**3D Game Programming 과제3 보고서**

게임공학과 2013180028 이우상

**[ 목표 ]**

1. 헬리곱터에서 미사일을 사용할 수 있다.
2. 미사일에는 날라갈때, 불꽃효과가 있음
3. 1인칭 모드일 때, 조종석이 보이는 텍스처를 보여줘야하고 적이 블랜딩되서 보여야 됨
4. 1인칭 모드일 때, 적을 포커싱 할 수 있다. 그리고, 포커싱하는 텍스처는 알아서 구함
5. 1인칭 모드일 때, 뒤에 있는 적을 백미러 거울로 볼 수 있다. – 구현 x

**[ 조작법 ]**

1. Ctrl키를 누르면 미사일을 발사할 수 있다.
2. F1키를 누르면 1인칭 모드로 변경가능, F3키를 누르면 3인칭 모드로 변경가능
3. 1인칭 모드일 때, 적을 클릭하면 포커싱 할 수 있음?

**[ 게임 이미지 및 설명]**



**[ Ctrl키를 누르면 미사일을 발사, 미사일이 날아갈 때 보여주는 FireParticle ]**

**CScene의 BuildObjects에서 미사일로 사용할 pBulletShader와 미사일이 터질 때 나오는 파티클 m\_pExplosionParticleShader, 미사일이 날라갈 때 불꽃이 나오게하는 m\_pFireParticleShader를 생성해준다.**

if (!m\_pBulletShader)

{

m\_pBulletShader = new CBulletShader;

m\_pBulletShader->CreateShader(pd3dDevice, pd3dCommandList, m\_pd3dGraphicsRootSignature);

m\_pBulletShader->BuildObjects(pd3dDevice, pd3dCommandList, NULL);

if (!m\_pExplosionParticleShader)

{

m\_pExplosionParticleShader = new CExplosionParticleShader;

m\_pExplosionParticleShader->CreateShader(pd3dDevice, pd3dCommandList, m\_pd3dGraphicsRootSignature);

m\_pExplosionParticleShader->BuildObjects(pd3dDevice, pd3dCommandList, NULL);

}

**// 미사일 셰이더 생성하고, 미사일이 터졌을 때 생성되는 나오는 불꽃이 나올 수 있게,**

**// 미사일 셰이더에서 ExplosionParticleShader를 갖고있게 Set해준다.**

m\_pBulletShader->SetExplosionParticleShader(m\_pExplosionParticleShader);

}

if (!m\_pFireParticleShader)

{

m\_pFireParticleShader = new CFireParticleShader;

m\_pFireParticleShader->CreateShader(pd3dDevice, pd3dCommandList, m\_pd3dGraphicsRootSignature);

m\_pFireParticleShader->BuildObjects(pd3dDevice, pd3dCommandList, NULL);

}

**// CScene에서 Control 키를 누르면, 미사일 셰이더에서 미사일을 만들 수 있도록**

**// 함수를 호출해준다. 그러나 미사일 셰이더가 그전에 생성되있어야하고, 플레이어, 미사일이**

**// 날라갈 때 보여주는 불꽃셰이더이 생성되있어야 함**

bool CScene::OnProcessingKeyboardMessage(. . .)

{

switch (nMessageID)

{

case WM\_KEYDOWN:

switch (wParam)

{

// 총알 발사 키

case VK\_CONTROL:

if (m\_pBulletShader && m\_pFramePlayer && m\_pFireParticleShader)

{

**// 미사일이 움직이는 위치에 불꽃이 있어야 하기 때문에 날라갈 때 보여주는 불꽃셰이더도**

**// 미사일셰이더에 set해준다.**

m\_pBulletShader->SetParticleShader(m\_pFireParticleShader);

m\_pBulletShader->OnProcessingKeyboardMessage(. . .);

}

break;

**// CBulletShader의 BuildObject( )에서 미사일에 사용할 텍스처와, 메쉬, 재질을 공유할 수**

**// 있도록 미리 생성해놓고, 상수버퍼와, 리소스뷰를 생성한다.**

void CBulletShader::BuildObjects(. . .)

{

**// 미사일로 사용할 큐브 메쉬 생성**

m\_pBulletTexturedMesh = new CCubeMeshTextured(pd3dDevice, pd3dCommandList, 2.5f, 2.5f, 5.f);

**// 미사일로 사용할 텍스처 생성**

m\_pBulletTexture = new CTexture(1, RESOURCE\_TEXTURE2D, 0);

m\_pBulletTexture->LoadTextureFromFile(pd3dDevice, pd3dCommandList, L"Image/Bullet.dds", 0);

**// 미사일의 재질을 만들어 줌**

m\_pBulletMaterial = new CMaterial;

m\_pBulletMaterial->SetTexture(m\_pBulletTexture);

UINT ncbElementBytes = ((sizeof(CB\_GAMEOBJECT\_INFO) + 255) & ~255);

**// 미사일의 최대 개수는 100개로 설정**

m\_nObjects = 100;

CreateCbvSrvDescriptorHeaps(pd3dDevice, pd3dCommandList, m\_nObjects, 1);

