기술소개서

양 시 몬

010 – 3296 - 0692

simonmatthew@naver.com

INDEX

기 술 소 개 서

게임소개

게임 장르 : RPG , 모험, 레이싱

제작 기간 : 19. 05. 10 ~ 19. 07. 10 (약 2달)

개발 언어 : C++, DirectX, HLSL, MFC

U R L (게임) : https://youtu.be/FZTZ4b0ZnBQ

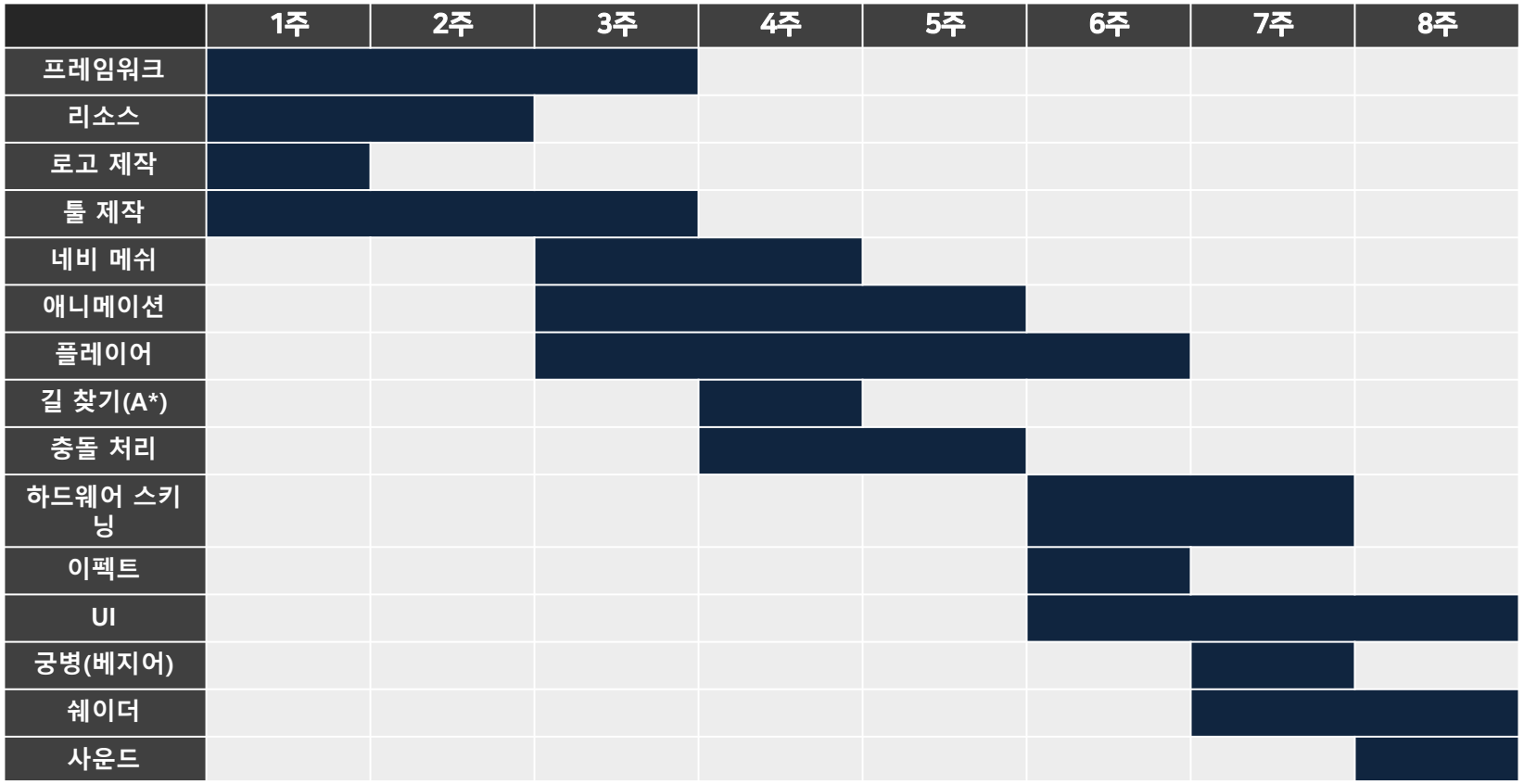
U R L (툴) : https://youtu.be/fwOmLR2OqtY

1. 마운트 앤 블레이드

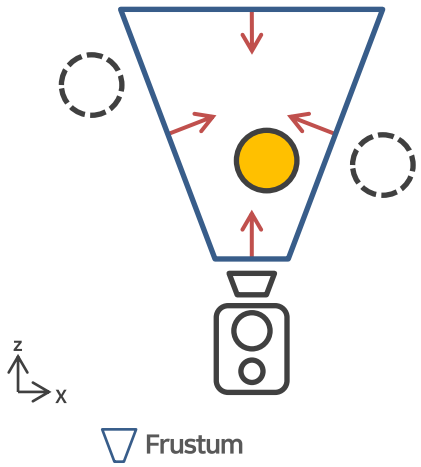
DirectX3D(개인)

개발일정

- 제작기간 : 19. 05. 10 ~ 19. 07. 10 (약 2달)



절두체 컬링



- 절두체 컬링을 이용해 화면 밖에 있는 객체들의 렌더 콜을 줄일 수 있어 불필요한 연산을 피함.

```
D3DXPLANE    tPlane[FRUSTOM_PLANE_END];
if (FAILED(Make_Plane(vFrustum_Pos, tPlane)))
{
    _MSG_BOX("CFrustum::Make_Plane()");
    return false;
}

_float fDistance = 0.f;

for (size_t iIndex = 0; iIndex < FRUSTOM_PLANE_END; ++iIndex)
{
    fDistance = D3DXPlaneDotCoord(&tPlane[iIndex], &vObjPos_World);
    if (fRadius < fDistance)
    {
        return false;
    }
}
```

6개의 평면 구성

평면과 점 사이의 거리를 알려주는 함수

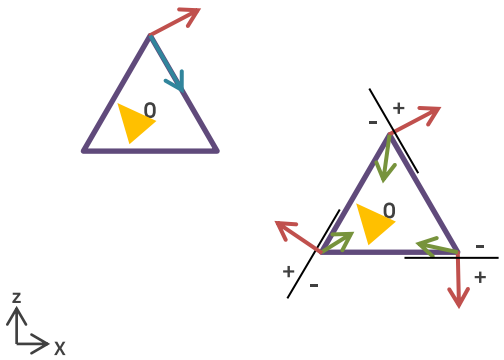
반지름보다 거리가 크면 false 리턴

< In Frustom >

< Out Frustom >

네비 메쉬

1. 현재 인덱스 검사



Player Nav Mesh

Normal Edge Axis

- 현재 어느 인덱스에 있는지 판별함.

```
_bool CCell::Is_In(const _vec3 * pPos, EDGE * _pOutEdge)
{
    for (size_t iIndex = 0; iIndex < EDGE_END; ++iIndex)
    {
        _vec3 vNormal = { -(m_vEdge[iIndex].z) , ZERO_VALUE, m_vEdge[iIndex].x };

        _vec3 vAxis = (*pPos) - m_vPoint[iIndex];

        if (0 < D3DXVec3Dot(D3DXVec3Normalize(&vNormal, &vNormal), D3DXVec3Normalize(&vAxis, &vAxis)))
        {
            *_pOutEdge = EDGE(iIndex);
            return false;
        }
    }
    return true;
}
```

