|  |
| --- |
| 3DGP |
| 과제 2 |
| Rollercoaster Application |

|  |
| --- |
| 후안  2016180046 |

# 구현한 기능

|  |  |  |
| --- | --- | --- |
| **Category** | **Feature** | **Details** |
| Direct 3D | 업그레이드 이동 | 모든 기본 기능 다 Direct 3D으로 이동해서 업그레이드 했습니다. ( 초기화, Mesh Loading, Rollercoaster로직, 등) |
| 성능을 위한 기능등 | Direct 3D Instancing , Frustum Culling, Index Buffer 기능들 다 적용해서 Rollercoaster프로그램에 쓰고 있습니다. |
| Camera | 여러 카메라 모드 | F1) 1인칭 카메라. Rollercoaster 안 그리고, Rail만 맨 앞에 자리에 앉을 정도로 가까이에서 보일 수 있습니다. Rollercoaster 렌더링을 안 합니다.  F2) Orbital 카메라. 마우스를 Capture해서 이동하면서 카메라는 옵젝트 주위를 회전합니다.  F3) 3인칭 카메라. 맨 앞에 있는 Wagon좌석에 앉아 있는 사람처럼 월세상을 보여주는 카메라.  F4) 고정 카메라. 마지막 이동된 카메라의 위치를 고정되고Player 계속 바라보는 카메라. |
| Time | Slow Time | TimeDilation 변수를 통해서 fElapsedTime 바로 쓰지 않고 ElapsedTime 쓰기 전에 TimeDilation와 곱합니다. 이런식으로 TimeDilation 변수만 바꾸면 게임 진행 속도가 달라집니다. 예를 들면 TimeDilation 0.1로 되면 게임속도가 10배로 더 늘여집니다. |
| Mesh | Mesh Loader | Wagon mesh는 Vertex하고 Polygon이 많아서 직접 찍는 게 힘들어서 OBJMesh이라는 클래스를 만들어서 이 클래스를 통해서 OBJ File읽어서 Vertex Buffer와 Index Buffer만들 수 있습니다. 예를 들면 OBJMesh통해서 Wagon mesh, Rail mesh하고 나무 Mesh를 GameplayScene에 그립니다. |
| Terrain | PNG/Image Loader | PNG를 로딩하기 위해서 CImage를 통해서 PNG 로드하고 Heightmap는 Grayscale (Red값 = Green값 = Blue값)이라 Red 색깔 값만 저장합니다. |
| Ocean/Land (Landscape) | 높이가0이면 바닥 (파란색) 그리고, 아니면 법선벡터로 계산한 색깔로 된다. 이런식으로 해서 Terrain이 섬 Terrain 느낌이 납니다. |
| Instancing  (1/2) | Tree (Landscape) | 섬 Terrain에 나무들을 추가했습니다. 나무들이 바다(Blue Terrain)에생성 안 하게 했고 나무의 Up vector는 Map Terrain(x, z) 의 계산한 법선벡터와 맞게 했습니다. 이런식으로 해서 산이나 기울기가 높은 곳에 나무를 실제와 같이 제데로 생성하게 했습니다. 나무들이 똑같은 Mesh들을 써서 Instancing 합니다. |
| Instancing  (2/2) | Rail | Rail도 계속 생성하고, World변환만 다르니까 Instancing을 하게 했습니다. |
| Cube | 시작하는 위치에다가 Instancing된 큐브를 생성하고 계속 회전을 합니다. (따라하기 실습) |
| Frustum Culling | OBJ Mesh | Frustum Culling 기능을 제데로 쓰기 위해서 BoundingBox 설정해야 합니다. OBJ Mesh를 Bounding Box 설정하기 위해서 최대 촤표들 (Min 하고 Max) 찾아서 Range(Extents)와 가운데 점을 계산합니다. |

# Controls

(Rollercoaster Scene의 Process Input 함수에서 처리)

