3D게임 프로그래밍 과제

게임공학과 2014182049

김승범

과제의 프레임워크는 교수님의 LabProject(Scene)을 기반으로 한다. 프레임워크는 내가 필요한 부분만을 남기고 삭제하여 코드가 깔끔하게 보이도록 하였다는 것을 미리 언급한다. 들어가기에 앞서서 나의 롤러코스터 과제 작동 방식에 대해서 설명하겠다. 플레이어는 키보드로 조작이 불가능하고 사용자는 오직 레일이 깔리는 경로만 조작이 가능하다. 조작은 마우스로 하며, 마우스 왼쪽버튼을 클릭한 상태에서 마우스를 위아래로 움직이면 레일이 위아래 방향으로, 왼쪽 오른쪽으로 움직이면 왼쪽 오른쪽 방향으로 깔린다. 그리고 마우스 오른쪽 버튼을 클릭한 상태에서는 위아래는 왼쪽버튼과 동일하고 왼쪽 오른쪽으로 움직였을 때 꽈배기 모양으로 회전하듯이 레일이 깔리게 된다. 레일이 깔리면 플레이어는 그 레일을 따라 나아가며 레일의 방향에 맞게 회전한다.

우선 레일이 필요하기 때문에 레일을 먼저 만들었다. 레일의 모양을 먼저 설명하자면, 레일을 타기 위해 가장자리에 평면모양의 메쉬가 깔리고 레일의 벽 역할을 하는 정사각형 메쉬 두개가 평면모양을 기준으로 양 옆으로 위치한 모양이다. 레일을 객체화 시키고 싶어 CRail이라는 클래스를 만들어 주었다.

class CRail

{

//생성자 소멸자 생략

public:

void SetRotation(const float& pitch, const float& yaw, const float& roll); // 레일을 회전 시킨다

void SetPosition(XMFLOAT3& xmf3Position); // 레일의 위치를 지정한다

XMFLOAT3& GetPosition(); // 레일의 위치를 가져온다

XMFLOAT3& GetLookDir(); // 레일이 바라보는 방향을 가져온다

const float& GetPitch() { return m\_fPitch; } // 레일의 Pitch를 가져온다

const float& GetYaw() { return m\_fYaw; } // 레일의 Yaw

const float& GetRoll() { return m\_fRoll; } // 레일의 Roll

public:

void InitRail(); // 레일 초기화

int Update(float fElapsedTime); // 레일 업데이트

void Render(HDC hDCFrameBuffer, class CCamera \*pCamera); // 레일 화면에 출력

void Release(); // 레일 해제

private:

class CGameObject \*m\_pBoxLeft = nullptr; // 레일 왼쪽 벽

class CGameObject \*m\_pBoxRight = nullptr; // 레일 오른쪽 벽

class CGameObject \*m\_pPlane = nullptr; // 레일 평면

float m\_fLifeTime = 0.f; // 생존 시간

float m\_fPitch = 0.f;

float m\_fYaw = 0.f;

float m\_fRoll = 0.f;

};

멤버변수는 양 옆에 정사각형 벽과 타는 평면을 동적할당으로 만들어 주기 위해 CGameObject\* 타입 세개의 변수와 레일이 회전할 때 적용된 Pitch, Yaw 그리고 Roll 세개의 변수와 일정 시간이 지나면 레일이 사라지게 하기 위해 LifeTime을 알 수 있는 변수 한 개로 구성되어 있다. 벽과 레일 평면을 서로 하나가 되어 움직여야 하기 때문에 벽이 레일 평면의 WorldMatrix를 ParentMatrix로 가질 수 있도록 하고 싶었다. 그래서 CGameObject에 XMFLOAT4X4타입 m\_xmf4x4Parent 멤버변수를 추가하였고, SetParentMatrix 함수를 추가하여 ParentMatrix의 값을 정해 줄 수 있도록 하였다.

void CGameObject::Render(HDC hDCFrameBuffer, CCamera \*pCamera)

{

XMFLOAT4X4 xmfTotalWorldMatrix = Matrix4x4::Multiply(m\_xmf4x4World, m\_xmf4x4Parent);

if (m\_pMesh)