엣지벡터를 90도회전 시켜 노멀벡터 만듦

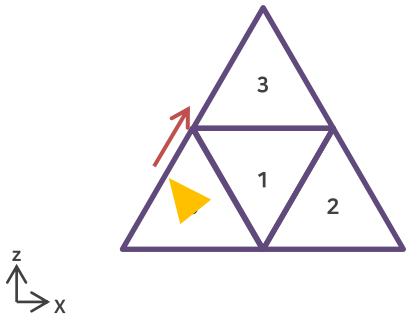
각 꼭지점에서 플레이어 위치까지 벡터를 만듦

노멀, 엑시스 단위 벡터를 내적해 Cos값을 알아낸 후
한 번이라도 양수 값이 나오면 인덱스를 넘어간 거로 판단

< In NavMesh >

네비 메쉬

2. 슬라이딩 벡터



Player Nav Mesh

Sliding Vector

- 네비 메쉬를 벗어나면 슬라이딩 벡터를 따라 미끄러지도록 만듦.

```
_vec3 vNowPos = *_pTransComp->Get_StateInfo(CTransform_Comp::STATE_POS);
_vec3 vDir = (*_pPrePos) - vNowPos;

_vec3 vNormal = (m_vecCell[m_iNowIndex]->Get_EdgeNormal(eEdge)) ;
INVERSE_VECTOR(vNormal)
D3DXVec3Normalize(&vNormal, &vNormal);

_vec3 vProjection = vNormal;
_float fDotResult = D3DXVec3Dot(&vNormal, &(-vDir));
vProjection *= fDotResult;
```

→ 노멀, 방향 벡터를 내적해 투영벡터를 구함

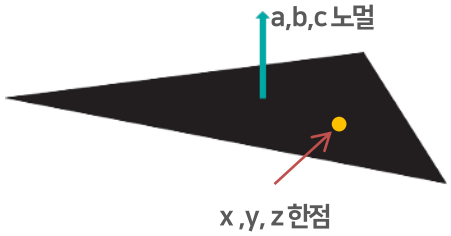
```
vDir += vProjection;
D3DXVec3Normalize(&vDir, &vDir);

//슬라이딩 벡터
_vec3 vSliding = vNowPos + (vDir * ((_fSpeed ) * _rfDeltatime) );
```

→ 방향벡터에 투영벡터를 더해 슬라이딩 벡터 구함

< Sliding Vector >

네비 메쉬

3. 네비 메쉬 태우기



- 평면의 방정식(ax + by + cz + d = 0)을 이용해 높이 값 구해냄.

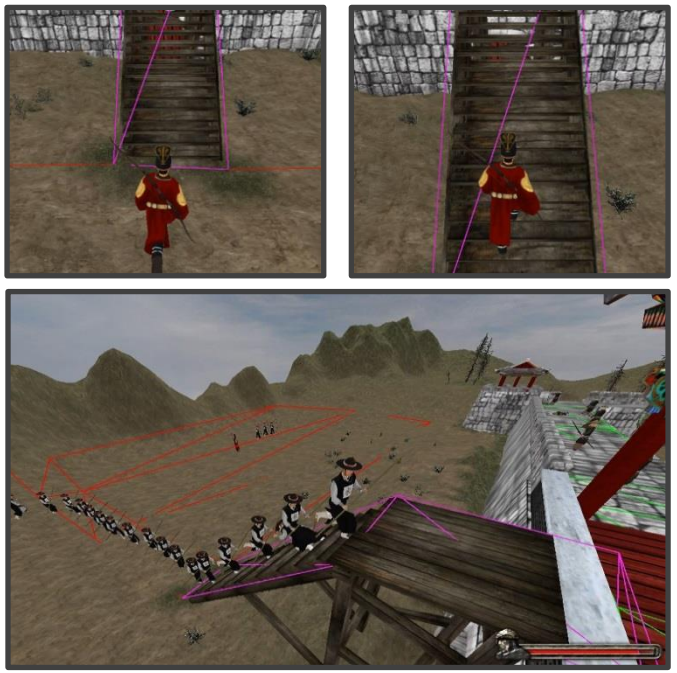
```
//ax + by + cz  + d = 0
_vec3 vNormal = m_vecCell[m_iNowIndex]->Get_Normal();

//ax + by + cz  = -d
// - N * 한점  = d;
_float fD = -D3DXVec3Dot(&vNormal, m_vecCell[m_iNowIndex]->Get_Point(CCell::POINT_A));
```

→ 노멀과 한 점을 내적해서 d값을 구해냄

```
// ax + cz + d   / -b = y
_float fPosY = ((vNormal.x * vPos.x) + (vNormal.z * vPos.z) + fD) / -vNormal.y;
```

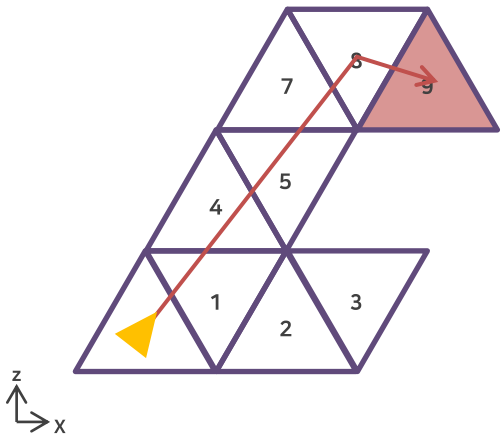
→ Y값을 구해냄

```
vPos.y = fPosY;

_pObjTransComp->Set_StateInfo(CTransform_Comp::STATE_POS, &vPos);
```



< 네비 메쉬를 타면서 Y값 변함 >

길 찾기 알고리즘 (A*)



Player Nav Mesh Goal Index

BestList

1. 주변에 이웃한 네비 메쉬가 있는지 판별.
2. 이웃한 메쉬가 있으면 거리 계산.
3. 목표지점까지 최단거리 탐색.
4. 객체에게 최단거리 넘겨줌.

```
if (0 >= m_fAstarInterval)
{
    m_pAstarMgr->Astar_Start(pNavMesh->Get_NowIndex(), iEnemyIndex, &m_BestList, pNavMesh, true);
    m_fAstarInterval = fEnemyDis;
}
else
    m_fAstarInterval -= _rfDeltaTime;

Update_AstarMove();
```

매 프레임 최단 거리 탐색 시 프레임 드랍이
생기기에 적과의 거리에 따라 호출 빈도 설정

··· 이하 생략 ···

```
if (nullptr != pNeighborCell)
{
    _rIndex = pNeighborCell->Get_Index();
    if (CheckList_IsANew(_rIndex))
    {
        _pNode = Make_Node(_rIndex, _pNodeRoot);
        m_OpenList.push_back(_pNode);
    }
}
```

주변에 이웃한 네비 메쉬가 있다면 탐색 노드에 넣음

< A* 적용한 공성전 >

하드웨어 스키닝

- **문제점 :** 소프트웨어 스키닝으로 다수의 객체의 정점 위치를 연산하면 CPU에 부하가 생김.