CreateShaderVariables(pd3dDevice, pd3dCommandList);

CreateConstantBufferViews(pd3dDevice, pd3dCommandList, m\_nObjects, m\_pd3dcbGameObjects, ncbElementBytes);

**// 미사일에 사용할 텍스처는 루트파라미터 16번에 연결되어있다.**

CreateShaderResourceViews(pd3dDevice, pd3dCommandList, m\_pBulletTexture, 16, false);

}

**// 미사일이 플레이어가 Control키를 누를 때만 생성할 수 있도록 CBulletShader의 OnProcessingKeyboardMessage( )에서 처리한다. 그리고 BuildObject에서 미리 생성해놓은 미사일의 메쉬와, 텍스처 재질을 Set해주고, 미사일이 플레이어가 바로보는 Look을 설정해야함. 그래야 플레이어가 바라보는 방향대로 미사일이 날아갈 수 있다. 미사일의 위치 또한 플레이어가 있는 위치로 Set해준다.**

bool CBulletShader::OnProcessingKeyboardMessage(. . .)

{

case VK\_CONTROL:

{

if (m\_pFramePlayer)

{

CGameObject\* pBullet = new CBullet;

**// 미사일마다 ID를 부여해준다. 왜냐하면, 미사일에 해당하는 ID에 미사일이 날아갈 때**

**// 보여주는 불꽃 파티클의 ID를 동일하게 주어, 삭제할 때,**

**// 해당하는 ID의 불꽃 파티클을 지워주기 위함이다.**

((CBullet\*)pBullet)->SetID(++m\_BulletCount);

**// 미리 만들어 놓은 메쉬와, 재질을 Set해준다.**

pBullet->SetMesh(m\_pBulletTexturedMesh);

pBullet->SetMaterial(m\_pBulletMaterial);

**// 플레이어의 LookVector를 주어야 미사일이 플레이어가 바라보는 방향을 향한다.**

pBullet->SetLook(m\_pFramePlayer->GetLookVector());

**// 또한 플레이어의 Up벡터, Right벡터도 똑같이 설정해주어야 플레이어가 회전했을 때,**

**// 미사일의 모양도 회전이 된 모양으로 바뀐다.**

pBullet->SetUp(m\_pFramePlayer->GetUpVector());

pBullet->SetRight(m\_pFramePlayer->GetRightVector());

**// 미사일의 생성위치는 플레이어의 위치로 설정**

pBullet->SetPosition(m\_pFramePlayer->GetPosition());

//pBullet->SetPosition(m\_pFramePlayer->GetCamera()->GetPosition());

**// 미사일이 나아가는 방향은 미사일이 바라보는 방향으로 준다.**

pBullet->SetMovingDirection(pBullet->GetLook());

**// 마지막으로 미사일을 삽입, 삭제에 효율적인 List로 관리해주다.**

m\_BulletList.push\_back(pBullet);

cout << "총알 생성 됨" << endl;

**// 미사일이 생성되었으므로, 미사일이 날아갈 때 나오는 불꽃 파티클도 생성하면서**

**// m\_BulletCount라는 미사일의 ID를 넘겨준다.**

m\_pFireParticleShader->Initialize(pBullet, m\_BulletCount);

}

**// 미사일의 Animate함수이다. 미사일은 유효 사거리가 존재해야 한다.**

**// 따라서, 미사일이 플레이어와 떨어진 거리를 계산하여, 유효사거리가 되면 미사일은 충돌되지**

**// 않아도 터져야한다. 그러면서 미사일을 날아갈 때 보여주는 불꽃 파티클도 같이 삭제해주어야 // 한다. 또한 충돌되었을 때도 마찬가지이다.**

**// 그리고 터지는 위치에서 터지는 불꽃파티클(ExplosionParticle)이 생성되어야 한다.**

**// 이러한 경우가 아니면, 계속 미사일은 자전하면서 움직여야 한다.**

void CBulletShader::AnimateObjects(float fTimeElapsed)

{

**// 미사일이 플레이어와 떨어진 거리를 계산하기위한 변수이다.**

double distance = 0.f;

if (m\_pExplosionParticleShader)

{

**// 플레이어의 미사일 List를 루프를 통해 순회하면서, 미사일마다 애니메이트 시켜준다.**

for (auto iter = m\_BulletList.begin(); iter != m\_BulletList.end();)

{

**// 미사일이 충돌되었다면,**

if (((CBullet\*)(\*iter))->GetCollision() == true)

{

**// 삭제해야하는 Fire파티클 ID를 넘겨준다. 이 ID는 미사일의 ID와 동일하므로,**

**// 해당하는 불꽃파티클ID를 삭제시키도록 ID를 넘겨준다.**

m\_pFireParticleShader->SetDeleteFireParticleID(((CBullet\*)(\*iter))->GetID());

**// 마지막에 터지는 위치에서 ExplosionParticle을 미사일의 위치에 생성해준다.**

m\_pExplosionParticleShader->Initialize((\*iter)->GetPosition());

**// 그리고 해당하는 미사일은 delete로 삭제시켜줌**

delete (\*iter);

iter = m\_BulletList.erase(iter);