|  |  |
| --- | --- |
| **Key** | **Action** |
| ‘W’ | 생성할Rail들이 Pitch 양수 회전을 해서 생성합니다. |
| ‘S’ | 생성할Rail들이 Pitch 음수 회전을 해서 생성합니다 |
| ‘D’ | 생성할 Rail들이 Yaw 양수 회전을 해서 생성합니다 |
| ‘A’ | 생성할 Rail들이 Yaw 음수 회전을 해서 생성합니다 |
| 'Q' | 생성할 Rail들이 Roll 양수 회전을 해서 생성합니다 |
| 'E' | 생성할 Rail들이 Roll 음수 회전을 해서 생성합니다 |
| Spacebar | Spacebar 계속 누르고 있으면 게임 시간이 10배로 더 느리게 됩니다. Spacebar release하면 게임 시간 다시 원래 속도로 됩니다. 이 기능은Camera회전, Camera Walking에 효과가 없습니다. |
| F1 | 1인칭 카메라. 이 카메라는 Player 회전에 따라서 회전을 합니다. |
| F2 | 3인칭 카메라. 이 카메라는 Player의 위치를 따라하지만 Player의 회전 정보 상관없고 Player위치만 바라보는 겁니다. |
| F3 | 3인칭 카메라. 이 카메라는 Player 회전에 따라서 회전을 합니다. |
| F4 | 위치를 고정되어 있는 카메라. 이 카메라로 계속 Wagon을 보이지만 위치를 안 바꿔서 Rollercoaster를 멀리 모습을 보일 수 있습니다. |

# Framework 구조

**Non Object Shaders (거의 안 씀)**

* Build (ShaderVars, etc)
* Release(ShaderVars, etc)
* Update(ShaderVars, etc)
* Render(SetGraphicsPipeline)

SetGraphics

PipelineState

**Entry Point**

int main() {}

000

초기화 & Execute Command Lists

SetGraphics

PipelineState

World Matrix하고 다른 정보를 GPU에게 보내기 위한 Upload Buffers

SetGraphicsRootSignature

Graphics Pipeline

**Game Objects**

* Build (itself, ShaderVars, etc)
* Release(itself, ShaderVars, etc)
* Update(itself, ShaderVars, etc)
* Render(itself, Non Object Shaders)

**Object Shaders (Instance Shaders포함)**

* Build (Objects, ShaderVars, etc)
* Release(Objects, ShaderVars, etc)
* Update(Objects, ShaderVars, etc)
* Render(Objects, SetGraphicsPipeline)

**Active Scene (Gameplay)**

* Init
* Destroy
* Update
* Render
* Process Input

**Game Framework**

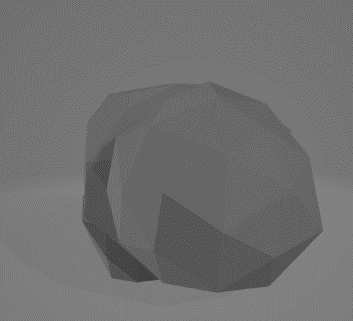
* Win32 초기화
* DirectX 초기화
* Game 루프
* Debugging 툴
* Scene 관리

# Framework 구조 설명

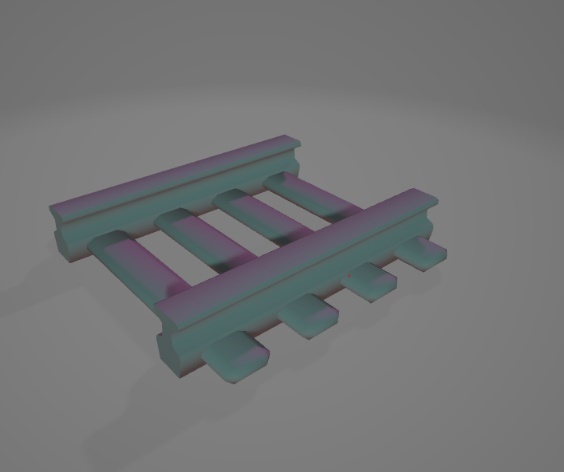
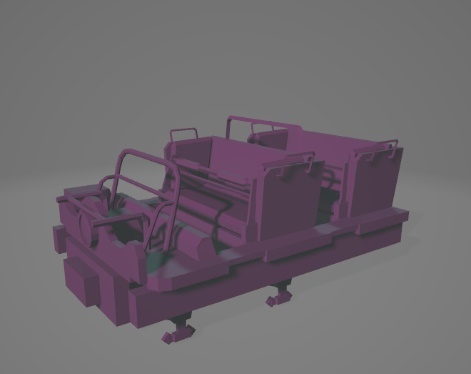
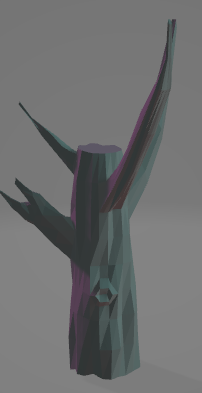
Scene에는 Shader만 있는 이유는 Shader 순서 없이 각 Object에 설정하면 SetGraphicsPipeline 많이 호출할 거라 성능 떨어질 수가 있어서 Shader으로 하면 SetGraphicsPipeline 몇번만 호출하기 때문입니다. 따라서, Instancing할 거와 Instancing 안 할 옵젝트들을 따로 Shader들에 분리했습니다. 예를 들면 Tree Instance는 Tree Shader, Rail Instance들이 Rail Shader, Cube Instance들 Cube Instance 있습니다. 이런 Shader들이 하나로 합칠 수가 있었지만 Rail이 특별한 변수와 함수들이 있어서 따로 분리했습니다.