해결 : 하드웨어 스키닝으로 버텍스 쉐이딩 단계에서 정점의 위치를 GPU가 연산을 처리하면서 속도 증가.

뼈의 행렬을 쉐이더로 넘겨줌.

```
_pEffect->SetMatrixArray("g_matBone_Array", pMeshContainer->pRD_Matrix_Array, pMeshContainer->dwNumPaletteEntries);
_pEffect->SetInt("g_iNumBoneInfluence", pMeshContainer->dwNumInfl);
```

HLSL_Vertex_Shader

```
for (int iIndex = 0; iIndex < iNumInfluence; ++iIndex)
{
    fLastWeight += In.vWeights[iIndex];

    vPos += fWieghtArray[iIndex] * mul(In.vPos, g_matBone_Array[iIndexArray[iIndex]]);
    vNormal += fWieghtArray[iIndex] * mul(vector(In.vNormal, 0.f), g_matBone_Array[iIndexArray[iIndex]]);
}

fLastWeight = 1.0f - fLastWeight;
vPos += mul(In.vPos, g_matBone_Array[iIndexArray[iNumInfluence]]) * fLastWeight;
vNormal +=  mul(vector(In.vNormal, 0.f), g_matBone_Array[iIndexArray[iNumInfluence]])* fLastWeight;
```

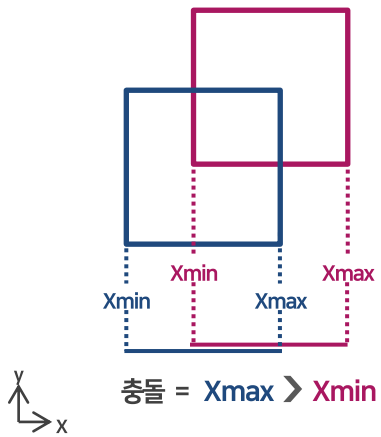
→ 정점에 영향을 주는 뼈 행렬과 가중치를 연산해 위치 , 노멀 값에 넣어줌

→ 마지막 가중치 값을 계산해 위치, 노멀 값에 더해줌

··· 이하 생략 ···

< 다수의 객체 하드웨어 스키닝 >

충돌처리(AABB)



두 박스에 최댓값 중에 작은 값과

두 박스에 최솟값 중에 큰 값을 비교한 충돌

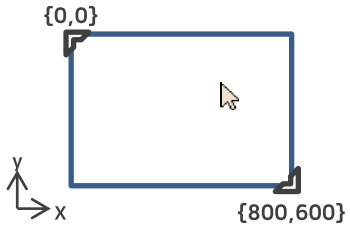
```
//X
if (min(vSourMax.x, vDestMax.x) < max(vSourMin.x, vDestMin.x))
    return false;
//Y
if (min(vSourMax.y, vDestMax.y) < max(vSourMin.y, vDestMin.y))
    return false;
//Z
if (min(vSourMax.z, vDestMax.z) < max(vSourMin.z, vDestMin.z))
    return false;


return true;
```

↳ 모든 축에 최솟값이 최댓값보다 작을 때 충돌 했음

< 장애물 박스와 객체 박스 충돌 >

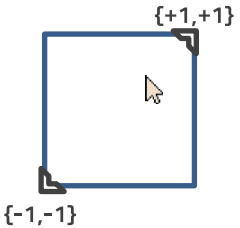
마우스 피킹

1. Screen



윈도우상의 마우스 좌표 가져옴

2. Projection



스크린 좌표를 투영 좌표로 바꿈

3. View

뷰 공간에서 원점에서 방향 벡터 생성

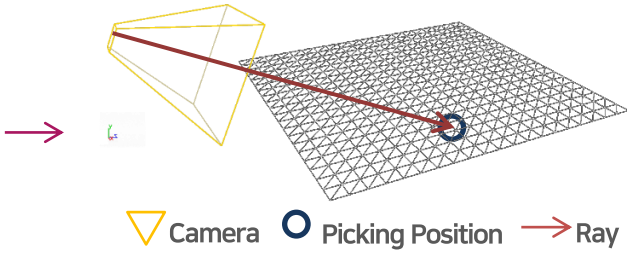
4. World

뷰 행렬의 역 행렬을 이용해 월드공간으로 이동

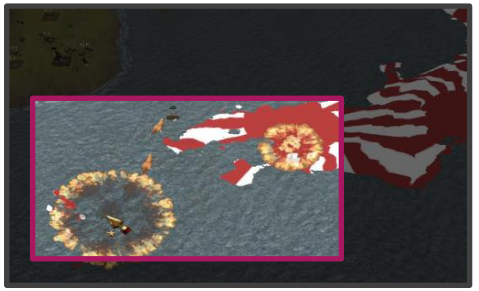
5. Obj Local

객체의 월드 행렬의 역을 이용해 로컬공간으로 이동

6. Ray Intersect



- DirectX IntersectTri 함수를 사용해 마우스 클릭 좌표로 쏜 Ray와 충돌 된 위치를 찾아 냄.

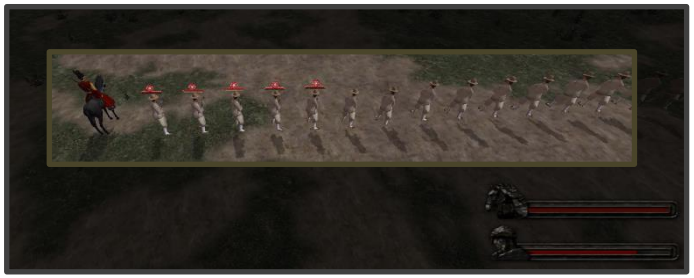


< 마우스 피킹을 이용한 폭탄 투하 >

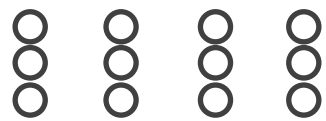
< 마우스 피킹 이동 >

부대 진형

한 줄로 플레이어 따라 옴



두 줄로 플레이어 따라 옴



세 줄로 플레이어 따라 옴

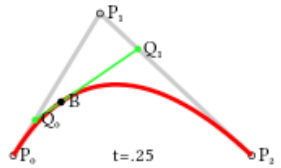
부대 진형

학익진을 그리며 플레이어 따라 옴

쥬신진을 그리며 플레이어 따라 옴

- 플레이어의 명령에 의해서 진형을 갖출 수 있도록 구현함.
- 기본적으로 플레이어를 따라다니고 공격 버튼을 누르면 공격함.