}

**// 충돌된 미사일이 아니라면,**

else

{

**// 플레이어 위치와 미사일의 위치 거리를 계산하는 공식이다.**

distance = sqrt((pow(((\*iter)->GetPosition().x - m\_pFramePlayer

->GetPosition().x), 2.0)

+ pow(((\*iter)->GetPosition().y - m\_pFramePlayer->GetPosition().y), 2.0)

+ pow(((\*iter)->GetPosition().z - m\_pFramePlayer->GetPosition().z), 2.0)));

**// 플레이어와 미사일의 거리가 250m보다 커지면, 미사일의 유효사거리를 벗어난거므로**

**// 총알을 계속 그리지 않고, 지워주어야 프레임레이트를 올릴 수 있다.**

if (distance >= MaxBulletDistance)

{

**// 삭제해야하는 미사일이 날라갈 때 보여주는 FireParticle ID를 넘겨준다.**

m\_pFireParticleShader->SetDeleteFireParticleID(((CBullet\*)(\*iter))->GetID());

**// 마지막에 터지는 ExplosionParticle을 미사일의 마지막 위치에 생성**

m\_pExplosionParticleShader->Initialize((\*iter)->GetPosition());

**// 그 후에 미사일을 삭제**

delete (\*iter);

iter = m\_BulletList.erase(iter);

cout << "플레이어 총알 거리벗어남 삭제" << endl;

}

**// 충돌된 것도 아니고, 유효사거리를 벗어난 미사일이 아니면 원래대로 애니메이트 시켜준다.**

else

{

((CBullet\*)(\*iter))->Animate(fTimeElapsed);

++iter;

}

}

}

}

}

**// 다음은 미사일이 날아갈 때, 생기는 불꽃 파티클을 생성할 때 필요한 메쉬와 텍스처 리소스를// 미리 BuildObject하는 함수이다.**

void CFireParticleShader::BuildObjects(. . .)

{

**// 불꽃 파티클 메쉬 생성**

**// 미사일을 따라다니는 불꽃 파티클은 미사일의 맨 뒤에 위치해야 한다.**

**// 따라서 미사일의 모델좌표계에서 미사일이 z크기 -5만큼 뒤에 위치하게 생성해야 미사일의**

**// 맨뒤에 위치하게 된다.**

m\_pFireParticleMesh = new CTexturedRectMesh(pd3dDevice, pd3dCommandList, 3.f, 3.f, 0.0f, 0.0f, 0.0f, -**5.0f**);

**// 불꽃 파티클 텍스처 생성**

m\_pFireParticleTexture = new CTexture(1, RESOURCE\_TEXTURE2D, 0);

m\_pFireParticleTexture->LoadTextureFromFile(pd3dDevice, pd3dCommandList, L"Image/FireParticle.dds", 0);

**// 불꽃 파티클 재질 생성**

m\_pFireParticleMaterial = new CMaterial;

m\_pFireParticleMaterial->SetTexture(m\_pFireParticleTexture);

**// 불곷 파티클은 미사일과 달리 다른 struct 글로벌 변수를 사용한다.**

**// 왜냐하면, 불꽃 파티클은 프레임시트마다 변경 되어야하기 때문에 elapsedTime을 넘겨야 한다.**

**// 따라서 CB\_TEXTURE\_ANIMATION에는 int형 elapsedTime을 추가적으로 넘긴다.**

UINT ncbElementBytes = ((sizeof((CB\_TEXTURE\_ANIMATION) + 255) & ~255);

m\_nObjects = 100;

CreateCbvSrvDescriptorHeaps(pd3dDevice, pd3dCommandList, m\_nObjects, 1);

CreateShaderVariables(pd3dDevice, pd3dCommandList);

CreateConstantBufferViews(pd3dDevice, pd3dCommandList, m\_nObjects, m\_pTextureAnimation, ncbElementBytes);

**// 불꽃 텍스처는 루트파라미터 17번에 연결되어 있다.**

CreateShaderResourceViews(pd3dDevice, pd3dCommandList, m\_pFireParticleTexture, 17, false);

}

**// 다음은 미사일이 생성될 때 불꽃 파티클이 생성되게하는 함수이다.**

**// 인자값으로 해당하는 미사일과, 미사일의 ID를 넘겨받는다.**

bool CFireParticleShader::Initialize(CGameObject\* pBullet, int id)

{

if (pBullet)

{

CGameObject\* pParticle = new CFireParticle;

**// id와 메쉬, 재질을 설정하고, 미사일과 같은 Look, Up, Right 벡터를 설정한다.**

((CFireParticle\*)pParticle)->SetID(id);

pParticle->SetMesh(m\_pFireParticleMesh);

pParticle->SetMaterial(m\_pFireParticleMaterial);

pParticle->SetLook(m\_pFramePlayer->GetLookVector());.

pParticle->SetUp(m\_pFramePlayer->GetUpVector());

pParticle->SetRight(m\_pFramePlayer->GetRightVector());

**// 마지막으로 불꽃 파티클의 위치는 인자로 받았던 해당 미사일의 위치를 설정한다.**

pParticle->SetPosition(pBullet->GetPosition());

pParticle->SetMovingDirection(pParticle->GetLook());