# Mesh

이 과제가 쓰는 Mesh들은 총 3개 종류가 있습니다:

1) Rail Mesh 2) Wagon Mesh 3) Other Scene Meshes

Cube Mesh



1) Rail Mesh, Wagon Mesh들하고 나무 Mesh를 인터넷에서 무료 3D Mesh 사이트에서 다운로드했습니다. 이런 Mesh 들이 OBJ 파일 형식으로 다운로드해서 OBJMesh 클래스를 통해서 읽어와서 Vertex Buffer와 Index Buffer를 만들었습니다.

2) Cube Mesh는 직접 OBJ loader없이 Vertex와 Index buffer를 만들었습니다.

3) OBJMesh 클래스에서 다음 Constructor안에서 OBJ 파일들을 읽어올 수 있습니다

*OBJMesh.h (69줄) 하고 Mesh.cpp (258줄부터 310줄까지)*

OBJMesh(ID3D12Device \* pDevice, ID3D12GraphicsCommandList \* pCommandList,

const STD string & filepath

, const DX XMFLOAT4& DominantColor

, const DX XMFLOAT3& Scale = DX XMFLOAT3(1.f, 1.f, 1.f)

, const DX XMFLOAT3& Offset = DX XMFLOAT3(0.f, 0.f, 0.f));

Filepath: 읽을 파일 이름 및 경로

Dominant Color: RANDOM\_COLOR Macro와 같이 Interpolate할 색깔

Scale: Mesh의 각 Vertex를 Scaling함

Offset: Mesh의 각 Vertex를 Offset 추가함

이런 함수를 통해서 복잡한 Mesh를 만들 수 있습니다. Rollercoaster Wagon Mesh를 3DS Max에서 만들어서 OBJ 파일로 저장했습니다. Normal, Material 정보 필요가 없어서 삭제했습니다. X

const DX XMFLOAT3& Offset

const DX XMFLOAT3& Scale

를 통해서 Mesh정보를 로드하기 전에Scaling을 먼저 하고 Translate할 수 있습니다. Mesh의 Vertex Buffer하고Index Buffer필요하는 정보를 다음과 같이 찾습니다:

*OBJMesh.cpp (62~66줄)*

DX XMFLOAT3 Pos = DX XMFLOAT3(

(x \* Scale.x) + Offset.x,

(y \* Scale.y) + Offset.y,

(z \* Scale.z) + Offset.z);

pVertices.emplace\_back(Pos, InterpolateColor(RANDOM\_COLOR, DominantColor));

Index 정보를 저장할 때:

OBJMesh.cpp (81줄부터 110줄까지)

STD vector<UINT> pPolygon;

while (lineparser >> vertex\_index)

{

--vertex\_index;

pPolygon.emplace\_back(vertex\_index); //Index in OBJ file start with 1 not 0

*. . . (UV + 법선 벡터를 처리하는 부분) . . .*

for(UINT& i : pPolygon)

{

pIndices.emplace\_back(i);

}

모든 Vertex하고 Index정보를 다 모인다음에 Buffer View들을 만듭니다.

OBJMesh.cpp(127줄부터 134줄까지)

m\_VertexCount = static\_cast<UINT>(pVertices.size());

UINT Stride = sizeof(DiffusedVertex);

m\_VertexBuffer = CreateBufferResource(pDevice, pCommandList, pVertices.data(),

m\_VertexCount \* Stride, D3D12\_HEAP\_TYPE\_DEFAULT, D3D12\_RESOURCE\_STATE\_VERTEX\_AND\_CONSTANT\_BUFFER, &m\_VertexUploadBuffer);

m\_VertexBufferView.BufferLocation = m\_VertexBuffer->GetGPUVirtualAddress();

m\_VertexBufferView.SizeInBytes = m\_VertexCount \* Stride;

m\_VertexBufferView.StrideInBytes = Stride;