베지어곡선



P0 : 시작위치

P1 : 중간위치

P2 : 도착위치

```
ComputeBezier(vStartPos, vMiddlePos, vGoalPos, m_fBezierTime);
m_fBezierTime += fDetila;
```

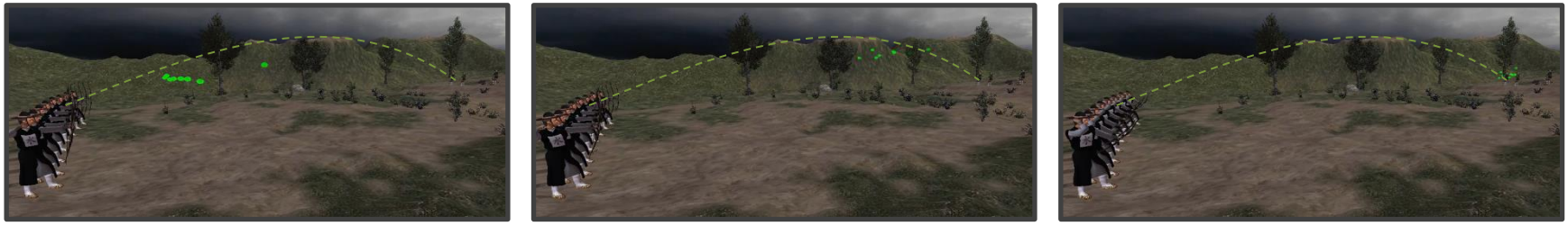
→ 처음, 중간, 끝 위치를 넘겨줌

↓

```
void CItem_ArrowCreture::ComputeBezier(const _vec3 & _rvStart, const _vec3 & _rvMiddle, const _vec3 & _rvGoal, _float _fDetail)
{
    m_vResultPos.x = _rvGoal.x * B1(_fDetail) + _rvMiddle.x * B2(_fDetail) + _rvStart.x * B3(_fDetail);
    m_vResultPos.y = _rvGoal.y * B1(_fDetail) + _rvMiddle.y * B2(_fDetail) + _rvStart.y * B3(_fDetail);
    m_vResultPos.z = _rvGoal.z * B1(_fDetail) + _rvMiddle.z * B2(_fDetail) + _rvStart.z * B3(_fDetail);

    m_pTransform_Comp->Set_StateInfo(CTransform_Comp::STATE_POS, &m_vResultPos);
}
```

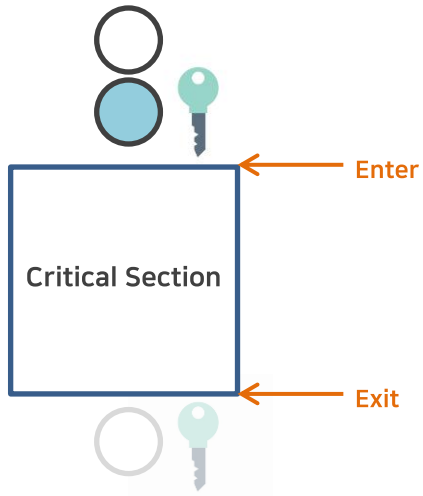
↳ 곡선의 위치를 구해냄

- 궁수들이 쏜 화살은 베지어 곡선을 이용해 목표지점에 곡선을 그리면서 날아가게 함.



< Arrow Bezier Animation >

로딩 쓰레드



키를 가지고 있는 쓰레드만 임계영역 접근

< 로딩 화면과 게임 데이터 로딩>

```
HRESULT CLoading::Ready_Loading(SCENE_ID _eSceneId)
{
    InitializeCriticalSection(&m_CriticalSection);  → 열쇠 생성
    m_hThread = (HANDLE)_beginthreadex(nullptr, m_iStackSize, ThreadMain, this, 0, nullptr);
                                                    → 쓰레드 생성
... 이하 생략 ...

EnterCriticalSection(&pLoading->m_CriticalSection);  → 열쇠가 있는 쓰레드만 접근

HRESULT hr;
switch (pLoading->m_eSceneId)
{
case SCENE_STATIC:
    hr = pLoading->Ready_SceneStatic_Comp();
    hr = pLoading->Ready_SceneStatic_Obj();
    break;
case SCENE_MAP:
    hr = pLoading->Ready_SceneMap_Comp();
    hr = pLoading->Ready_SceneMap_Obj();
    break;
default:
    break;
}
if (FAILED(hr))
    return -1;                                       로딩 데이터

LeaveCriticalSection(&pLoading->m_CriticalSection);  → 로딩이 끝나면 열쇠 반납
```

- 쓰레드를 이용해서 로딩 화면 렌더와 게임 데이터의 파일 입출력을 동시에 수행.

UI

UI_직교투영

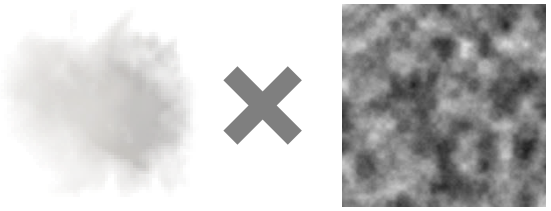
```
_pEffect->SetMatrix("g_matWorld", m_pTransform_Comp->Get_WorldMatrixPoint());

D3DXMatrixIdentity(&matView);   → 카메라를 무시하고 월드 공간의 정점을 그대로 넘김
_pEffect->SetMatrix("g_matView", &matView);

D3DXMatrixOrthoLH(&matProj, _float(g_iWINCX), _float(g_iWINCY), UI_NEAR, UI_FAR);
_pEffect->SetMatrix("g_matProj", &matProj);   → 직교 투영을 사용
```

- 원근감을 주지 않고 정사영하여 객체를 표현하는 직교투영을 사용해서 UI 표현

이펙트

← --Alpha

← ++Scale



HLSL_Pixel_Shader

```
float2 vMoveUV = float2(g_fDeltaTimeACC  , g_fDeltaTimeACC * g_fTimeMul);
                                          ↳ 노이즈 UV를 흐르게 함

vector vNoiseTex = tex2D(NoiseSampler, (In.vTexUV * g_fUvArrow) + vMoveUV);
                                          ↳ 노이즈 텍스처를 가져옴

vParticleTex.a *= (vNoiseTex.x); → 파티클 알파에 노이즈 텍스처를 곱함
```

- 스모크 한 장의 이미지를 가공해서 말들의 먼지를 만듦.
- 이미지의 크기를 시간에 따라 키우고 알파 값을 점점 옅어지게 만듦.
- 노이즈 텍스처를 흐르게 한 후 먼지 이미지 알파 값에 곱해서 같은 이미지라도 스모크 양이 다르게 보이게 함.