**// 불꽃 파티클도 미사일처럼 List로 관리한다.**

m\_FireParticleList.push\_back(pParticle);

cout << "미사일 파티클 생성 됨" << endl;

return true;

}

return false;

}

**// 다음은 미사일이 날아갈 때, 나오는 불꽃 파티클의 애니메이트 함수이다.**

void CFireParticleShader::AnimateObjects(float fTimeElapsed)

{

**// 불꽃 파티클의 List를 통해 루프를 돌면서 애니메이트를 시켜준다.**

for (auto iter = m\_FireParticleList.begin(); iter != m\_FireParticleList.end();)

{

**// m\_DeleteID의 값이 미사일이 충돌되거나 유효사거리를 벗어났을 때,**

**// 넘겨받은 DeleteID에 해당하는 불꽃 파티클을 삭제시켜준다.**

if (((CFireParticle\*)(\*iter))->GetID() == m\_DeleteID)

{

//cout << "미사일이 삭제되서 날라가는 Fire파티클 삭제" << endl;

delete (\*iter);

iter = m\_FireParticleList.erase(iter);

}

**// 미사일이 충돌되거나 유효사거리를 벗어나지 않았으므로, 계속 애니메이트 시켜준다.**

else

{

((CFireParticle\*)(\*iter))->Animate(fTimeElapsed);

++iter;

}

}

}



**[ 미사일이 터졌을 때, 발생하는 ExplosionParticle ]**

**// 다음은 미사일이 충돌되거나 유효사거리를 벗어났을 때, 터지는 ExplosionParticle의**

**// Initialize( )이다. 미사일이 날아갈 때 보여주는 FireParticle과 비슷하다.**

**// BuildObject( )는 FireParticle과 동일**

void CExplosionParticleShader::Initialize(XMFLOAT3 position)

{

**// 미사일이 터지는 위치를 인자로 넘겨받아 그위치에 생성되도록 한다.**

**// 나머지는 동일**

cout << "Explosion 파티클 생성" << endl;

CGameObject\* pExplosionParticle = new CExplosionParticle;

**pExplosionParticle->SetPosition(position);**

. . .

}

**// 다음은 ExplosionParticle의 애니메이트 부분이다. FireParticle과 비슷하지만,**

**// 삭제해주는 부분에서 ExplosionParticle이 미사일이 터지자마자 생성되고, 사라지면,**

**// 터지는 텍스처 애니메이션을 볼 수가 없다. 따라서, ExplosionParticle이 일정시간동안**

**// 보이도록 하고, 그 시간이 지나면 텍스처 애니메이션이 끝났으므로, 지워도 된다는 것을**

**// 알려주게 하여, 지우는 방식으로 했다.**

**// 즉, ExplosionParticle오브젝트에서 elapsedTime을 누적시켜 1초동안 보여줄 수 있도록함**

**// if (m\_ExplosionTime > 1.f)**

**// m\_OnExplosion = true;**

**// ExplosionParticle을 관리하는 List로 루프를 돌면서 애니메이트 시켜준다.**

for (auto iter = m\_ExplosionParticleList.begin(); iter != m\_ExplosionParticleList.end(); )

{

**// 1초동안 Explosion애니메이션을 보여주었으면 ExplosionParticle의 m\_OnExplosion 값이 true가 된다.**

**// 따라서 삭제해도 된다.**

if (((CExplosionParticle\*)(\*iter))->GetOnExplosion() == true)

{

delete (\*iter);

iter = m\_ExplosionParticleList.erase(iter);

}

**// 1초가 지나지 않아서 텍스처 애니메이션이 진행중이므로 계속 애니메이트 시킨다**

else

{

(\*iter)->Animate(fTimeElapsed);

++iter;

}

}

**// 다음은 FireParticle과 ExplosionParticle이 텍스처 애니메이션을 하게해주는 픽셀셰이더 함수**

**// 부분이다.**

float4 PSFireParticle(VS\_TEXTURED\_OUTPUT input) : SV\_TARGET

{

**// 텍스처이미지의 uv를 저장한다.**

float2 texUV = input.uv;

**// 그리고 이미지가 u로 6개의 시트가 있으므로 u를 1/6을 나누어,**

**// 상수버퍼로 넘겨받은 elapsedTime을 곱해서 u를 옆으로 이동시켜준다.**

**// u를 0 ~ 1/6까지만 설정하면, 첫번째 시트이미지가 보여지고, 그다음은 1/6 ~ 2/6, 이런식으로**

**// elapsedTime에 따라 u를 이동시켜주어 텍스처 애니메이션을 만든다.**

texUV.x = texUV.x / 6.0f + elapsedTime \* (1.0f / 6.0f);

float4 cColor = FireParticleTexture.Sample(gWrapSamplerState, texUV);

return(cColor);

}



**[ 1인칭 모드일 때, 조종석UI와 적을 블랜딩]**

**// 1인칭 모드일 때, 조종석 텍스처를 보여주어야 하는 것을 UI로 처리하였다.**

**// UI로 처리하면, 플레이어 회전을 하더라도, 조종석 텍스처는 항상 화면 좌표계에 그려진다.**

**// UI를 만들기 위해서 빌보드를 이용하였다.**

**// 1인칭 UI는 CGameFramework( )의 BuildObject( )에서 UIShader( )를 빌드해준다.**

void CGameFramework::BuildObjects()

{

. . .