OBJMesh.cpp(127줄부터 134줄까지)

m\_IndexCount = static\_cast<UINT>(pIndices.size());

m\_IndexBuffer = CreateBufferResource(pDevice, pCommandList, pIndices.data(),

m\_IndexCount \* sizeof(UINT), D3D12\_HEAP\_TYPE\_DEFAULT, D3D12\_RESOURCE\_STATE\_INDEX\_BUFFER, &m\_IndexUploadBuffer);

m\_IndexBufferView.BufferLocation = m\_IndexBuffer->GetGPUVirtualAddress();

m\_IndexBufferView.SizeInBytes = m\_IndexCount \* sizeof(UINT);

m\_IndexBufferView.Format = DXGI\_FORMAT\_R32\_UINT;

따라서 SceneShader에서 WagonPlayer을 위한 Main Wagon Mesh을 다음과 같이 만들었습니다:

SceneShader.cpp (42 & 43줄)

Mesh\* MainWagon = new OBJMesh(pDevice, pCommandList, "Wagon1.obj", DX XMFLOAT4(0.75f, 0.f, 1.f, 1.f), XMFLOAT3(75.f, 75.f, -75.f));

# Camera

**------------------------------------------------------------------------------------------------------------------------------------------**

**Camera Update 함수**

**------------------------------------------------------------------------------------------------------------------------------------------**

카메라의 선택된 Option (Camera모드)에 따라서 카메라 업데이트하는 정보가 다릅니다.

Camera Mode:

F1) 1인칭 카메라. 카메라의 회전 정보 (Right, Up, Look)는 Player의 회전 정보로 설정합니다. Player의 회전 변환만 Matrix으로 바꿔서 이 Matrix를 통해서 Camera의 Offset을 변환하고 1인칭 위치 Offset 계산 할 수 있습니다. 회전할때, 이 카메라가 회전 하는 축 다음과 같다:

Pitch: 자기 로컬 X축 ( Camera Right Vector)

Yaw: 해당하는 Player의 로컬 Y축 (Player Up Vector)

Roll: 해당하는 Player의 로컬 Z축 (Player Look Vector)

카메라 위치도 Player를 계속 따라하기 위해서 설정합니다. Rollercoaster렌더링을 안 합니다.

F2) Orbit 카메라. 마우스를 이동하면서 카메라가 Player 주위를 회전합니다. 이런 것 구현하기 위해서 카메라 Offset을 회전 변환해서 Target (Player의 위치)와 더해서 결과가 Camera 위치로 됩니다.

F3) 3인칭 카메라. 1인칭과 같이 Update 함수를 비슷하지만 Player위치로 바라봐야 되기 때문에 XMMatrixLookToLH함수를 쓰지 않고 XMMatrixLookAtLH 함수를 씁니다.

F4) 위치가 고정되어 있는 카메라. 업데이트를 안 합니다. LookAt Matrix를 계산하기 위해서 다음 함수와 Parameter를 씁니다:

DX XMMatrixLookAtLH(Camera위치, Player위치, Camera의 Up)

# Input 로직 (GameplayScene.cpp)

**-----------------------------------------------------------------------------------------------------------------------------------------**

**Process Input**

**------------------------------------------------------------------------------------------------------------------------------------------**

입력을 처리하고 카메라 모드 바꿀 수 있고, 다음 생성할 레일의 회전을 다르게 할 수 있고, 게임 시간도 10배로 늘여질 수도 있습니다. 입력 처리 다음과 같습니다:

KEY\_PRESSED(pKeyBuffer, 'W')

Rotation.x = 1.f \* RotationScale;

PRESSED는 입력 처리 체크하는 Macro입니다: (stdafx.h에 있습니다)

#define KEY\_PRESSED(pKeyBuffer, VirtualKey) if(pKeyBuffer[VirtualKey] & 0xF0)

**------------------------------------------------------------------------------------------------------------------------------------------**

# Heightmap Image Loading (HeightMapImage.cpp)

**-----------------------------------------------------------------------------------------------------------------------------------------**

**CImage**

**------------------------------------------------------------------------------------------------------------------------------------------**

CImage를 통해서 여러 형태의 이미지 파일들을 읽어와서 Heightmap 만들어 줄 수 있습니다. 다음과 알고리즘은 CImage는 Pixel 배열 형태로 바꾸는 알고리즘입니다:

*HeightMapImage.cpp (10~42줄)*

CImage image;

image.Load(filename.c\_str());

int PixelStride = image.GetBPP() / 8;

m\_Width = image.GetWidth();

m\_Depth = image.GetHeight();

m\_HeightMapPixels.clear();

BITMAP bmp;

GetObject(image, sizeof(BITMAP), &bmp);

