DirectX 11 3D Team Project

Black Death

싱글 FPS PC 게임

CONTENTS

Introduce

- 목적 : 3D FPS 게임 개발
- 개발 기간 : 60일
- 사용 도구 : C++, Direct X 11
- 개발 인원 : 5명(클라이언트 4명, 기획 1명)
- 담당 업무 : 이펙트 툴을 활용한 **이펙트 제작** 및 **적용**
  3D 충돌 처리 구현
  스킬 구현
  최적화 및 **게임 시스템 개발**

Game

https://youtu.be/LwX0ViK7Yzs

Effect

이펙트

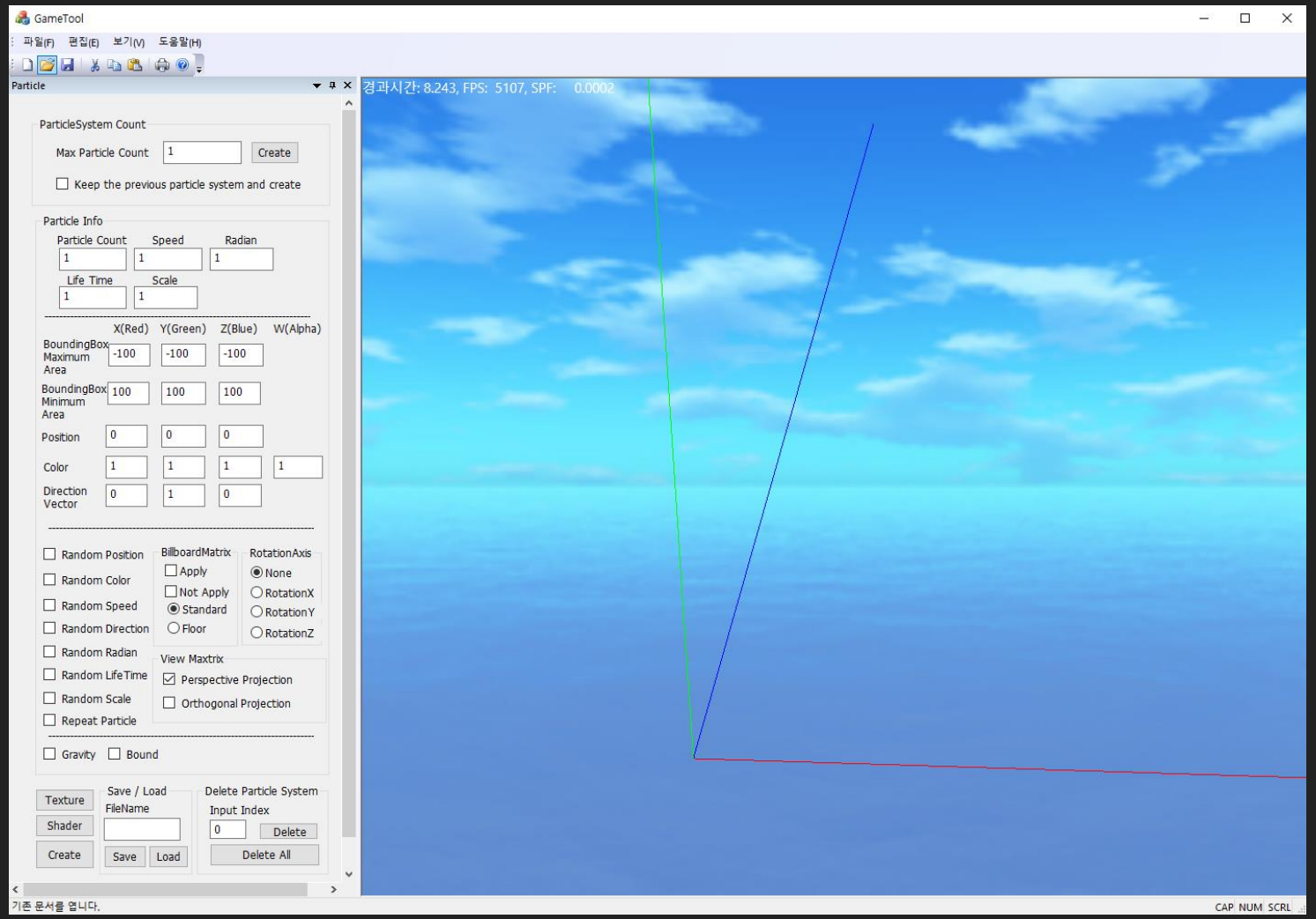
Effect - 요약

기획된 **이펙트(20가지)**를 **이펙트 툴**을 사용하여 이펙트 제작

엔진과 호환 가능한 **엔진 버전 이펙트 파일 생성**

각 파트(맵, 캐릭터) 오브젝트에 **파티클 매니저**를 활용하여 연동

Effect - Tool



[그림 1-1] 이펙트 툴

- MFC와 C++로 개발된 이펙트 툴 프로그램을 통해 **이펙트 리소스 제작**
- 툴의 각 속성값을 통해 **이펙트의 다양한 움직임**이 가능하고, **애니메이션 및 멀티 텍스처**를 통해 다양하게 표현할 수 있음
- 엔진에서 사용할 **이펙트 리소스 파일** 생성