{

HPEN hPen = ::CreatePen(PS\_SOLID, 0, m\_dwColor);

HPEN hOldPen = (HPEN)::SelectObject(hDCFrameBuffer, hPen);

m\_pMesh->Render(hDCFrameBuffer, xmfTotalWorldMatrix, pCamera);

::SelectObject(hDCFrameBuffer, hOldPen);

::DeleteObject(hPen);

}

}

그러면 기존 m\_xmf4x4World에 m\_xmf4x4Parent를 곱하여 xmfTotalWorldMatrix를 만들어 주고 이 행렬 정보를 바탕으로 메쉬를 그려줄 수 있도록 하였다. 이렇게 하면 부모 오브젝트의 WorldMatrix를 자식들이 ParentMatrix로 가지도록 하여 마치 부모의 한 부분이 된 듯 부모를 따라 같이 움직이도록 하였다. 다시 돌아와 CRail 클래스 멤버함수에 대해서 자세하게 설명하겠다.

void CRail::InitRail()

{

CCubeMesh \*pObjectCubeMesh = new CCubeMesh(2.f, 2.f, 2.f);

CPlaneMesh \*pObjectPlaneMesh = new CPlaneMesh(8.f, 5.f);

if (!m\_pPlane)

{

m\_pPlane = new CGameObject();

m\_pPlane->SetMesh(pObjectPlaneMesh);

m\_pPlane->SetColor(RGB(0, 0, 0));

m\_pPlane->SetPosition(0.f, -1.f, 0.f);

}

if (!m\_pBoxLeft)

{

// 위와 동일 하므로 부분 생략

m\_pPlane->SetMesh(pObjectCubeMesh);

m\_pBoxLeft->SetPosition(-5.f, 1.f, 0.f);

}

if (!m\_pBoxRight)

{

// 위와 동일 하므로 부분 생략

m\_pBoxRight->SetPosition(+5.f, 1.f, 0.f);

}

}

우선 CRail::InitRail()은 Rail의 모양을 위해 선언한 3개의 CGameObject\* 멤버변수를 동적할당 하여 메쉬, 컬러, 위치를 Set해준다. 평면을 표현할 메쉬가 필요하여 CMesh를 상속받는 CPlaneMesh 자식 클래스를 만들었다. 클래스 생성시 Width값과 Depth값을 넘겨주면 그에 맞는 정점들을 Set하여 평면 메쉬를 만든다. 레일 평면의 메쉬는 이 CPlaneMesh 클래스를 이용하여 만들어 주었다. 양쪽 벽 메쉬는 기존에 있는 CCubeMesh 클래스를 사용하였다.

int CRail::Update(float fElapsedTime)

{

m\_fLifeTime += fElapsedTime;

if (m\_fLifeTime >= 5.f)

{

m\_fLifeTime = 0.f;

return -1;

}

if (m\_pPlane)

{

if (m\_pBoxLeft)

m\_pBoxLeft->SetParentMatrix(m\_pPlane->GetWorldMatrix());

if (m\_pBoxRight)

m\_pBoxRight->SetParentMatrix(m\_pPlane->GetWorldMatrix());

}

return 0;

}

그리고 CRail::Update(float fElapsedTime)은 m\_fLifeTime을 fElapsedTime와 축약연산으로 더하여 시간을 축적 시키고 약 5초가 지나면 -1을 반환하여 외부에서 레일을 삭제 할 수 있도록 만들었다. 그리고 이 함수에서 양 옆 벽이 레일 평면의 WorldMatrix를 ParentMatrix로 가지도록 하여 위에서 설명했던 의도대로 작동되도록 하였다. CRail::Render(HDC hDCFrameBuffer, CCamera \* pCamera)는 세개의 오브젝트를 Render하는 함수이고, CRail::Release()는 동적할당 해제하는 함수이다. CRail의 모든 움직임은 m\_pPlane에서 하며 나머지는 부모로 Set하여 그것을 따라간다.