BITMAPINFOHEADER bmih{ 0 };

bmih.biSize = sizeof(BITMAPINFOHEADER);

bmih.biWidth = bmp.bmWidth;

bmih.biHeight = bmp.bmHeight;

bmih.biPlanes = 1;

bmih.biBitCount = image.GetBPP();

bmih.biCompression = (BI\_RGB);

HDC hdc = GetDC(NULL);

GetDIBits(hdc, image, 0, bmp.bmHeight, NULL, (LPBITMAPINFO)&bmih, DIB\_RGB\_COLORS);

m\_HeightMapPixels.resize(bmih.biSizeImage);

GetDIBits(hdc, image, 0, bmp.bmHeight, &(m\_HeightMapPixels[0]),

(LPBITMAPINFO)&bmih, DIB\_RGB\_COLORS);

ReleaseDC(NULL, hdc);

for (int y = 0; y < m\_Depth; y++)

for (int x = 0; x < m\_Width; x++)

{

int IndexDst = x + y \* m\_Width;

int IndexSrc = (x + (y\*m\_Width)) \* PixelStride;

m\_HeightMapPixels[IndexDst] = m\_HeightMapPixels[IndexSrc];

}

*(PixelStride이라는 변수는 Image Pixel 사이즈를 알려 주는 변수입니다. 예: R8G8B8 Format 경우에는 Pixel Stride는 3 입니다 – 3 byte)*

**-----------------------------------------------------------------------------------------------------------------------------------------**

**나무 생성 로직**

**------------------------------------------------------------------------------------------------------------------------------------------**

나무들을 Spawning하기 위해서 다음 줄들을 실행합니다:

*TreeObjectShader.cpp (20~70줄)*

// 나무 쓰는 Mesh들을 생성

Mesh\* pLeavesMesh = new OBJMesh(pDevice, pCommandList, "tree\_leaves.obj",

XMFLOAT4(0.f, 0.5f, 0.f, 1.f));

Mesh\* pTrunkMesh = new OBJMesh(pDevice, pCommandList, "tree\_trunk.obj",

XMFLOAT4(0.8f, 0.4f, 0.1f, 1.f));

//나무 사이 거리 (거리 커지면, 나무 갯수가 적어집니다)

float DistanceBetweenTrees = 50.f;

//맵에 존재하는 Terrain의 정보 읽어오기

float TerrainWidth = static\_cast<float>((\*m\_Terrain)->GetHeightMapWidth());

float TerrainDepth = static\_cast<float>((\*m\_Terrain)->GetHeightMapDepth());

UINT ObjectRow = (int)((TerrainWidth / DistanceBetweenTrees) + 1);

DX XMFLOAT3 Scale = (\*m\_Terrain)->GetScale();

DX XMFLOAT3 Offset = (\*m\_Terrain)->GetOffset();

//나무 최대 갯수 계산해서 Memory Allocation 하기

m\_ObjectCount = ObjectRow \* (UINT)((TerrainDepth / DistanceBetweenTrees) + 1);

m\_Objects.reserve(m\_ObjectCount);

//Terrain따라서 나무 Terrain의 Offset 더하기 전의 위치를 계산하기

auto GetNewTreePosition = [&](int r)->DX XMFLOAT3

{

float X = (r % (int)ObjectRow) \* DistanceBetweenTrees \* Scale.x;

float Z = (r / (int)ObjectRow) \* DistanceBetweenTrees \* Scale.z;

return DX XMFLOAT3(X , (\*m\_Terrain)->GetHeight(X, Z), Z);

};

GameObject \*pObject = NULL;

for (UINT i = 0; i < m\_ObjectCount; ++i)

{

pObject = new GameObject;

DX XMFLOAT3 Pos = GetNewTreePosition(i);

if (Equal(Pos.y, 0.f))

delete pObject;

else

{

//Terrain의 법선 벡터 따라서 Quaternion Rotation 다릅니다.

XMFLOAT4 Quat = GetLookRotationQuaternion(gWorldUp,

XMLoadFloat3(&(\*m\_Terrain)->GetNormal(Pos.x, Pos.z)));

pObject->Rotate(Quat);

//나무 렌덤 Yaw(로컬 Y축) 회전. 이런씩으로 Tree중복성 느낌이 안 납니다.

pObject->Rotate(0.f, 180.f\* ((float)rand() / (float)(RAND\_MAX)),

0.f);

//Position + Offset = 월드 좌표

pObject->SetPosition(DX XMFLOAT3(Pos.x + Offset.x, Pos.y + Offset.y,

Pos.z + Offset.z));

m\_Objects.emplace\_back(pObject);

}

}

m\_Objects[0]->AddMesh(pTrunkMesh);

m\_Objects[0]->AddMesh(pLeavesMesh);