Effect – Particle System Manger

```cpp
INT ParticleSystemManager::Add(ParticleSystem* newParticle)
{
    INT iCnt = 0;
    for (m_ParticleNameItor = m_ParticleNameList.begin();
        m_ParticleNameItor != m_ParticleNameList.end();
        m_ParticleNameItor++, iCnt++)
    {
        TCHAR* pName = (TCHAR*)(*m_ParticleNameItor);
        if (!_tcsicmp(pName, newParticle->m_szParticleName))
        {
            return iCnt;
        }
    }

    ParticleSystem *pPoint = NULL;
    S_NEW(pPoint, ParticleSystem);
    assert(pPoint);
    *pPoint = *newParticle;
    m_ParticleList.insert(make_pair(++m_iCurIndex, pPoint));
    m_ParticleNameList.push_back(newParticle->m_szParticleName);

    return m_iCurIndex;
}

ParticleSystem* ParticleSystemManager::GetPtr(INT iIndex)
{
    m_Itor = m_ParticleList.find(iIndex);
    if (m_Itor == m_ParticleList.end())
        return NULL;
    ParticleSystem* pPoint = (*m_Itor).second;
    return pPoint;
}
```

[그림 1-2] 파티클 삽입 및 객체 반환

```cpp
namespace PARTICLE
{
    #define WalkingDust          0
    #define EnemyBlood1          1
    #define EnemyBlood2          2
    #define HeroDamaged          3
    #define GrenadeEffect        4
    #define RifleShotEffect      5
    #define PistolShotEffect     6
    #define SnowDrop1            7
    #define SnowDrop2            8
    #define SnowDrop3            9
    #define SnowDrop4            10
    #define CarFire              11
    #define StageBoom            12
    #define ItemEffect           13
    #define HealingEffect        14
    #define HealingBackground    15
    #define BloodFloor1          16
    #define BloodFloor2          17
    #define BloodFloor3          18
    #define HeroElectroDamaged   19
};
```

[그림 1-3] 이펙트 리스트

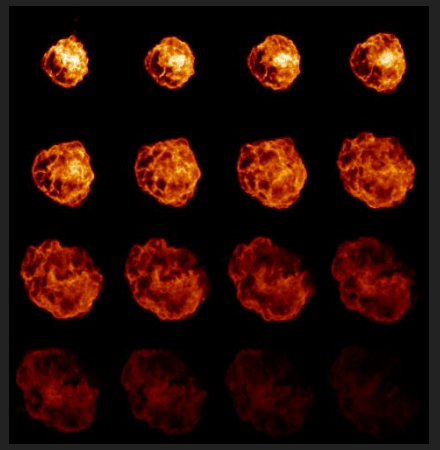
- Particle system Manager class에 **Singleton 패턴**을 적용하여 매니저 객체로 불러온 이펙트 관리
- 이펙트의 **중복 생성을 제한**하고 포인터를 통해 객체 반환
- 협업 과정에서 **매니저**와 **파티클(이펙트) 리스트**의 이펙트 이름을 인덱스로 하여 다른 작업자(맵, 캐릭터)가 쉽게 사용할 수 있도록 제작

Effect – Sprite

```
void Sprite::SetRect(int iRow, int iCol, float fOffset)
{
    m_fLifeTime = fOffset;
    m_fOffsetTime = fOffset / (iRow * iCol);
    RectUV rt;
    float fOffsetX = 1.0f / iRow;
    float fOffsetY = 1.0f / iCol;
    m_uvList.resize(iRow * iCol);
    int iIndex = 0;

    for (int iV = 0; iV < iRow; iV++)
    {
        for (int iU = 0; iU < iCol; iU++)
        {
            rt.uv[0].x = iU * fOffsetX;
            rt.uv[0].y = iV * fOffsetY;
            rt.uv[1].x = rt.uv[0].x + fOffsetX;
            rt.uv[1].y = rt.uv[0].y;
            rt.uv[2].x = rt.uv[1].x;
            rt.uv[2].y = rt.uv[1].y + fOffsetY;
            rt.uv[3].x = rt.uv[0].x;
            rt.uv[3].y = rt.uv[2].y;

            m_uvList[iIndex++] = rt;
        }
    }
}
```