레일에 대한 객체가 준비되어 있으므로 이제 레일을 마우스 입력에 맞게 깔 수 있도록 하여야 한다. 우선 CPlayerScene 클래스를 살펴보도록 하겠다.

class CPlayerScene : public CScene

{

//생성자 소멸자 생략

public:

void SetPitch(const float& pitch) { m\_fPitch = pitch;} // 마우스 움직임에 맞게 Pitch값을 지정

void SetYaw(const float& yaw) { m\_fYaw = yaw; } // 마우스 움직임에 맞게 Yaw값을 지정

void SetRoll(const float& roll) { m\_fRoll = roll; } // 마우스 움직임에 맞게 Roll값을 지정

private:

// 설명하지 않은 멤버변수들은 뒤에서 설명한다.

vector<CRail\*> m\_vecRails;

int m\_iIndex = -1;

float m\_fAmount = 0.f;

float m\_fSpawnTime = 0.f;

float m\_fPitch = 0.f;

float m\_fYaw = 0.f;

float m\_fRoll = 0.f;

XMFLOAT3 m\_xmf3OldPos = { 0.f, 0.f, 0.f };

bool m\_isStartCheck = true;

XMFLOAT3 m\_xmf3PlayerStartPos = { 0.f, 0.f, 0.f };

public:

virtual void Update(float fElapsedTime); // Scene 업데이트

virtual void Render(HDC hDCFrameBuffer, CCamera \*pCamera); // Scene 출력

public:

void RidePlayerRail(float fElapsedTime); // 플레이어가 레일을 탈 수 있도록 해준다.

void BuildRail(); // 레일을 만든다.

};

레일을 한곳에 보관하기 용이하도록 하기 위해서 STL vector 컨테이너를 사용하였다. 멤버변수Vector<CRail\*> m\_vecRails가 바로 그것이다.

void CPlayerScene::Update(float fElapsedTime)

{

m\_fSpawnTime += fElapsedTime;

if (m\_fSpawnTime >= 0.1f)

{

BuildRail();

m\_fSpawnTime = 0.f;

}

// 뒤에서 설명하기 위해 생략

}

CPlayerScene::Update(float fElapsedTime)에서 시간 간격으로 레일 생성을 위해 필요한 멤버 변수 m\_fSpawnTime을 fElapsedTime와 축약연산으로 더하여 시간을 축적 시키고 약 0.1초가 지나면 CPlayerScene::BuildRail()함수를 이용해 레일이 생성되도록 하였다. m\_fSpawn값은 다시 사용하기 위해 0.f로 값을 바꿔주었다. 이렇게 하면 0.1초마다 레일이 생성된다.

CPlayerScene::BuildRail() 함수를 설명하기에 앞서서 마우스 움직임에 따른 레일의 회전을 위해 필요한 함수인 CGameFramework::ProcessInput()와 CGameFrameword::OnProcessingMouseMessage(생략)을 설명하겠다.

void CGameFramework::OnProcessingMouseMessage(HWND hWnd, UINT nMessageID, WPARAM wParam, LPARAM lParam)

{

switch (nMessageID)

{

case WM\_LBUTTONDOWN:

case WM\_RBUTTONDOWN:

::SetCapture(hWnd);

::GetCursorPos(&m\_ptOldCursorPos);

break;

case WM\_LBUTTONUP:

case WM\_RBUTTONUP:

::ReleaseCapture();

break;

case WM\_MOUSEMOVE:

break;

default:

break;

}

}

CGameFrameword::OnProcessingMouseMessage(생략)함수를 통해 마우스의 입력시 그 입력값에 맞는 동작을 수행하는데, 왼쪽과 오른쪽 마우스 버튼이 눌린 경우 ::SetCapure(hWnd)를 하여 프로그램 핸들 영역 외에서도 마우스의 값을 받아 올 수 있도록 하였고 ::GetCursorPos(&m\_ptOldCursorPos)로 마우스 좌표를 m\_ptOldCursorPos 멤버변수에 Set해주었다.

void CGameFramework::ProcessInput()

{

// 키보드 입력 부분 생략

float cxDelta = 0.0f, cyDelta = 0.0f;

POINT ptCursorPos;

if (GetCapture() == m\_hWnd)

{

SetCursor(NULL);