**// 1인칭 모드일 때, 보여지는 UI(조종석 텍스처)를 미리 빌드 한다.**

if (!m\_pFirstPersonUIShader)

{

m\_pFirstPersonUIShader = new CUIShader;

m\_pFirstPersonUIShader->CreateShader(m\_pd3dDevice, m\_pd3dCommandList, m\_pScene->GetGraphicsRootSignature());

m\_pFirstPersonUIShader->BuildObjects(m\_pd3dDevice, m\_pd3dCommandList, NULL);

}

}

**// 그리고 CGameFramework의 OnProcessingKetboardMessage( )에서 F1키를 눌렀을 때,**

**// UI 객체를 생성하도록 하고, F2(스페이스 모드), F3(3인칭 모드)로 가면 1인칭 UI를 삭제하도록 한다.**

case VK\_F1:

**// F1키를 누르면 UI 객체를 생성해주고, 플레이어의 위치를 카메라의 위치로 바꿔준다.**

if (m\_pFirstPersonUIShader)

m\_pFirstPersonUIShader->Initialize();

m\_pCamera = m\_pFramePlayer->ChangeCamera(FIRST\_PERSON\_CAMERA, m\_GameTimer.GetTimeElapsed());

m\_pFramePlayer->SetPosition(m\_pCamera->GetPosition());

break;

**// 1인칭 모드에서 스페이스, 3인칭모드로 바뀌면, 1인칭 UI 객체를 지워준다.**

case VK\_F2:

if (m\_pFirstPersonUIShader->GetFirstPersonUI() != nullptr)

m\_pFirstPersonUIShader->DeleteUI();

break;

case VK\_F3:

if (m\_pFirstPersonUIShader->GetFirstPersonUI() != nullptr)

m\_pFirstPersonUIShader->DeleteUI();

break;

**// 다음은 CUIShader( )의 BuildObject( )와 객체를 생성해주는 Initialize( ) 부분이다.**

void CUIShader::BuildObjects(. . .)

{

**// 1인칭 UI(조종석 모드)를 사용할 빌보드 메쉬를 미리 생성해준다.**

m\_pFirstPersonUIMesh = new CTexturedRectMesh(pd3dDevice, pd3dCommandList, 20.f, 20.f, 0.0f, 0.0f, 0.0f, 0.0f);

**// 1인칭 UI 텍스처와 재질, 리소스를 생성해준다.**

m\_pFirstPersonUITexture = new CTexture(1, RESOURCE\_TEXTURE2D, 0);

m\_pFirstPersonUITexture->LoadTextureFromFile(pd3dDevice, pd3dCommandList, L"Image/FirstPersonUI.dds", 0);

m\_pFirstPersonUIMaterial = new CMaterial;

m\_pFirstPersonUIMaterial->SetTexture(m\_pFirstPersonUITexture);

m\_nObjects = 1;

CreateCbvSrvDescriptorHeaps(pd3dDevice, pd3dCommandList, m\_nObjects, 1);

CreateShaderResourceViews(pd3dDevice, pd3dCommandList, m\_pFirstPersonUITexture, 3, false);

}

bool CUIShader::Initialize()

{

if (!m\_pFirstPersonUI)

{

**// 1인칭 UI를 객체를 생성해준다**

m\_pFirstPersonUI = new CFirstPersonUI;

**// BuildObject에서 미리 생성해놓은 UI 메쉬와 재질을 Set해준다.**

m\_pFirstPersonUI->SetMesh(m\_pFirstPersonUIMesh);

m\_pFirstPersonUI->SetMaterial(m\_pFirstPersonUIMaterial);

cout << "1인칭 UI 생성" << endl;

return true;

}

return false;

}

**// 다음은 UI를 화면 좌표계에 보여주게하는 정점 셰이더 부분이다.**

**// 정점 셰이더의 인풋으로 정점의 ID를 입력받는다. 입력된 정점의 ID로 구분하여**

**// 위치와 텍스처 uv를 지정해주기 위함이다.**

VS\_TEXTURED\_OUTPUT VSFirstPersonUI(uint nVertexID : SV\_VertexID)

{

VS\_TEXTURED\_OUTPUT output;

if (nVertexID == 0)

{

output.position = float4(-1.f, 1.f, 0.f, 1.f);

output.uv = float2(1.f, 0.f);

}

else if (nVertexID == 1)

{

output.position = float4(-1.f, -1.f, 0.f, 1.f);

output.uv = float2(1.f, 1.f);

}

else if (nVertexID == 2)

{

output.position = float4(1.f, -1.f, 0.f, 1.f);

output.uv = float2(0.f, 1.f);

}

else if (nVertexID == 3)

{

output.position = float4(1.f, -1.f, 0.f, 1.f);

output.uv = float2(0.f, 1.f);