[그림 1-4] 스프라이트 RECT 설정

- 이펙트 **애니메이션**은 **스프라이트** 사용하여 표현
- 4 by 4 스프라이트를 사용할 경우, 16개 이미지의 **UV 좌표를 연산**하여 UV 리스트에 저장



[그림 1-5] 4 by 4 스프라이트

Effect – Effect Parser

```
bool EffectParser::LoadEffect(TCHAR* szFileName, TCHAR* szParticleName)
{
    _tfopen_s(&m_pStream, szFileName, _T("rt"));
    if (m_pStream == NULL)
    {
        return false;
    }
    while (!feof(m_pStream))
    {
        TCHAR* szString[256];
        int iReadBoolValue;
        // 최대 파티클 개수
        _fgetts(m_pBuffer, 256, m_pStream);
        _stscanf_s(m_pBuffer, _T("%d"), &m_iNumParticleSys);

        for (int iEffect = 0; iEffect < m_iNumParticleSys; iEffect++)
        {
            // 파티클 생성
            ParticleSystem newParticle;

            // 파티클 이름 저장
            newParticle.m_szParticleName = szParticleName;
```

[그림 1-6] 이펙트 parser(1)

파티클 데이터

```
            // 이펙트 속도
            _fgetts(m_pBuffer, 256, m_pStream);
            _stscanf_s(m_pBuffer, _T("%f %f %f"),
                &newParticle.m_Particle[iObj].m_vSpeed.x,
                &newParticle.m_Particle[iObj].m_vSpeed.y,
                &newParticle.m_Particle[iObj].m_vSpeed.z);

            // 반지름
            _fgetts(m_pBuffer, 256, m_pStream);
            _stscanf_s(m_pBuffer, _T("%f"), &newParticle.m_Particle[iObj].m_fRadian);
        }
        // 파티클 시스템 매니저에 추가
        INT iIndex = I_ParticleSystem.Add(&newParticle);
        if (iIndex > m_iIndexParticleSys)
            m_iIndexParticleSys++;
    }
    _fgetts(m_pBuffer, 256, m_pStream);
}
fclose(m_pStream);

return true;
}
```

[그림 1-7] 이펙트 Parser(2)

이펙트 툴을 사용하여 제작된 엔진용 **이펙트 파일**을
게임 엔진의 이펙트 Parser를 이용하여 **불러오기**

Collision

충돌

Collision – 바운딩박스

```cpp
bool BBoundingBox::CreateBoundingBox(TVector3 vMax, TVector3 vMin, TVector3 vPos)
{
    m_BoundingBoxMax = vMax;
    m_BoundingBoxMin = vMin;

    // 구의 센터값
    m_vCenter   = vPos;
    m_vCenter.y = vPos.y + ((m_BoundingBoxMax.y + m_BoundingBoxMin.y) / 2.0f);

    // 구의 반지름
    TVector3 vRadius = (m_BoundingBoxMax + vPos) - m_vCenter;
    if (vRadius.x > vRadius.y)
    {
        if (vRadius.x > vRadius.z)
            m_fRadius = vRadius.x;
        else
            m_fRadius = vRadius.z;
    }
    else
    {
        if (vRadius.y > vRadius.z)
            m_fRadius = vRadius.y;
        else
            m_fRadius = vRadius.z;
    }

    m_fRadius = fabs(m_fRadius);

    m_matWorld._41 = vPos.x;
    m_matWorld._42 = vPos.y;
    m_matWorld._43 = vPos.z;
    return true;
}
```

[그림 2-1] 바운딩박스 생성

```cpp
void BBoundingBox::UpdateBoundingBox(TVector3 vPos)
{
    TMatrix matScale, matRotation;
    D3DXMatrixIdentity(&matScale);
    D3DXMatrixIdentity(&matRotation);
    D3DXMatrixRotationYawPitchRoll(&matRotation, g_pMainCamera->m_fYaw, 0.0f, 0.0f);
    D3DXMatrixMultiply(&m_matWorld, &matScale, &matRotation);

    m_vCenter = vPos;
    m_vCenter.y = vPos.y + ((m_BoundingBoxMax.y + m_BoundingBoxMin.y) / 2.0f);

    // 구의 반지름
    TVector3 vRadius = (m_BoundingBoxMax + vPos) - m_vCenter;
    if (vRadius.x > vRadius.y)
    {
        if (vRadius.x > vRadius.z)
            m_fRadius = vRadius.x;
        else
            m_fRadius = vRadius.z;
    }
    else
    {
        if (vRadius.y > vRadius.z)
            m_fRadius = vRadius.y;
        else
            m_fRadius = vRadius.z;
    }

    m_matWorld._41 = vPos.x;
    m_matWorld._42 = vPos.y;
    m_matWorld._43 = vPos.z;
}
```