GetCursorPos(&ptCursorPos);

cxDelta = (float)(ptCursorPos.x - m\_ptOldCursorPos.x) / 3.0f;

cyDelta = (float)(ptCursorPos.y - m\_ptOldCursorPos.y) / 3.0f;

SetCursorPos(m\_ptOldCursorPos.x, m\_ptOldCursorPos.y);

}

if ((dwDirection != 0) || (cxDelta != 0.0f) || (cyDelta != 0.0f))

{

if (cxDelta || cyDelta)

{

if (pKeyBuffer[VK\_RBUTTON] & 0xF0)

{

dynamic\_cast<CPlayerScene\*>(m\_pScene)->SetRoll(-cxDelta \* 2.f);

dynamic\_cast<CPlayerScene\*>(m\_pScene)->SetPitch(cyDelta \* 2.f);

}

else

{

dynamic\_cast<CPlayerScene\*>(m\_pScene)->SetPitch(cyDelta \* 2.f);

dynamic\_cast<CPlayerScene\*>(m\_pScene)->SetYaw(cxDelta \* 2.f);

}

}

if (dwDirection) m\_pPlayer->Move(dwDirection, 0.55f);

}

m\_pPlayer->Update(m\_GameTimer.GetTimeElapsed());

}

GetCapture()로 현재 마우스가 캡쳐 하고 있는 윈도우 핸들이 프로그램 핸들인지 확인한다. 프로그램 핸들이 맞을 경우 GetCursorPos를 이용하여 현재 마우스 위치를 ptCursorPos에 넣어주고 왼쪽이나 오른쪽 마우스 버튼이 눌렸을 때의 처음 위치 m\_ptOldCursorPos와 뺄셈을 하여 x축과 y축으로 얼마만큼 마우스를 움직여 주었는가에 대한 변수 cxDelta와 cyDelta에 값을 집어넣어준다. 그리고 마우스 위치를 다시 m\_ptOldCursorPos로 Set해준다. 만약 움직임이 있었다면 마우스 왼쪽버튼인지 오른쪽 버튼인지 검사하고 CScene에서 CPlayerScene으로 다운 캐스팅하여 CPlayerScene에서 레일의 회전 값을 정해줄 m\_fPitch, m\_fYaw, m\_fRoll에 cxDelta와 cyDelta값을 마우스 입력 값에 맞게 Set해준다.

void CPlayerScene::BuildRail()

{

CRail\* pRail = new CRail;

pRail->InitRail();

pRail->SetRotation(m\_fPitch, m\_fYaw, m\_fRoll);

XMFLOAT3 xmf3RailLookDir = pRail->GetLookDir();

XMFLOAT3 xmf3MoveDistance = Vector3::ScalarProduct(xmf3RailLookDir, 5.f, false);

XMFLOAT3 xmf3NewPosition = Vector3::Add(m\_xmf3OldPos, xmf3MoveDistance);

pRail->SetPosition(xmf3NewPosition);

m\_vecRails.emplace\_back(pRail);

m\_xmf3OldPos = xmf3NewPosition;

// 추후 설명을 위해 생략

}

레일을 만들기 위한 준비는 끝이 났으니 이제 CPlayerScene::BuildRail()를 설명하겠다. 앞서 설명한 CRail 객체를 동적할당을 하고 InitRail()을 해준다. 그리고 앞에서 받아온 Pitch, Yaw, Roll 정보를 이용하여 레일을 회전시킨다. 레일은 한곳에만 깔리는게 아니라 앞으로 나아가면서 깔려야 하므로, 이제 GetLookDir()함수를 이용하여 회전한 상태에서의 Z축 Look 벡터를 받아온다. GetLookDir()은 방향 값을 반환해주는 함수이므로 벡터 Nomalize를 한 상태로 반환된다. 받아온 Look 벡터에 5.f를 스칼라곱 한 값을 이전 레일의 위치를 합해주고 그 값을 SetPosition()을 해준다면 이전 레일에서 회전한 방향으로 5.f거리 만큼 이동하게 된다.