}

else if (nVertexID == 4)

{

output.position = float4(1.f, 1.f, 0.f, 1.f);

output.uv = float2(0.f, 0.f);

}

else if (nVertexID == 5)

{

output.position = float4(-1.f, 1.f, 0.f, 1.f);

output.uv = float2(1.f, 0.f);

}

return(output);

}

**// 그 외, 플레이어의 미사일과 적의 충돌 체크하는 부분이다.**

**// 충돌체크를 해주는 부분은 CScene 클래스에서 관리한다. 왜냐하면, CScene에서**

**// 모든 객체를 관리하고 있기 때문이다.**

void CScene::CheckObjectByObjectCollisions()

{

if (m\_pFramePlayer)

{

**// 미사일셰이더와 적 오브젝트가 존재할 때,**

**// 미사일을 관리해주는 List를 CBulletShader에서 넘겨받아 for문으로 루프를 돌면서**

**// 바운딩 박스끼리 intersects해준다. 만약 충돌이 되었을 때,**

**// 해당하는 미사일과 해당하는 적 헬기를 충돌되었다고 SetCollision을 true로 설정해줌**

**// 충돌이 되면, 각 셰이더에서 충돌 된 오브젝트를 처리하면 된다.**

if (m\_ppFrameObjects && m\_pBulletShader)

{

list<CGameObject\*> pBulletList = m\_pBulletShader->GetBulletList();

for (auto iter = pBulletList.begin(); iter != pBulletList.end(); ++iter)

{

for (int i = 0; i < m\_nFrameObjects; ++i)

{

if ((\*iter)->m\_xmOOBB.Intersects(m\_ppFrameObjects[i]->m\_xmOOBB))

{

(\*iter)->SetFrameObjectCollided(m\_ppFrameObjects[i]);

m\_ppFrameObjects[i]->SetObjectCollided((\*iter));

cout << i << "번째 몬스터랑 충돌 됨" << endl;

((CBullet\*)(\*iter))->SetCollision(true);

}

}

}

}

}

}

**// 적 헬기가 플레이어를 쫓아오게 하는 부분이다.**

void CFrameObject::Animate(float fTimeElapsed, XMFLOAT4X4 \*pxmf4x4Parent)

{

. . .(생략)

if (m\_Type == FRAME\_ENEMY && m\_pTarget)

{

**// 적 헬기의 월드 좌표를 임시변수에 넣어준다**

XMFLOAT3 xmf3Position(m\_xmf4x4World.\_41, m\_xmf4x4World.\_42, m\_xmf4x4World.\_43);

**// 플레이어의 위치도 넣어준다.**

XMFLOAT3 xmf3TargetPosition = m\_pTarget->GetPosition();

**// 타겟인 플레이어의 위치 벡터와, 적 헬기의 위치벡터를 빼주고, 정규화를 해준다.**

XMFLOAT3 xmf3ToTarget = Vector3::Normalize(Vector3::Subtract(xmf3TargetPosition, xmf3Position));

**// 위에서 구해준 벡터와, 적 헬기의 룩벡터를 내적을 하여, 회전할 각도를 구해준다.**

float fDotProduct = Vector3::DotProduct(Vector3::Normalize(m\_xmf3Look), xmf3ToTarget);

float fAngle = ::IsEqual(fDotProduct, 1.0f) ? 0.0f : ((fDotProduct > 0.0f) ? XMConvertToDegrees(acos(fDotProduct)) : 90.0f);

**// 회전을 할 축을 구해야하므로 외적을 사용한다.**

**// 내적 때 사용했던 플레이어 위치벡터 - 적 헬기의 위치벡터와 룩벡터를 외적한다.**

XMFLOAT3 xmf3CrossProduct = Vector3::CrossProduct(Vector3::Normalize(m\_xmf3Look), xmf3ToTarget);

if (xmf3CrossProduct.x > 0.f)

xmf3CrossProduct.x = 1.0f;

else

xmf3CrossProduct.x = -1.0;

if (xmf3CrossProduct.y > 0.f)

xmf3CrossProduct.y = 1.0f;

else

xmf3CrossProduct.y = -1.0;

if (xmf3CrossProduct.z > 0.f)

xmf3CrossProduct.z = 1.0f;

else

xmf3CrossProduct.z = -1.0;

**// 외적, 내적한 결과 값을 바탕으로 회전을 시켜준다.**

**// 그러나 헬기가 x와 z축을 회전하는 일은 없어야 하므로, y축만 회전 시킨다.**

Rotate(0, fAngle \* fTimeElapsed \* ((xmf3CrossProduct.y > 0.0f) ? 1.0f : -1.0f), 0);

**// 마지막에 적 헬기의의 바뀐 Right, Up, Look벡터를 월드변환 행렬에서**

**// 가져와 업데이트 시킨다.**

m\_xmf3Right.x = m\_xmf4x4World.\_11; m\_xmf3Right.y = m\_xmf4x4World.\_12; m\_xmf3Right.z = m\_xmf4x4World.\_13;

m\_xmf3Up.x = m\_xmf4x4World.\_21; m\_xmf3Up.y = m\_xmf4x4World.\_22; m\_xmf3Up.z = m\_xmf4x4World.\_23;

m\_xmf3Look.x = m\_xmf4x4World.\_31; m\_xmf3Look.y = m\_xmf4x4World.\_32; m\_xmf3Look.z = m\_xmf4x4World.\_33;