[그림 2-2] 바운딩박스 갱신

- 오브젝트의 **바운딩박스**를 **구(Sphere)** 형태로 구성
- 연산한 **반지름 벡터의 축** 길이 중 **가장 긴 길이**를 기준으로 **반지름** 채택
- 매 프레임마다 **동적 오브젝트**의 위치 값이 변경될 때마다 **바운딩박스 동기화**
- 플레이어 캐릭터의 경우, **시점 변경**하면 캐릭터도 회전하므로 **카메라의 y 값**을 기준으로 **바운딩박스 회전**

Collision – 충돌 구현

```
class BCollision
{
public:
    //------------------------------------------------------------------------------
    // 구 vs 구 충돌
    //------------------------------------------------------------------------------
    BOOL CheckSphereInSphere(TVector3 vSphereCenter,
        TVector3 vOtherSphereCenter, float vSphereRadius, float vSphereOherRadius);
    //------------------------------------------------------------------------------
    // 구 vs 구 충돌시 충돌된 지점 찾아서 반대 방향 반환
    //------------------------------------------------------------------------------
    TVector3 SphereInSphereOppositeDir(TVector3 vFrontSphereCenter,
        TVector3 vBackSphereCenter, float vFrontSphereRadius, float vBackSphereRadius);
    //------------------------------------------------------------------------------
    // 벽면 미끌림
    //------------------------------------------------------------------------------
    bool SlipDetection(TVector3& vCurrentPos, TVector3& vBeforePos, float fSpeed);
public:
    BCollision();
    virtual ~BCollision();
};
```

[그림 2-3] 충돌 함수 정의

```
BOOL BCollision::CheckSphereInSphere(TVector3 vSphereCenter,
    TVector3 vOtherSphereCenter, float vSphereRadius, float vSphereOherRadius)
{
    float fDistance;
    TVector3 vDiff;

    vDiff = vSphereCenter - vOtherSphereCenter;
    fDistance = D3DXVec3Length(&vDiff);

    if (fDistance <= (vSphereRadius + vSphereOherRadius))
        return TRUE;
    return FALSE;
}

TVector3 BCollision::SphereInSphereOppositeDir(TVector3 vFrontSphereCenter,
    TVector3 vBackSphereCenter, float vFrontSphereRadius, float vBackSphereRadius)
{
    TVector3 vDir;
    vDir = (vFrontSphereCenter - vBackSphereCenter);
    float fDistance = (vFrontSphereRadius + vBackSphereRadius) - D3DXVec3Length(&vDir);
    D3DXVec3Normalize(&vDir, &vDir);
    vDir = vDir * fDistance;
    return vDir;
}

bool BCollision::SlipDetection(TVector3& vCurrentPos, TVector3& vBeforePos, float fSpeed)
{
    TVector3 vSlip = vBeforePos - vCurrentPos;
    vCurrentPos = vCurrentPos + (vSlip * fSpeed);
    return true;
}
```

[그림 2-4] 충돌 함수 구현

게임에서의 모든 충돌은 3가지 함수로 처리

1. 일반 오브젝트 충돌 처리
2. 충돌 후 오브젝트 밀림 처리를 위한 방향 벡터 반환
3. 맵 오브젝트와 충돌 시 밀림 처리

Skill

기술

Skill – 총알

```
class BBullet
{
public:
    BBoundingBox    m_BoundingBox;          // 총알 충돌바운딩박스
    float           m_fBulletSpeed;         // 총알 속도
    float           m_fBulletIntersection;  // 총알 사거리
    float           m_fOffensePower;        // 총알 공격력
    bool            m_bTrigger;             // 총알 트리거
    TVector3        m_vBulletFirstPos;      // 총알 z방향 초기 위치
    TVector3        m_vBulletPos;           // 총알 위치
    TVector3        m_vBulletDir;           // 총알 방향
    bool            m_bMapCheck;            // 총알 맵 체크
public:
    //-------------------------------------------------------------------
    // 총알 요소 셋팅 함수
    //-------------------------------------------------------------------
    void            SetBullet(float fBulletIntersection,
                              float fOffensePower, float fBulletSpeed);
    //-------------------------------------------------------------------
    // 총알 위치 셋팅 함수
    //-------------------------------------------------------------------
    void            SetBulletPos(TVector3 vPos);
    //-------------------------------------------------------------------
    // 총알 방향 셋팅 함수
    //-------------------------------------------------------------------
    void            SetBulletDir(TVector3 vDir);
    //-------------------------------------------------------------------
    // 총알 실시간 업데이트 함수
    //-------------------------------------------------------------------
    void            UpdateBullet();
    //-------------------------------------------------------------------
    // 총알 사거리 체크
    //-------------------------------------------------------------------
    bool            CheckBulletIntersection();
```