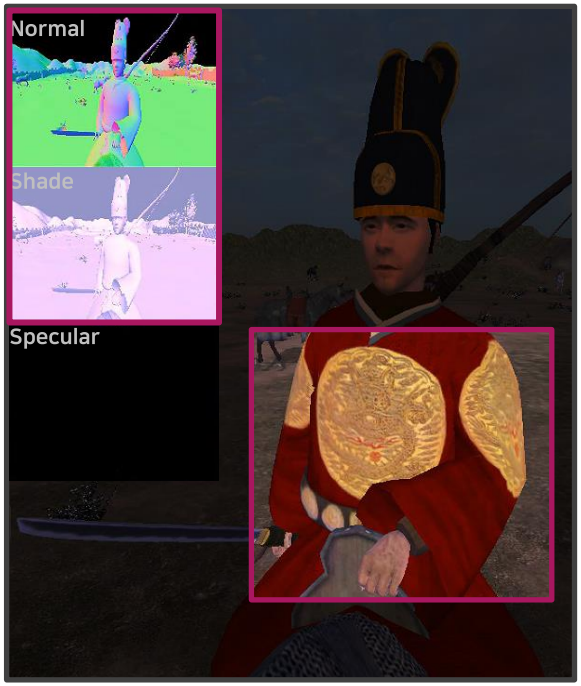
< Horse Smoke Effect >

쉐이딩

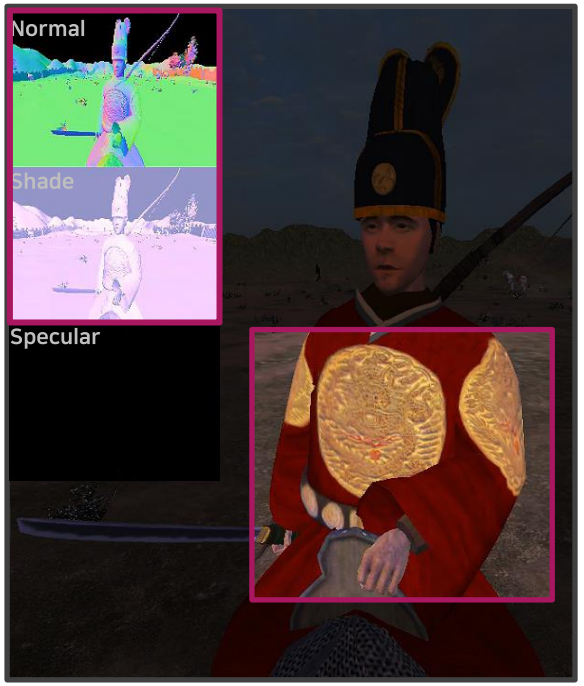
- Defferd Shader
  - 수많은 동적 라이팅을 실시간으로 보여줌.
  - 타겟을 이용해 그림자 효과를 손쉽게 그려줌.
  - 화면 픽셀 기준으로 연산하기 때문에 객체가 많아도 연산량이 일정함.

쉐이딩

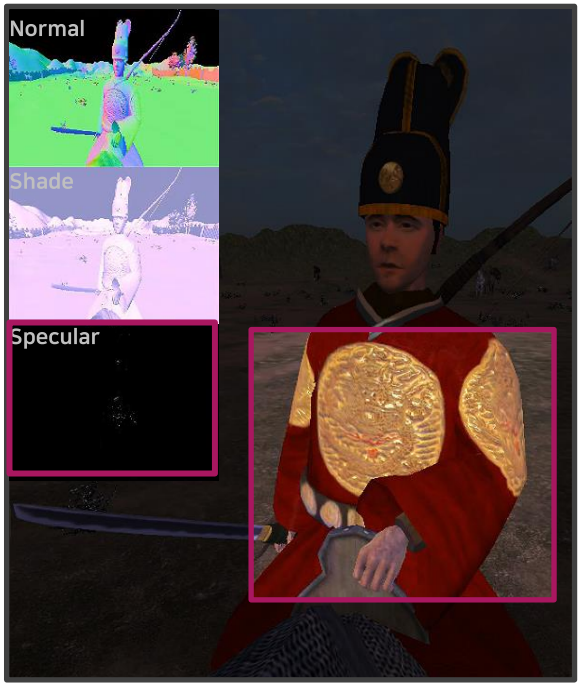
- Diffuse

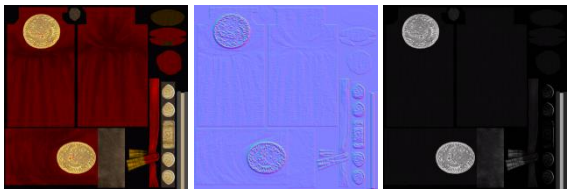


- Normal



- Specular





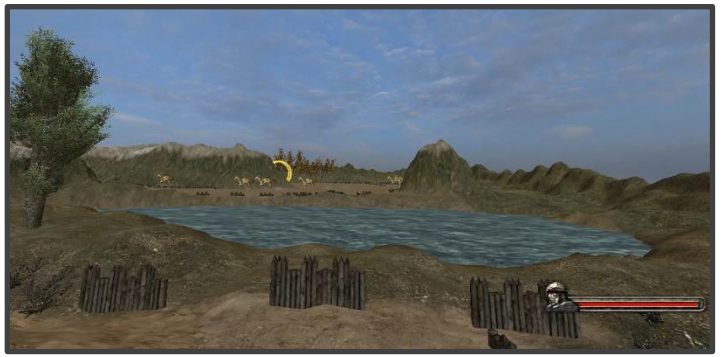
쉐이딩

- **환경 매핑**

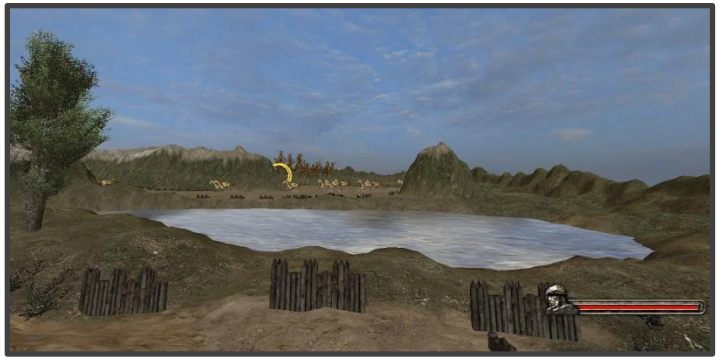
HLSL_Pixel_Shader

```
float3 vEnvironment = texCUBE(CubeSampler, vViewReflect).rgb;

Out.vDiffuse.rgb = Out.vDiffuse.rgb + (vEnvironment.rgb * g_fColor_Mul);
```