GetLookRotationQuaternion이라는 함수는 다음과 같이 선언합니다:

inline DX XMFLOAT4 XM\_CALLCONV GetLookRotationQuaternion(DX XMVECTOR\_P0 Source, DX XMVECTOR\_P1 Target);

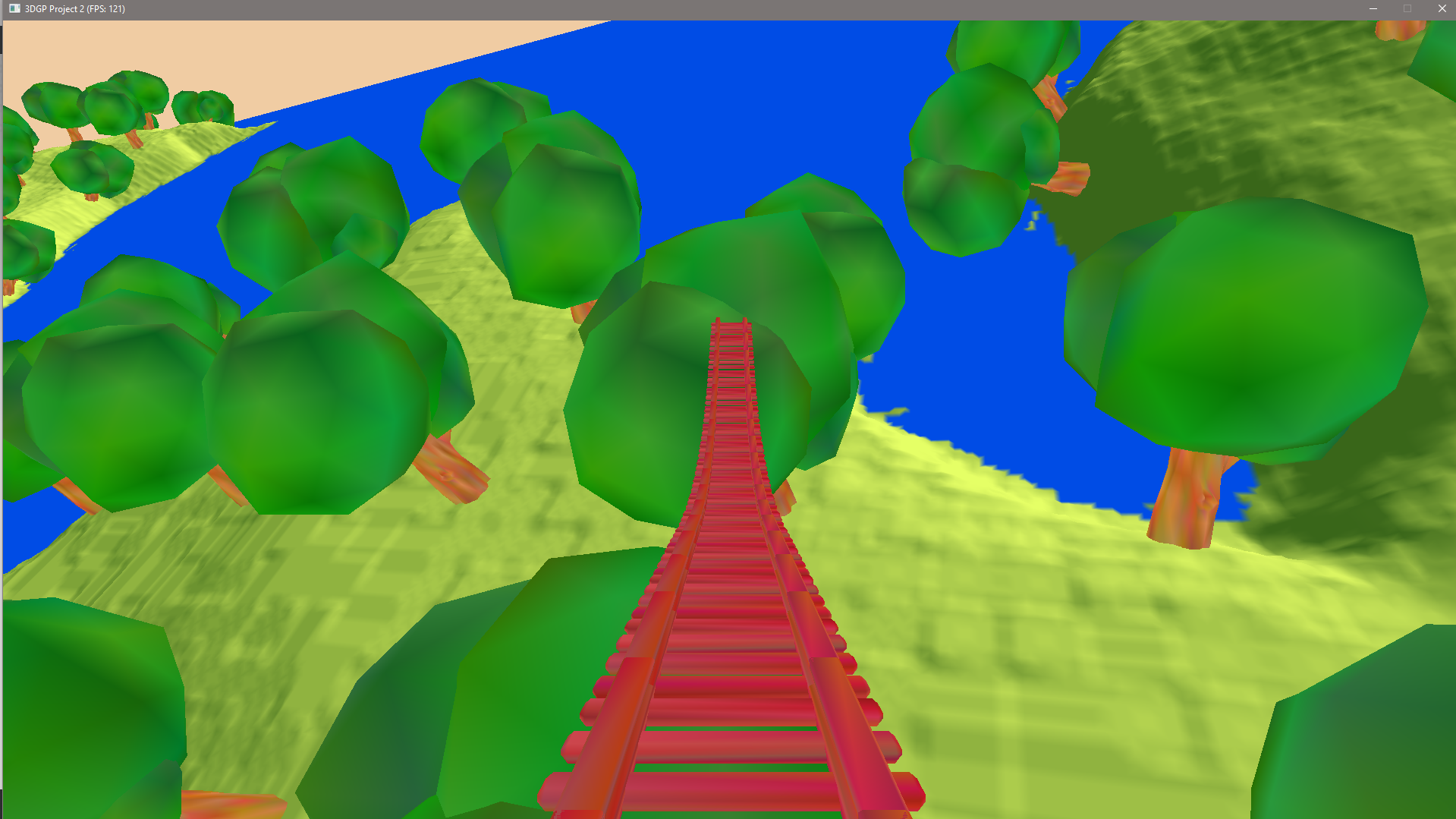
이 함수를 통해서 ‘Source’라는 벡터가 ‘Target’라는 Normalized벡터로 방향으로 바라보기 위해서 Rotation 정보를 계산해서 Quaternion으로 리턴하는 함수입니다. 이 함수를 통해서 나무의 로칼 Y축은 Terrain의 법선 벡터와 맞게 할 수 있습니다.