[그림 3-1] 총알 클래스 정의

```
//-------------------------------------------------------------------
// 총알 실시간 업데이트 함수
//-------------------------------------------------------------------
void BBullet::UpdateBullet()
{
    TVector3 vCenter = m_BoundingBox.m_BoundingBoxMax + m_BoundingBox.m_BoundingBoxMin;
    vCenter /= 2.0f;

    // 속도 전진
    m_vBulletPos += m_vBulletDir * m_fBulletSpeed * g_fSecPerFrame;
    m_BoundingBox.m_vCenter = m_vBulletPos + vCenter;
}
```

[그림 3-2] 총알 업데이트 함수

```
//-------------------------------------------------------------------
// 총알 사거리 체크
//-------------------------------------------------------------------
bool BBullet::CheckBulletIntersection()
{
    if (m_vBulletDir.z < 0.0f)
    {
        // 총알 사거리 처리
        if (m_vBulletFirstPos.z + (m_vBulletDir.z * m_fBulletIntersection) > m_vBulletPos.z)
        {
            return false; // 사거리를 넘어감
        }
    }
    else
    {
        // 총알 사거리 처리
        if (m_vBulletFirstPos.z + (m_vBulletDir.z * m_fBulletIntersection) < m_vBulletPos.z)
        {
            return false; // 사거리를 넘어감
        }
    }
    return true; // 사거리 내에 존재
}
```

[그림 3-3] 총알 사거리 체크 함수

- 게임에서 주공격 수단인 총알 객체는 렌더링 되지 않으며 구(Sphere) 바운딩 박스를 통해 충돌 처리
- 트리거가 된 총알은 매 프레임마다 **z 방향으로 직선 운동** 및 **바운딩박스 업데이트**
- 총알 **오브젝트 풀**에 등록된 모든 총알은 매 프레임마다 사거리 계산을 하며 **최대 사거리에서 벗어나면 풀에서 제거**

Skill – 수류탄(1)

```cpp
#define GRAVITY 9.80665f    // 수류탄 중력
#define THROWPOWER 5.0f     // 수류탄 던지는 힘

class BGrenade
{
public:
    BBoundingBox    m_BoundingBox;              // 수류탄 충돌바운딩박스
    float           m_fHeight;                  // 수류탄 높이
    float           m_fVelocity;                // 수류탄 속도
    float           m_fAngle;                   // 수류탄 날아가는 각도
    bool            m_bTrigger;                 // 수류탄 트리거
    TVector3        m_vGrenadeDir;              // 수류탄 방향
    TVector3        m_vGrenadePos;              // 수류탄 위치

    TVector3        m_vVelocity;                // 수류탄 현재 속도
    TVector3        m_vFirstGrenadePos;         // 수류탄 초기 위치
    float           m_fShootTime;               // 수류탄 던져지고난 후 시간
    float           m_fGrenadeSeatedFloorTime;  // 수류탄 바닥에 안착되고난 후 시간
    float           m_fExplosionDelayTime;      // 수류탄 터지는 시간
    float           m_fGrenadeRadius;           // 수류탄 반경
    float           m_fGrenadeAttack;           // 수류탄 공경력
    bool            m_bMapCheck;                // 수류탄 맵 체크
public:
    //-----------------------------------------------------------------
    // 수류탄 던지는 함수
    //-----------------------------------------------------------------
    void            ThrowingAGrenade();
    //-----------------------------------------------------------------
    // 수류탄 위치 셋팅 함수
    //-----------------------------------------------------------------
    void            SetGrenadePos(TVector3 vPos);
```

[그림 3-4] 수류탄 클래스 정의

```cpp
//-----------------------------------------------------------------
// 수류탄 던지는 함수
//-----------------------------------------------------------------
void BGrenade::ThrowingAGrenade()
{
    TVector3 vFloorDir = TVector3(m_vGrenadeDir.x, 0.0f, m_vGrenadeDir.z);
    float fTheta = acosf(D3DXVec3Dot(&m_vGrenadeDir, &vFloorDir)
                         / (D3DXVec3Length(&m_vGrenadeDir) * D3DXVec3Length(&vFloorDir)));
    m_fShootTime += g_fSecPerFrame;

    // THROWPOWER = 5.0f / GRAVITY = 9.80665f
    m_vVelocity.x = m_fVelocity * THROWPOWER * cosf(fTheta);
    m_vVelocity.y = m_fVelocity * THROWPOWER * sinf(fTheta);
    m_vVelocity.z = m_fVelocity * THROWPOWER * cosf(fTheta);

    m_vGrenadePos.x = m_vFirstGrenadePos.x + (m_vGrenadeDir.x * m_vVelocity.x * m_fShootTime);
    m_vGrenadePos.y = m_vFirstGrenadePos.y + (m_vGrenadeDir.y * m_vVelocity.y * m_fShootTime
                      + (0.5f * -GRAVITY * m_fShootTime * m_fShootTime));
    m_vGrenadePos.z = m_vFirstGrenadePos.z + (m_vGrenadeDir.z * m_vVelocity.z * m_fShootTime);
}
```