**// 플레이어와 몬스터의 거리가 5이상일 때,**

if (Vector3::Distance(xmf3TargetPosition, xmf3Position) > DISTANCE\_TO\_TARGET\_OBJECT)

{

XMFLOAT3 xmf3Look = GetLook();

**// 플레이어의 위치와 몬스터의 위치를 빼주고, 이동속도만큼 곱해준다.**

XMFLOAT3 Temp = Vector3::ScalarProduct(Vector3::SubtractNormalize(xmf3TargetPosition, xmf3Position, true), m\_fMovingSpeed \* fTimeElapsed);

**// 그 값을 적 헬기의 위치와 더해주어 따라오게 만든다.**

SetPosition(Vector3::Add(xmf3Position, Temp));

}

}

m\_xmOOBBTransformed.Transform(m\_xmOOBB, XMLoadFloat4x4(&m\_xmf4x4World));

XMStoreFloat4(&m\_xmOOBBTransformed.Orientation, XMQuaternionNormalize(XMLoadFloat4(&m\_xmOOBBTransformed.Orientation)));

}

**// 플레이어가 적을 피킹하는 부분이다.**

bool CScene::OnProcessingMouseMessage(HWND hWnd, UINT nMessageID, WPARAM wParam, LPARAM lParam, CCamera\* pCamera)

{

switch (nMessageID)

{

case WM\_LBUTTONDOWN:

case WM\_RBUTTONDOWN:

**// 마우스가 눌려지면 마우스 픽킹을 하여 선택한 게임 객체를 찾는다.**

**// 피킹된 오브젝트가 있으면 m\_pSelectedObject가 null이 아니라 선택한 오브젝트가 됨**

m\_pSelectedObject = PickObjectPointedByCursor(LOWORD(lParam), HIWORD(lParam), pCamera);

if (m\_pSelectedObject)

cout << "피킹됨" << endl;

else

cout << "피킹안됨" << endl;

break;

}

CFrameObject \*CScene::PickObjectPointedByCursor(int xClient, int yClient, CCamera \*pCamera)

{

if (!pCamera)

return(NULL);

XMFLOAT4X4 xmf4x4View = pCamera->GetViewMatrix();

XMFLOAT4X4 xmf4x4Projection = pCamera->GetProjectionMatrix();

D3D12\_VIEWPORT d3dViewport = pCamera->GetViewport();

XMFLOAT3 xmf3PickPosition;

**// 화면 좌표계의 점 (xClient, yClient)를 화면 좌표 변환의 역변환과 투영 변환의 역변환을 한다.**

**// 그 결과는 카메라 좌표계의 점이다. 투영 평면이 카메라에서 z-축으로 거리가 1이므로 z-좌표는 // 1로 설정한다.\***

xmf3PickPosition.x = (((2.0f \* xClient) / d3dViewport.Width) - 1) / xmf4x4Projection.\_11;

xmf3PickPosition.y = -(((2.0f \* yClient) / d3dViewport.Height) - 1) / xmf4x4Projection.\_22;

xmf3PickPosition.z = 1.0f;

int nIntersected = 0;

float fHitDistance = FLT\_MAX, fNearestHitDistance = FLT\_MAX;

CFrameObject\* pSelectedObject = NULL;

**// 현재 CScene에 있는 적 헬기의 개수만큼 루프를 돌면서 피킹이 되었는지 비교한다.**

for (int i = 0; i < m\_nFrameObjects; ++i)

{

nIntersected = m\_ppFrameObjects[i]->PickObjectByRayIntersection(xmf3PickPosition, xmf4x4View, &fHitDistance);

if ((nIntersected > 0) && (fHitDistance < fNearestHitDistance))

{

fNearestHitDistance = fHitDistance;

pSelectedObject = m\_ppFrameObjects[i];

}

}

return pSelectedObject;

}

void CFrameObject::GenerateRayForPicking(XMFLOAT3& xmf3PickPosition, XMFLOAT4X4& xmf4x4View, XMFLOAT3 \*pxmf3PickRayOrigin, XMFLOAT3 \*pxmf3PickRayDirection)

{

XMFLOAT4X4 xmf4x4WorldView = Matrix4x4::Multiply(m\_xmf4x4World, xmf4x4View);

XMFLOAT4X4 xmf4x4Inverse = Matrix4x4::Inverse(xmf4x4WorldView);

XMFLOAT3 xmf3CameraOrigin(0.0f, 0.0f, 0.0f);

**//카메라 좌표계의 원점을 모델 좌표계로 변환한다.**

\*pxmf3PickRayOrigin = Vector3::TransformCoord(xmf3CameraOrigin, xmf4x4Inverse);

**//카메라 좌표계의 점(마우스 좌표를 역변환하여 구한 점)을 모델 좌표계로 변환한다.**

\*pxmf3PickRayDirection= Vector3::TransformCoord(xmf3PickPosition, xmf4x4Inverse);