이제 사용자가 마우스 입력에 따라 레일이 그것에 맞게 깔리게 된다. 이제 플레이어를 그 레일에 맞게 태울 것이다. 여기서 많은 고민을 했고 여러 시도를 했는데 우선 첫번째로 레일에 안보이는 Bounding Box를 깔아 플레이어와 레일이 최초로 충돌했을 경우 그 레일에 맞는 Pitch, Yaw, Roll값을 플레이어에 대입하고 플레이어는 그저 자신의 Look 방향으로만 가면 알아서 레일에 맞게 회전하기 때문에 레일에 탈수 있도록 하는 방법이다. 이 방법은 XNA Math에 있는 OOBB Contains()함수를 이용하여 플레이어의 위치 벡터가 레일의 OOBB 안에 포함되는 경우를 검사하여 그 레일에 맞게 회전시키도록 설계를 했지만, 충돌처리가 정확하지 않고 XNA Math 충돌처리 부분 학습 미흡으로 구현에 실패하였다. 두번째 방법은 레일을 담고있는 vector를 index 접근하여 플레이어가 있는 레일 위치에서 그 다음 레일의 위치 만큼 보간하여 자연스럽게 레일들의 위치를 이어가도록 하고 회전 값 또한 보간을 통해 회전이 가능하도록 하여 플레이어가 레일을 타는 방법이다. 이 방법은 보간 하여 위치를 이어가는 것에는 성공하였지만 회전값을 그렇게 하지 못하여서 다른 방법을 이용하여 레일을 태우는 데에 성공했다. 그래서 나는 두번째 방법을 자세히 설명하고자 한다.

우선 플레이어를 레일에 태우는 CPlayerScene::RidePlayerRial(float fElapsedTime)을 설명하기에 앞서서 사용하는 멤버변수들과 인라인 보간 함수를 설명하겠다. 우선 인덱스 접근을 위한 m\_iIndex 멤버변수가 있다. 이 변수는 vector가 비어 있을 경우 인덱스 접근을 하면 안되기 때문에 -1로 초기화 한다. 그리고 Rail을 태우는 작업을 시작할 수 있도록 도와주는 m\_isStartCheck, 그리고 플레이어가 레일에서 다음레일로 보간 하여 갈 때 필요한 한 레일에서의 출발 위치벡터 m\_xmf3PlayerStartPos와 보간 함수에 필요한 amount값을 위해 m\_fAmount를 멤버변수로 선언하였다.

namespace Lerp

{

inline float lerp(float fSrc, float fDst, float fAmount)

{

return float(fSrc + ((float)fDst - fSrc) \* fAmount);

}

}

그리고 인라인 함수를 만들어 사용자가 원하는 보간 값을 반환하여 보간 하는데에 있어서 편의성을 제공하였다. fSrc에서 fDst로 fAmount 만큼 보간 하여 가는데 fAmount의 범위는 0.f ~ 1.f이고 0.f로 갈수록 fSrc에 1.f로 갈수록 fDst에 가까워진다.

void CPlayerScene::BuildRail()

{

//앞부분은 이미 설명 했으므로 생략

if (m\_isStartCheck)

{

m\_iIndex = 0;

m\_pPlayer->SetPosition(xmf3NewPosition.x, xmf3NewPosition.y, xmf3NewPosition.z);

m\_xmf3PlayerStartPos = xmf3NewPosition;

m\_pPlayer->Rotate(m\_fPitch, m\_fYaw, m\_fRoll);

m\_isStartCheck = false;

}

}

CPlayerScene이 Update될 때 최초로 실행되는 조건문이다. 앞부분에서 최초로 레일을 컨테이너에 집어넣은 후 플레이어가 레일을 탈 수 있도록 m\_iIndex값을 -1에서 0으로 바꾸어 주고 플레이어의 위치와 회전 을 첫 레일의 위치와 회전값으로 Set하고 xmf3PlayerStartPos도 첫 레일 위치로 Set한다. m\_isStartCheck는 false가 되어 다음 BuildRail() 함수부터 조건문은 실행되지 않는다.

void CPlayerScene::RidePlayerRail(float fElapsedTime)

{

if (m\_vecRails.empty())

return;

if (-1 == m\_iIndex)

return;

if (m\_vecRails.size() == m\_iIndex)

{

m\_iIndex = 0;

return;