[그림 3-5] 수류탄 Throwing 함수

포물선 방정식을 이용하여 **수류탄 운동 구현**

Skill – 수류탄(2)

```cpp
void LAParticleBomb::GrenadeFrame()
{
    for (int iGrenade = 0; iGrenade < m_Grenade.size(); iGrenade++)
    {
        // 수류탄이 던져졌는지 체크
        if (m_Grenade[iGrenade].m_bTrigger)
        {
            // 수류탄이 오브젝트와 충돌 체크
            TVector4    vMapCheck = m_pMapParser->GetHeight(m_Grenade[iGrenade].m_vGrenadePos, 0, &m_Grenade[iGrenade].m_bMapCheck);

            if (m_Grenade[iGrenade].m_vGrenadePos.y <= 0.0f)
            {
                // 수류탄 바닥에 안착(충돌시 언덕에 떨어지면 처리 추가)
                m_Grenade[iGrenade].m_vGrenadePos.y = 0.0f;
                // 수류탄 바닥에 안착시 폭파 시간까지 시간 추가
                m_Grenade[iGrenade].m_fGrenadeSeatedFloorTime += g_fSecPerFrame;
                // 수류탄 터지는 시간
                if (m_Grenade[iGrenade].m_fGrenadeSeatedFloorTime >= m_Grenade[iGrenade].m_fExplosionDelayTime)
                {
                    // 수류탄 이펙트 추가
                    m_bIsBomb = true;
                    m_ParticleSystem.push_back(I_ParticleSystem.GetValue(GrenadeEffect));
                    for (int iExp = 0; iExp < m_ParticleSystem.size(); iExp++)
                    {
                        // 수류탄 바운딩 박스 생성
                        m_ParticleSystem[iExp].CreateBoundingBox(m_Grenade[iGrenade].m_vGrenadePos);
                        for (int iEffect = 0; iEffect < m_ParticleSystem[iExp].m_Particle.size(); iEffect++)
                        {
                            // 수류탄 이펙트 위치 셋팅
                            m_ParticleSystem[iExp].m_Particle[iEffect].m_vPos = m_Grenade[iGrenade].m_vGrenadePos;
                            m_ParticleSystem[iExp].m_Particle[iEffect].m_vPos.y += m_ParticleSystem[iExp].m_Particle[iEffect].m_vScale.x - 1.0f;
                        }
                    }
```

[그림 3-7] 수류탄 프레임 함수(1)

```cpp
                    std::vector<std::shared_ptr<LAUnit>>& data = LOAD_OBJECT_MGR.GetMonsterList();
                    int iMonsterCount = data.size();
                    for (int iObj = 0; iObj < iMonsterCount; iObj++)
                    {
                        TVector3 vDamaged;
                        if (CollisionObjectInBombRadian(data[iObj]->m_vObjectPosition))
                        {
                            data[iObj]->SubtractHpDamage(m_Grenade[iGrenade].m_fGrenadeAttack);
                        }
                    }
                    // 터지면 수류탄 삭제
                    m_Grenade.erase(m_Grenade.begin() + iGrenade);
                }
            }
            else
            {
                // 수류탄 이 못가는지역에 도착했을 때 땅으로 떨어짐
                if (m_Grenade[iGrenade].m_bMapCheck == true)
                {
                    m_Grenade[iGrenade].m_vGrenadePos.y -= GRAVITY * g_fSecPerFrame;
                }
                else
                {
                    m_Grenade[iGrenade].ThrowingAGrenade();
                }
            }
        }
    }
}
```

[그림 3-8] 수류탄 프레임 함수(2)

트리거가 활성화된 수류탄은 날아가는 동안 **오브젝트와 충돌 처리**를 하며, 충돌되었을 때는 포물선 운동을 멈추고 y 축으로 중력 가속도를 적용하여 맵 바닥으로 떨어뜨림
맵 바닥에 **안착** 시, 수류탄 **폭발 범위 계산**