- **큐브 텍스처를 이용해서 강물에 반사되는 주변 색을 표현함.**



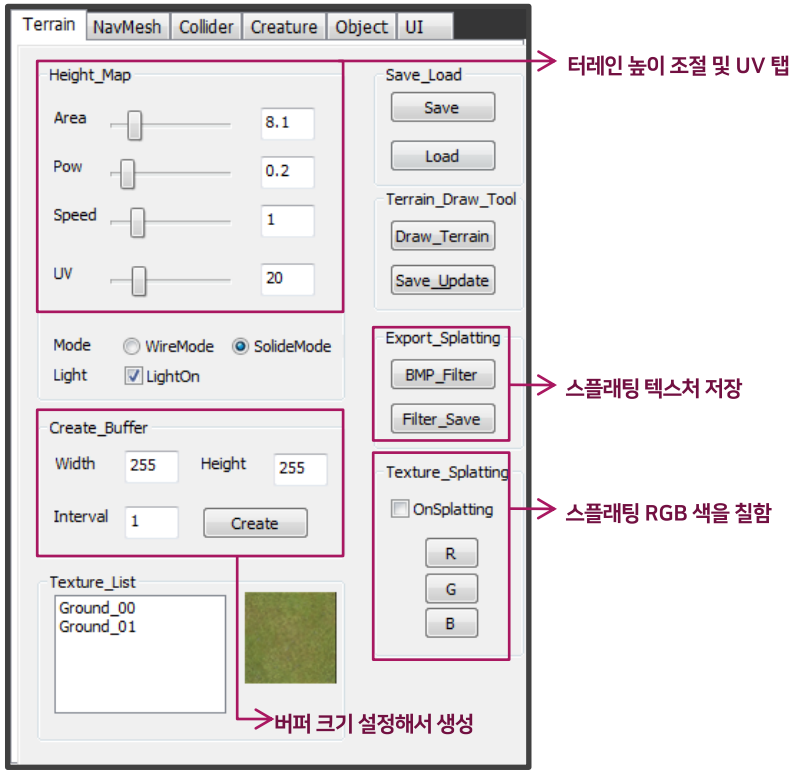
< Before >



< After >

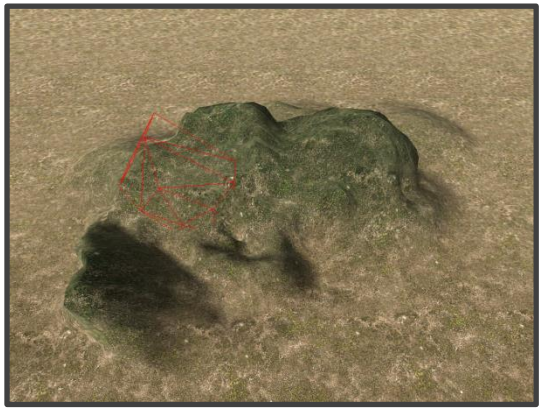
MFC – 맵 툴

- 터레인 높낮이 조절



- 지형의 높낮이와 UV를 조절 할 수 있음.
- 스플레팅 텍스쳐의 RGB 색을 칠하고 저장 할 수 있음.

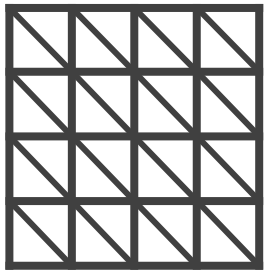
< Before >



< After >

터레인

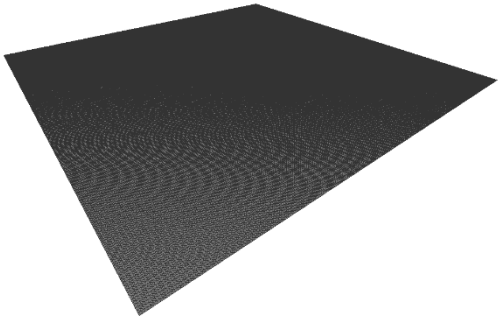
1. 버퍼 생성



< Terrain_Buffer >

```
m_fInterval = _fInterval;

m_iWidth = _iWidth_Num;
m_iHeight = _iHeightNum;
m_iVerticesNum = m_iWidth * m_iHeight;

m_dwVtxFVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1;
```

→ 가로, 세로, 넓이 정보 갱신



< 터레인 버퍼를 생성 >

```
for (size_t iHeightIndex = 0; iHeightIndex < m_iHeight; ++iHeightIndex)
{
    for (size_t iWidthIndex = 0; iWidthIndex < m_iWidth; ++iWidthIndex)
    {
        iIndex = (m_iWidth * iHeightIndex) + iWidthIndex;

        vPos = { _float(iWidthIndex * m_fInterval) , ZERO , _float(iHeightIndex * m_fInterval) };
```

→ 입력된 가로, 세로 수에 따라서 정점 위치 정의

··· 이하 생략 ···

- 먼저 터레인 가로, 세로, 넓이를 가지고 크기와 간격을 조절해 버퍼를 생성

터레인

2. 높이 맵 적용

Import_HeightMap

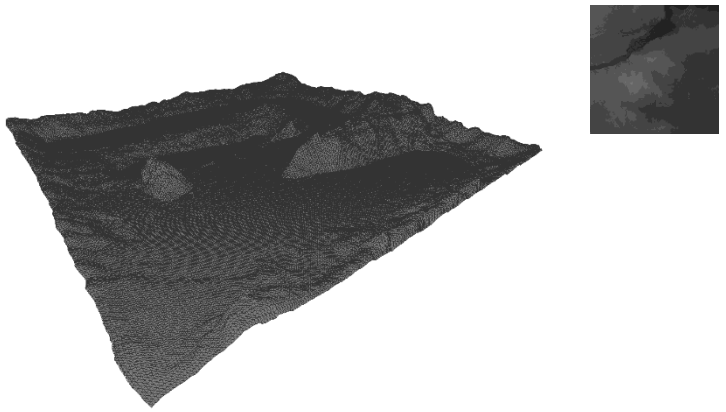
```
_ulong* pPixel = new _ulong[ih.biWidth * ih.biHeight];

ReadFile(hFile, pPixel, sizeof(_ulong) * (ih.biWidth * ih.biHeight), &dwByte, nullptr);
```

→ 파일 입출력으로 높이 맵 BMP파일 컬러 값 가져옴

Define_Terrain_Position.Y

```
vPosition = {
        _float(iHoriIndex * m_fInterval),
        (_float(pPixel[iIndex] & 0x000000ff) / LANCER_STAGE_HEIGHTMAP_DIVIDE),
        _float(iVertiIndex* m_fInterval)
        };
```

→ 하나의 색상 값으로 높이 적용

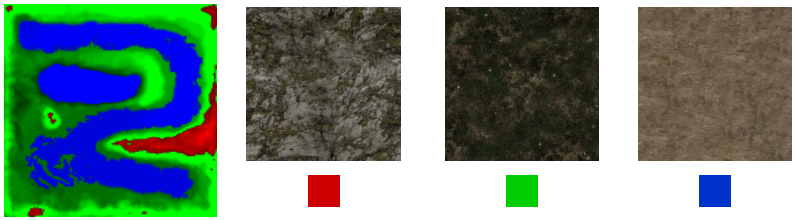


< 높이 맵 적용 및 툴에서 높이 수정 >

- 생성된 버퍼에 높이 맵으로 기본 높이를 잡고 MFC 지형 툴로 수정함.