# 소감

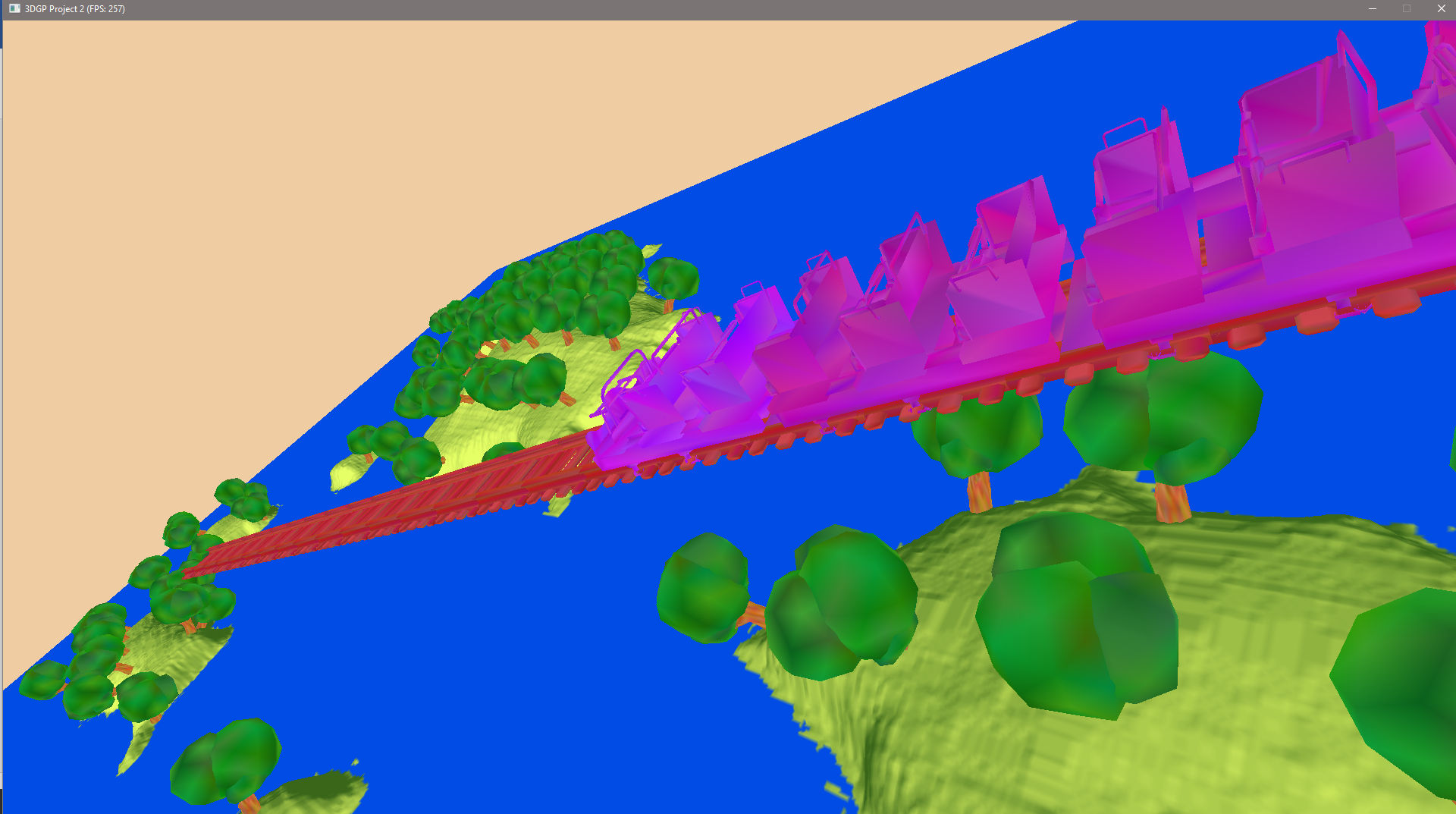
Win32를 Direct 3D로 업그레이드하면서 힘든게 많았습니다. 첫째, Direct 3D 초기화 했을때 생성해야 되는 Interface가 많아서 세팅 하나만 틀리면 결과가 많이 달라질수 있습니다. 또한, 이 프레임워크안에서 IUnknown Interface들이 썼을 때 AddRef하고 Release쓰는 것보다 ComPtr 써 보고 싶어서 썼는데 처음에 문제가 많이 생겼습니다. AddRef, Release자동으로 해서 Buffer 리소스를 생성했을때 그 리소스를 해당하는 ComPtr 자동으로 Release했습니다. 그래서 ComPtr Detach와 다른 ComPtr에 대한 함수 알게 됐습니다. 앞으로 이런 문제 생기면 어떻게 해결할지 알고 있습니다.