ETC

기타

ETC – 초기화 스레드

```
bool Sample::Frame()
{
    if (m_UI.isGameStart && ThreadFunctionCall == false &&g_LoadingNum == 0)
    {
        //쓰레드 콜
        pSample->ThreadFunctionCall = true;
        pSample->Thread_FirstInit = (HANDLE)_beginthreadex(NULL, 0, Thread_Function_FirstInit, NULL, 0, (unsigned*)&pSample->Thread_FirstInit_ID);
        if (pSample->Thread_FirstInit == 0)
        {
            //쓰레드 생성 실패
            exit(1);
        }
    }
```

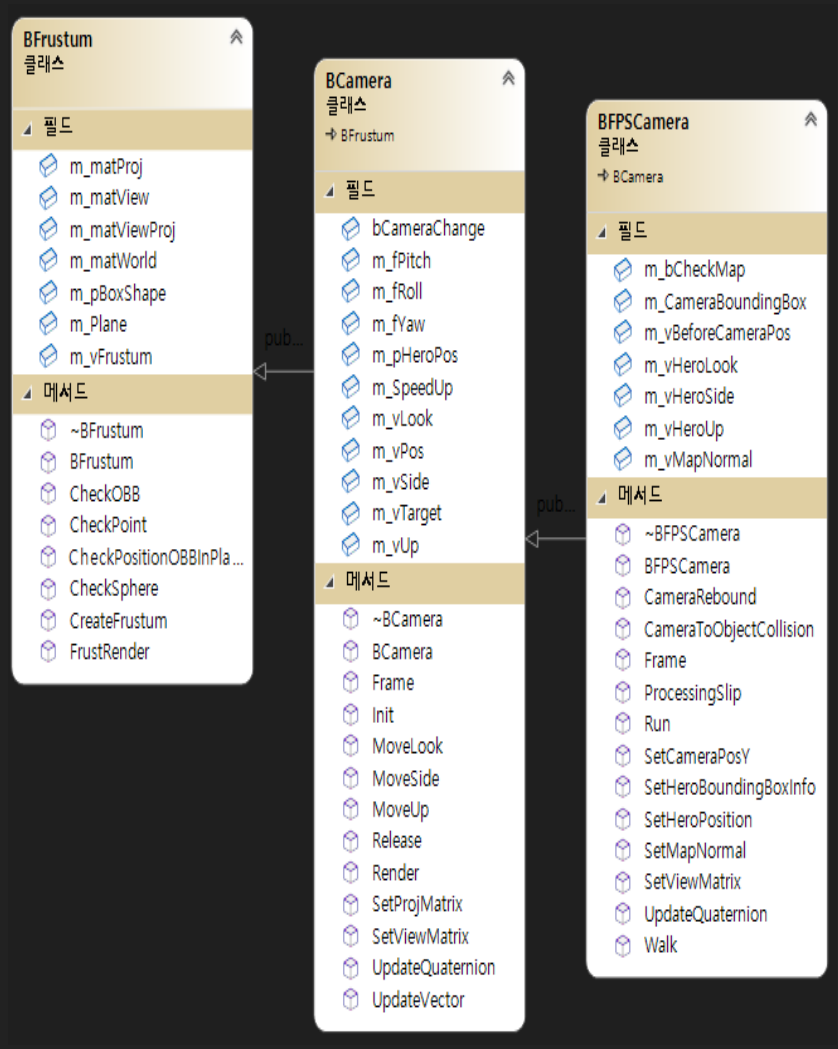
[그림 4-1] 스레드 생성

```
unsigned int WINAPI Thread_Function_FirstInit(void*avg)
{
    pSample->m_Effect.Init();
    g_LoadingNum = 30;
    if (pSample->m_Map.blsmapCreated == false)
    {
        pSample->m_Map.Init();
    }
    g_LoadingNum = 60;

    pSample->m_Character.Init(&pSample->m_Map);
    if (pSample->m_Map.blsmapCreated == false)
    {
        pSample->m_Map.SettingParser(&pSample->m_Character, &pSample->m_UI);
        pSample->m_Map.blsmapCreated = true;
    }
    pSample->m_UI.CharacterUIDataLoad();
    pSample->m_UI.isGameStart = false;

    g_LoadingNum = 100;

    CloseHandle((HANDLE)pSample->Thread_FirstInit_ID);
    return 0;
}
```

[그림 4-2] 게임 초기화 스레드 함수

- 게임 프로그램이 실행되고 **게임 시작 이벤트**가 발생하였을 때, **게임 데이터 생성과 동시에 사운드, 로딩 바(UI)**가 **비동기로 실행**돼야 함
- 프레임 함수에서 **초기화 스레드**를 **생성**하고 **초기화 함수 등록** 및 **호출**

ETC – 카메라(1)