터레인

3. 텍스처 스플래팅



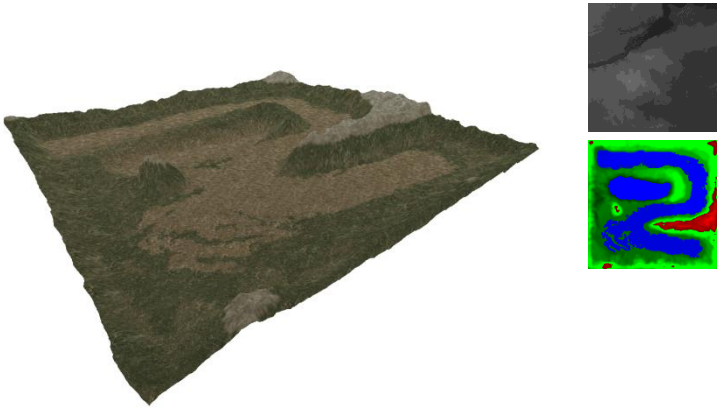
HLSL_Pixel_Shader

```
vector vTex[TEXTURE_END];
vTex[TEXTURE_R] = tex2D(FirSampler, In.vUV * g_fUvMul);
vTex[TEXTURE_G] = tex2D(SecSampler, In.vUV * g_fUvMul);
vTex[TEXTURE_B] = tex2D(ThirdSampler, In.vUV * g_fUvMul);
vTex[TEXTURE_FILTER] = tex2D(FilterSampler, In.vUV);

vector vR = (vector)vTex[TEXTURE_FILTER].r;
vector vG = (vector)vTex[TEXTURE_FILTER].g;

float4 vDiffuseMap = (vTex[TEXTURE_R] * vR);
float4 vDiffuseMap_GB = ((vTex[TEXTURE_G] * (MINUS_ONE - vR)) * vG) +
                        ((vTex[TEXTURE_B] * (MINUS_ONE - vR)) * (MINUS_ONE - vG));

vDiffuseMap += vDiffuseMap_GB;
```

3장의 텍스처

스플래팅 텍스처

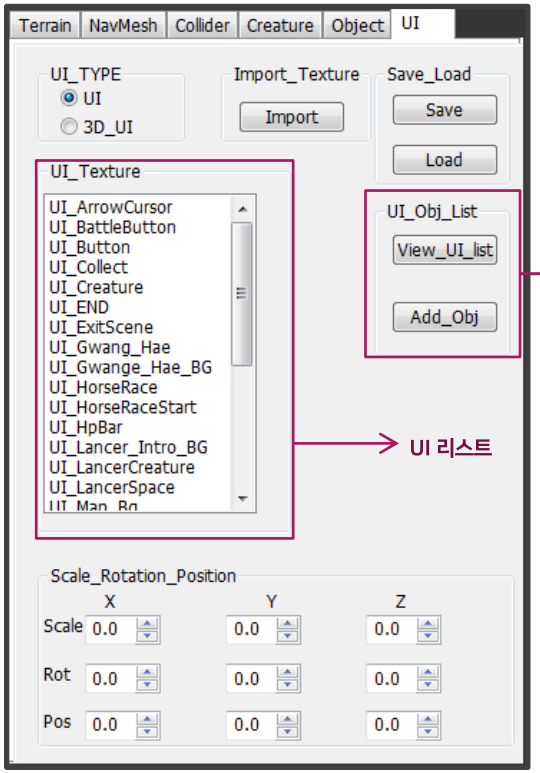
RGB값에 따라 텍스처 나눠서 적용

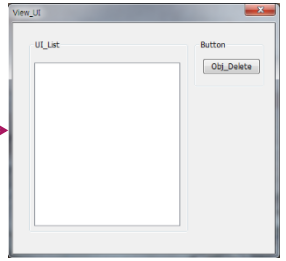


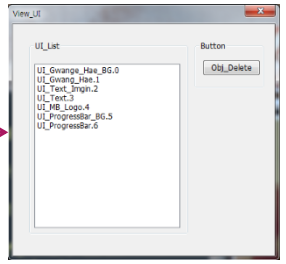
< 텍스쳐를 RGB에 따라 적용 >

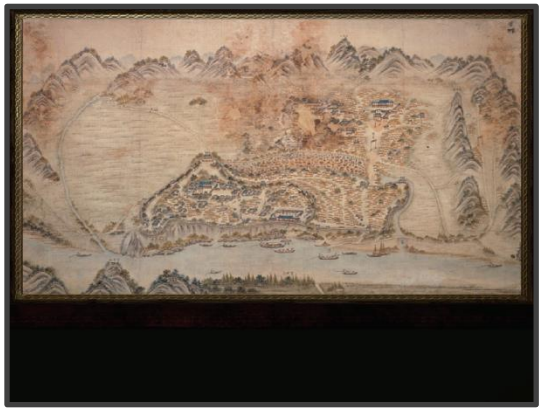
- 스플래팅 텍스쳐는 1차적으로 높이에 따라서 나눠지고 툴에서 RGB값을 색칠해 텍스처로 저장 후 클라이언트에 적용.