}

if (m\_fAmount > 1.f)

{

m\_xmf3PlayerStartPos = m\_vecRails[m\_iIndex]->GetPosition();

++m\_iIndex;

m\_fAmount = 0.f;

return;

}

m\_pPlayer->SetRollPitchYaw(m\_vecRails[m\_iIndex]->GetPitch(),

m\_vecRails[m\_iIndex]->GetRoll(),

-m\_vecRails[m\_iIndex]->GetYaw());

XMFLOAT3 xmfLerpedPos = XMFLOAT3(

Lerp::lerp(m\_xmf3PlayerStartPos.x, m\_vecRails[m\_iIndex]->GetPosition().x, m\_fAmount),

Lerp::lerp(m\_xmf3PlayerStartPos.y, m\_vecRails[m\_iIndex]->GetPosition().y, m\_fAmount),

Lerp::lerp(m\_xmf3PlayerStartPos.z, m\_vecRails[m\_iIndex]->GetPosition().z, m\_fAmount));

m\_pPlayer->SetPosition(xmfLerpedPos.x, xmfLerpedPos.y, xmfLerpedPos.z);

m\_fAmount += fElapsedTime \* 10.f;

}

우선 첫번째, 두번째, 세번째 조건문은 예외처리이다. 첫번째는 CRail을 담는 컨테이너가 비어 있을 경우, 두번째는 컨테이너를 접근할 인덱스 값이 초기값일 경우, 그리고 세번째는 인덱스 값이 컨테이너의 크기를 넘어가는 경우이다. 이런 경우들은 플레이어를 레일에 태울 수 없는 상황이기 때문에 함수를 빠져나온다. 우선 인덱스 접근을 이용하여 현재 플레이어가 지나가는 레일의 Pitch, Yaw, Roll을 받아와 플레이어에 SetRollPitchYaw 함수에 매개변수로 넣어주면 플레이어는 레일의 회전 값에 맞게 회전하게 된다. 그리고 현재 레일의 포지션과 플레이어의 시작위치를 앞에서 설명한 Lerp::lerp 함수를 이용하여 m\_fAmount만큼 보간된 x, y, z값을 플레이어에게 SetPosition 해주었고. m\_fAmount 값은 fElapsedTime \* 10.f만큼 축적된다. m\_fAmount의 값이 축적되다 1.f를 넘어가게 되면 m\_xmf3PlayerStartPos는 현재 레일의 위치가 되고 m\_iIndex값은 1만큼 증가하며 m\_fAmount는 다시 사용을 위해 0.f를 대입한다.

레일이 계속 생성된다면 분명 프레임이 떨어질 것이다. 그래서 이미 지나간 레일은 지워줘야 한다.

void CPlayerScene::Update(float fElapsedTime)

{

// 앞에서 설명 했으므로 생략

vector<CRail\*>::iterator iter;

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

{

if (-1 == (\*iter)->Update(fElapsedTime))

{

(\*iter)->Release();

iter = m\_vecRails.erase(iter);

--m\_iIndex;

}

else

++iter;

}

RidePlayerRail(fElapsedTime);

}

앞서 설명한 CRail::Update(float fElapsedTime)에서 -1을 반환하면 그 레일은 시간이 지나 지워야 할 레일 이므로 iterator를 사용하여 반복문을 통해 조건 성립 시 CRail::Release()를 통해 CRail에 있는 동적할당 멤버변수들을 할당 해제 시켜주고 vector 컨테이너에서 그 객체를 지운다. 또한 size가 하나 줄었으므로 m\_iIndex도 하나 감소시킨다.

이제 플레이어가 마우스 입력에 따라 생성된 레일을 타고 이동할 수 있게 되었다. 일반적인 방법은 앞서 설명한 첫번째 방법일 것이다. 편법으로 한 것 같은 느낌이라 찝찝한 감이 없지 않아 있지만 충돌처리가 부족하여 학습이 더 필요하다는 것을 깨닫게 되었고 교수님 프로젝트의 로직과 구현된 기능들을 좀 더 꼼꼼히 볼 수 있는 기회가 되어 뜻 깊었다.

* 결과화면

 