[그림 4-3] 카메라 클래스 다이어그램

```cpp
bool BFPSCamera::SetViewMatrix(TVector3 vPos, TVector3 vTarget, TVector3 vUp)
{
    m_vPos = vPos;
    m_vTarget = vTarget;
    D3DXMatrixLookAtLH(&m_matView, &m_vPos, &m_vTarget, &vUp);

    TMatrix matInvView;
    D3DXMatrixInverse(&matInvView, NULL, &m_matView);

    // yaw/pitch 값을 알아내려면 카메라의 z축 벡터 필요
    TVector3* pZBasis = (TVector3*)&matInvView._31;
    m_fYaw = atan2f(pZBasis->x, pZBasis->z);
    float fLen = sqrtf(pZBasis->z * pZBasis->z + pZBasis->x * pZBasis->x);
    m_fPitch = -atan2f(pZBasis->y, fLen);

    UpdateVector();
    return true;
}
```

[그림 4-4] 카메라 뷰 행렬 생성 및 회전을 위한 축 값 계산

```cpp
bool BFPSCamera::UpdateQuaternion()
{
    TQuaternion qRotation;
    D3DXQuaternionRotationYawPitchRoll(&qRotation, m_fYaw, m_fPitch, m_fRoll);
    TMatrix matRotation;
    ZeroMemory(&matRotation, sizeof(matRotation));
    D3DXMatrixRotationQuaternion(&matRotation, &qRotation);
    D3DXMatrixInverse(&m_matView, NULL, &matRotation);
    UpdateVector();

    m_matView._41 = -(D3DXVec3Dot(&m_vPos, &m_vSide));
    m_matView._42 = -(D3DXVec3Dot(&m_vPos, &m_vUp));
    m_matView._43 = -(D3DXVec3Dot(&m_vPos, &m_vLook));
    return true;
}
```

[그림 4-5] 실시간 카메라 회전 함수

- FPS 카메라 구현
- 카메라 **회전** 시, **짐벌락 문제**를 해결하기 위해 **사원수를 활용**
- 카메라 이동은 **이동하는 방향의 축 벡터**에 **이동 속도**를 **곱**하여 현재 위치에 적용
- 카메라 **충돌**은 **캐릭터의 바운딩박스**를 카메라에 적용하여 충돌 처리

ETC – 카메라(2)

```
//-----------------------------------------------------------------------------
// FPS 카메라와 오브젝트 충돌처리
//-----------------------------------------------------------------------------
void BFPSCamera::CameraToObjectCollision(BBoundingBox& objectBox)
{
    // 충돌체크
    if (m_CameraBoundingBox.m_Collision.CheckSphereInSphere(
        m_CameraBoundingBox.m_vCenter,
        objectBox.m_vCenter,
        m_CameraBoundingBox.m_fRadius,
        objectBox.m_fRadius) == true)
    {
        // 카메라와 오브젝트 정보를 통해 충돌 방향 계산
        TVector3 vBack = m_CameraBoundingBox.m_Collision.SphereInSphereOppositeDir(
            m_CameraBoundingBox.m_vCenter,
            objectBox.m_vCenter,
            m_CameraBoundingBox.m_fRadius,
            objectBox.m_fRadius);

        // 카메라 속도만큼 뒤로 이동
        m_vPos = m_vPos + (vBack * m_SpeedUp * g_fSecPerFrame);

        UpdateQuaternion();
    }
}
```

[그림 4-6] 동적 오브젝트와 충돌 처리 함수

```
    // 눌린 이동 키 수에 따른 속도 처리
    // (눌릴 수 있는 최대 이동 키의 수 = 2개)
    // 대각선으로 이동 시, 직선의 방정식을 적용하여 이동 길이를 계산
    if (iKeyCountCheck >= 2)
    {
        fCameraSpeed = sqrt(pow(m_SpeedUp, 2) + pow(m_SpeedUp, 2));
        fCameraSpeed = fCameraSpeed * g_fSecPerFrame;
    }

    // 벽면 충돌처리
    ProcessingSlip(fCameraSpeed);
}
```

[그림 4-7] 카메라 미끌림 처리(1)

```
//-----------------------------------------------------------------------------
// FPS 카메라 미끌림 처리
//-----------------------------------------------------------------------------
void BFPSCamera::ProcessingSlip(float fSpeed)
{
    if (m_bCheckMap == true)
    {
        // FPS 카메라 벽면 미끌림
        m_CameraBoundingBox.m_Collision.SlipDetection(m_vPos, m_vBeforeCameraPos, fSpeed);
    }
}
```

[그림 4-8] 카메라 미끌림 처리(2)

카메라가 이동할 때,
오브젝트와 충돌이 발생하는 충돌 처리 적용

동적 오브젝트와 충돌: 오브젝트와 충돌된 반대 방향을 계산하여 위치 갱신
정적 오브젝트(맵 이동 불가 지역)와 충돌 : 카메라 밀림 처리

THANK YOU

감사합니다