초기화 한 다음에, 따라하기 제데로 하면서 문제가 거의 없었습니다.

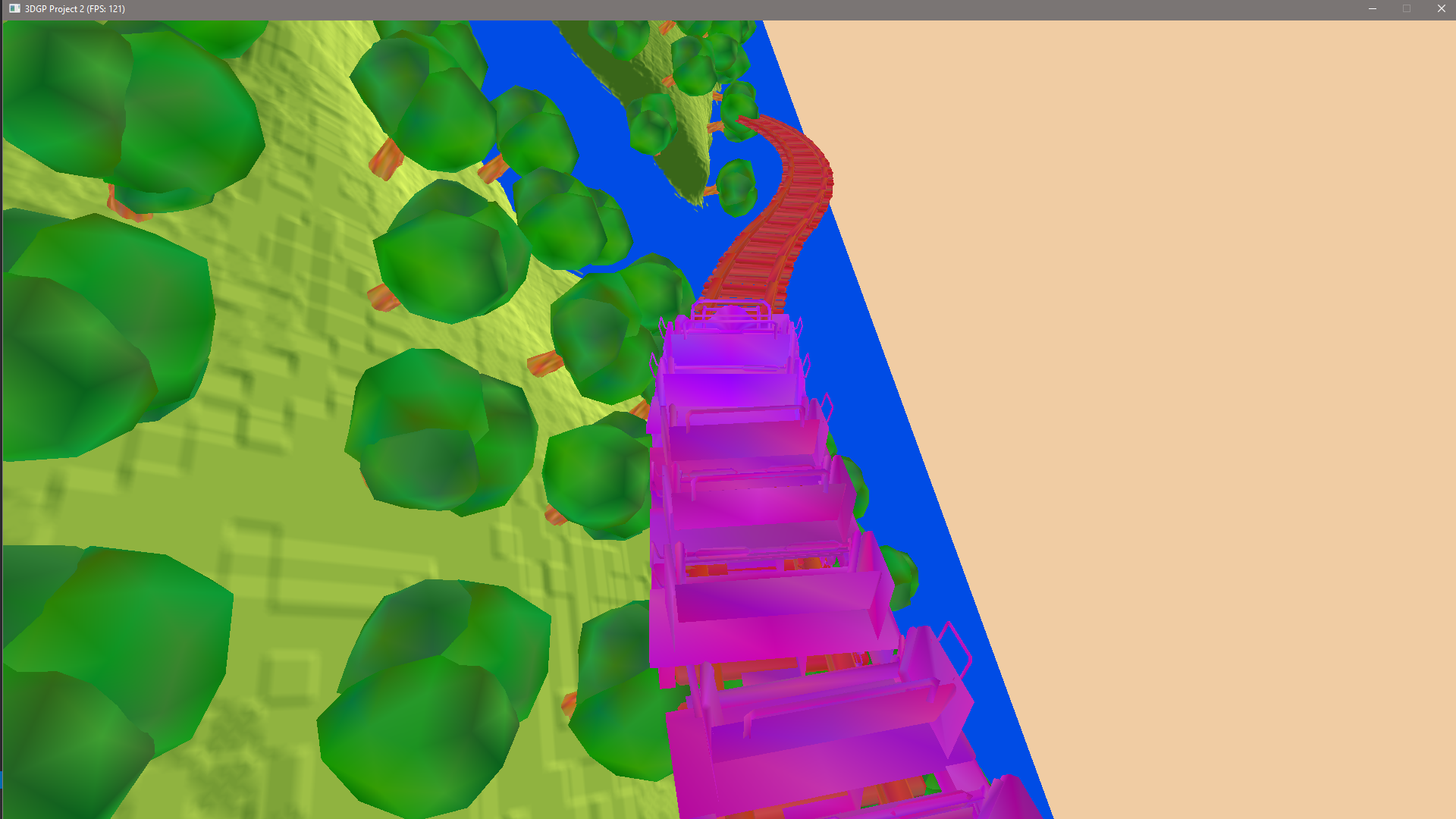
# Gameplay Screenshots

Camera Mode F1 (1인칭 카메라, 플레이어 회전 의존성이 있습니다)

Camera Mode F2 (Orbital Camera)



Camera Mode 3 (3인칭 카메라)



Camera Mode 4 (위치 고정된 카메라)

