1. **계획**
2. Terrain의 height값을 .raw파일 말고 constant buffer로 만들어 pipe line에서 사용한다.
3. 나의 mouse point가 terrain의 어느 곳에 위치하는지 알아내어 해당 위치에 원을 표시한다.
4. 해당 원영역의 constant buffer값을 증가 시키거나 감소시킨다.
5. 증가시키는 방법을 cos그래프로 설정한다.
6. 후처리 bloom, hdr등 을 컨트롤 할 수 있는 ui를 제작한다.
7. 화면공간 효과를 컨트롤 할 수 있는 ui를 제작한다.
8. 조명의 범위를 설정 할 수 있는 ui와 framework 구조를 구현한다.
9. Scene의 정보를 완성하고 write 및 load할 수 있도록 한다.