MFC – UI







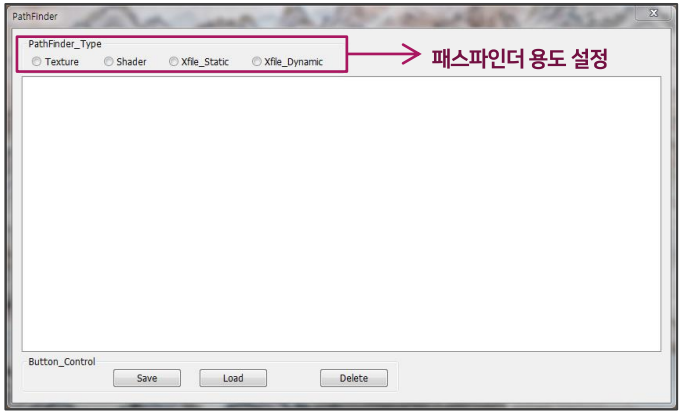


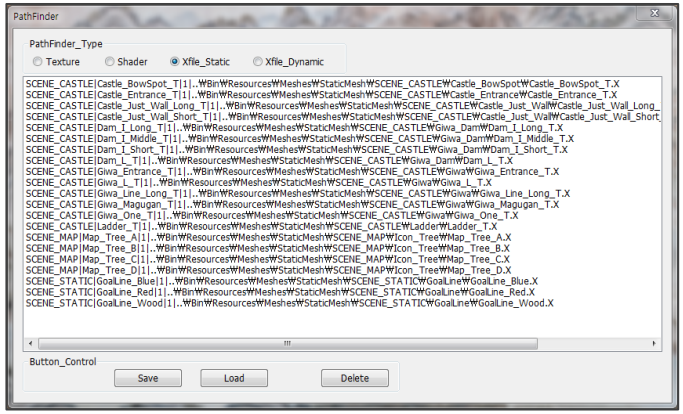
< Before >

< After >

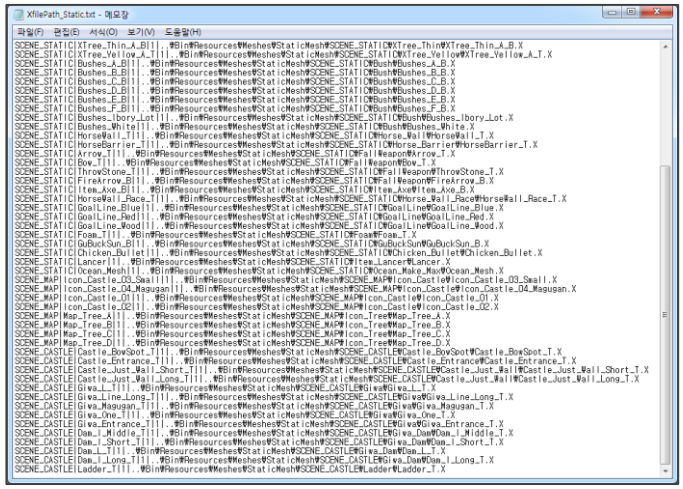
- UI의 크기, 회전, 위치를 설정하고 저장 함.

MFC – PathFinder





메모장 파일로 저장

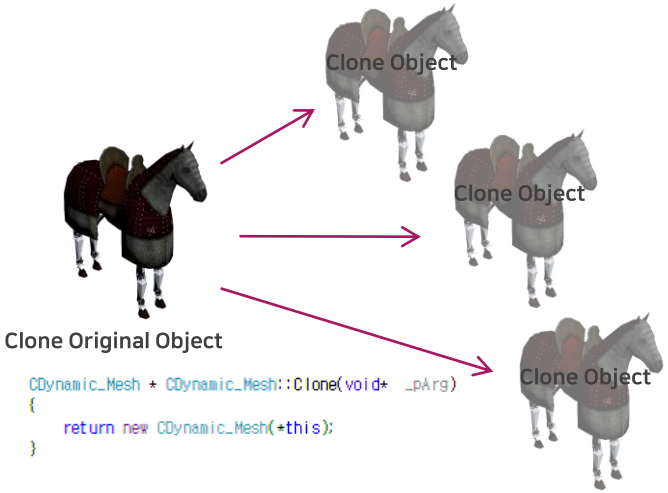


- 패스파인더 툴은 텍스처, 메쉬, 쉐이더 파일의 경로들을 추가하고 메모장으로 저장해 클라이언트에서 쉽게 파일들을 관리하도록 도와주는 툴임.

디자인 패턴 - 프로토타입

Create Original Object

```
if (FAILED(CComponentMgr::GetInstance()->Add_Component_Prototype(_dwSceneId, szStateKey
    , CDynamic_Mesh::Create(_pGpDev, szImgPath))))
{
    MessageBox(0, szImgPath, L"Load_CDynamic_Mesh", MB_OK);
    return E_FAIL;
}
```



Clone Original Object

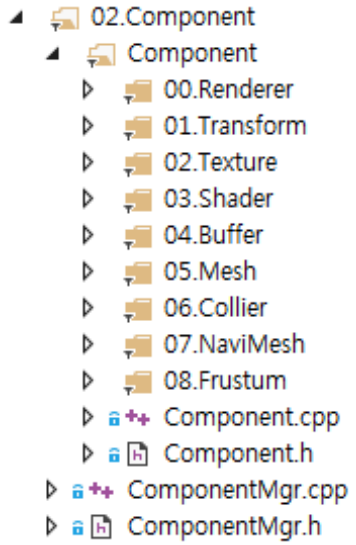
```
CDynamic_Mesh * CDynamic_Mesh::Clone(void*  _pArg)
{
    return new CDynamic_Mesh(*this);
}
```

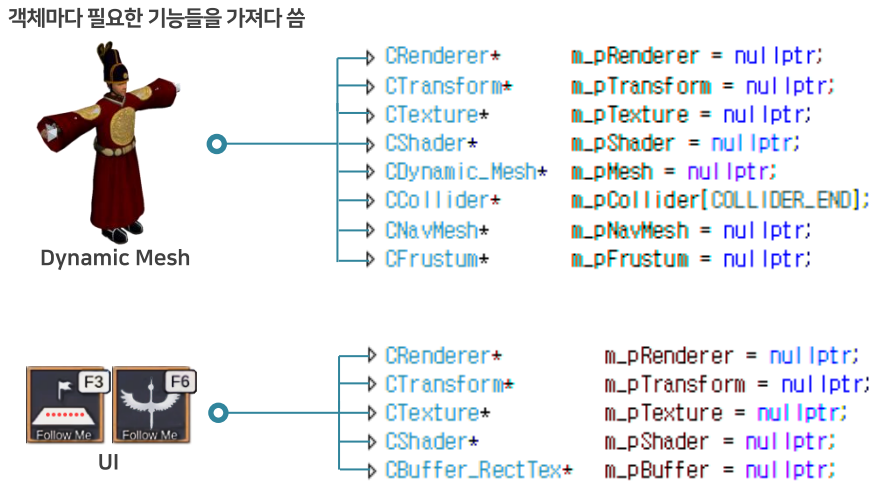
- **프로토타입 패턴을 쓰면서 좋았던 점**

1. 메쉬를 부를 때 객체마다 파일 입출력 시 프레임 드랍이 발생하였는데 원본 객체만 파일 입출력하고 나머지는 복제하는 식으로 하면서 가벼워짐.
2. 원본 객체를 맵 컨테이너에서 관리하고 있기 때문에 Run-Time에도 쉽게 접근해서 복제하기 용이함.

< 다수의 말 복제 >

디자인 패턴 - 컴포넌트





- **컴포넌트 패턴을 쓰면서 좋았던 점**

1. **기능을 한번 만들어 놓으면 가져다 쓰기만해서 좋음.**
2. **하나의 기능을 서로 공유하고 있기 때문에 수정이 쉬움.**
3. **기능별로 클래스를 따로 만들기 때문에 코드 비대화를 막음.**

감사합니다.

010 – 3296 - 0692

simonmatthew@naver.com