**//광선의 방향 벡터를 구한다.**

\*pxmf3PickRayDirection = Vector3::Normalize(Vector3::Subtract(\*pxmf3PickRayDirection, \*pxmf3PickRayOrigin));

}

int CFrameObject::PickObjectByRayIntersection(XMFLOAT3& xmf3PickPosition, XMFLOAT4X4& xmf4x4View, float \*pfHitDistance)

{

int nIntersected = 0;

if (m\_pMesh)

{

XMFLOAT3 xmf3PickRayOrigin, xmf3PickRayDirection;

**//모델 좌표계의 광선을 생성한다.**

GenerateRayForPicking(xmf3PickPosition, xmf4x4View, &xmf3PickRayOrigin, &xmf3PickRayDirection);

**//모델 좌표계의 광선과 메쉬의 교차를 검사한다.**

nIntersected = m\_pMesh->CheckRayIntersection(xmf3PickRayOrigin, xmf3PickRayDirection, pfHitDistance);

}

return nIntersected;

}

int CFrameMesh::CheckRayIntersection(XMFLOAT3& xmf3RayOrigin, XMFLOAT3& xmf3RayDirection, float \*pfNearHitDistance)

{

**// 하나의 메쉬에서 광선은 여러 개의 삼각형과 교차할 수 있다.**

**// 교차하는 삼각형들 중 가장 가까운 삼각형을 찾는다.**

int nIntersections = 0;

XMFLOAT3\* pbPositions = m\_pxmf3Positions;

int nOffset = (m\_d3dPrimitiveTopology == D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST) ? 3 : 1;

int m\_nStride = sizeof(XMFLOAT4);

**// 메쉬의 프리미티브(삼각형)들의 개수이다.**

**// 삼각형 리스트인 경우 (정점의 개수 / 3) 또는 (인덱스의 개수 / 3),**

**// 삼각 형 스트립의 경우 (정점의 개수 - 2) 또는 (인덱스의 개수 – 2)이다.**

int nPrimitives = (m\_d3dPrimitiveTopology == D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST) ? (m\_nVertices / 3) : (m\_nVertices - 2);

if (m\_nSubMeshes > 0)

nPrimitives = (m\_d3dPrimitiveTopology == D3D\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST) ? (m\_nSubMeshes / 3) : (m\_nSubMeshes - 2);

**// 광선은 모델 좌표계로 표현된다.**

XMVECTOR xmRayOrigin = XMLoadFloat3(&xmf3RayOrigin);

XMVECTOR xmRayDirection = XMLoadFloat3(&xmf3RayDirection);

**//모델 좌표계의 광선과 메쉬의 바운딩 박스(모델 좌표계)와의 교차를 검사한다.**

bool bIntersected = m\_xmOOBB.Intersects(xmRayOrigin, xmRayDirection, \*pfNearHitDistance);

**//모델 좌표계의 광선이 메쉬의 바운딩 박스와 교차하면 메쉬와의 교차를 검사한다.**

if (bIntersected)

{

float fNearHitDistance = FLT\_MAX;

**// 메쉬의 모든 프리미티브(삼각형)들에 대하여 픽킹 광선과의 충돌을 검사한다.**

**// 충돌하는 모든 삼각형을 찾아 광선의 시작점(실제로는 카메라 좌표계의 원점)에**

**// 가장 가까운 삼각형을 찾는다.**

for (int i = 0; i < nPrimitives; i++)

{

XMVECTOR v0 = XMLoadFloat3((XMFLOAT3 \*)(pbPositions + ((m\_pnSubSetIndices) ? (m\_pnSubSetIndices[(i\*nOffset) + 0]) : ((i\*nOffset) + 0)) \* m\_nStride));

XMVECTOR v1 = XMLoadFloat3((XMFLOAT3 \*)(pbPositions + ((m\_pnSubSetIndices) ? (m\_pnSubSetIndices[(i\*nOffset) + 1]) : ((i\*nOffset) + 1)) \* m\_nStride));

XMVECTOR v2 = XMLoadFloat3((XMFLOAT3 \*)(pbPositions + ((m\_pnSubSetIndices) ? (m\_pnSubSetIndices[(i\*nOffset) + 2]) : ((i\*nOffset) + 2)) \* m\_nStride));

float fHitDistance;

BOOL bIntersected = TriangleTests::Intersects(xmRayOrigin, xmRayDirection, v0, v1, v2, fHitDistance);

if (bIntersected)

{

if (fHitDistance < fNearHitDistance)

\*pfNearHitDistance = fNearHitDistance = fHitDistance;

nIntersections++;

}

}

}

return(nIntersections);

}

**// 마지막, 플레이어의 백미러를 만들기위해 스텐실을 이용하였지만, 구현을 하지 못했습니다.**

**// 왜냐하면, 평면거울을 만들 때, 거울에 반사되는 객체가 정해져있는 것이 아니라, 플레이어가**

**// 움직일 때마다, 실시간으로 바뀌어야 되는데, 그 부분을 처리하지 못했습니다. 죄송합니다.**