한국공학대학교 게임공학과

포트폴리오

게임 클라이언트

오정훈

목차

UE5: The Toys

항목	내용
게임 이름	The Toys
게임 장르	비대칭 PvP
기획 의도	대중적인 비대칭 PvP 게임 만들기
개발 인원	3 명(서버 1, 클라 1, 기획/모델 1)
작업 기간	2023/12 ~ 2024/07/29
개발 툴	Unreal Engine 5.3
개발 환경	Windows 11 64bit
나의 역할	클라이언트
타겟 플랫폼	Windows 10/11 64 bit

Git

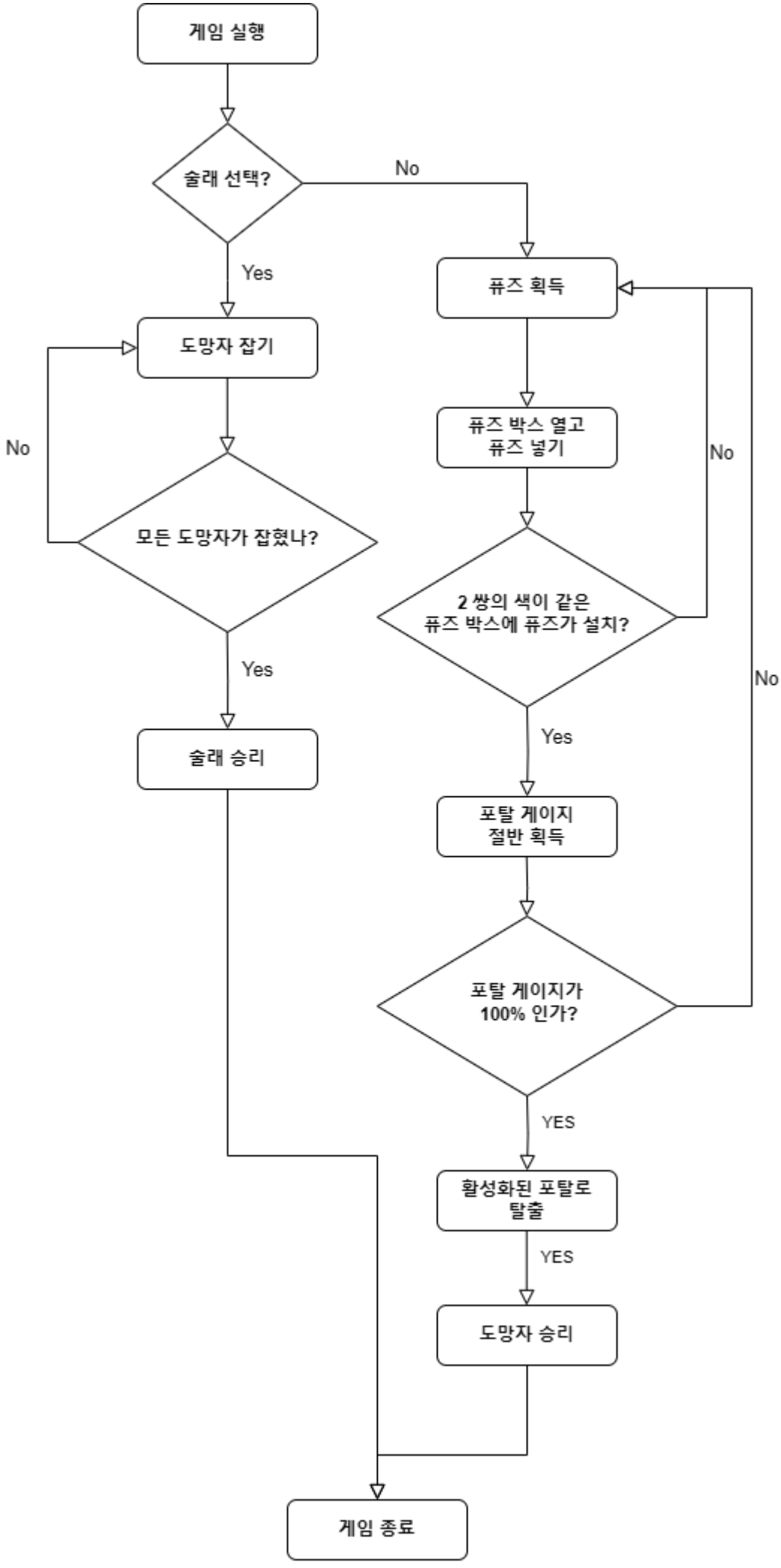
https://github.com/NewbieProgrammerCrew/graduation-project

Youtube

https://www.youtube.com/watch?v=FtYJOIR3v90

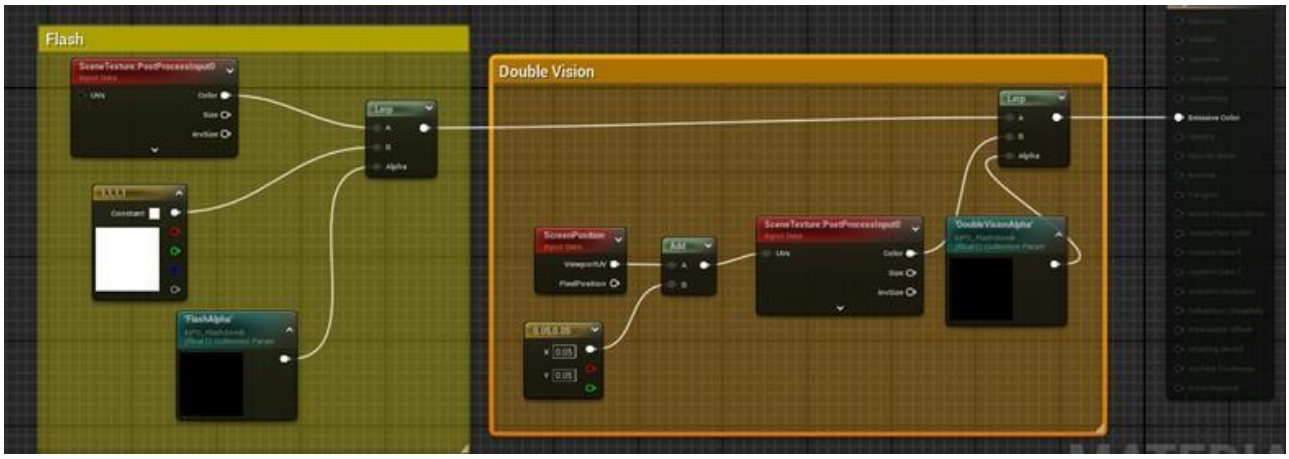
< 인 게임 화면 >

< 게임 플로우 차트 >


게임 실행
술래 선택?
No
Yes
도망자 잡기
No
모든 도망자가 잡혔나?
Yes
술래 승리
퓨즈 획득
퓨즈 박스 열고
퓨즈 넣기
No
2 쌍의 색이 같은
퓨즈 박스에 퓨즈가 설치?
No
Yes
포탈 게이지
절반 획득
포탈 게이지가
100% 인가?
YES
활성화된 포탈로
탈출
YES
도망자 승리
게임 종료

< 구현 >

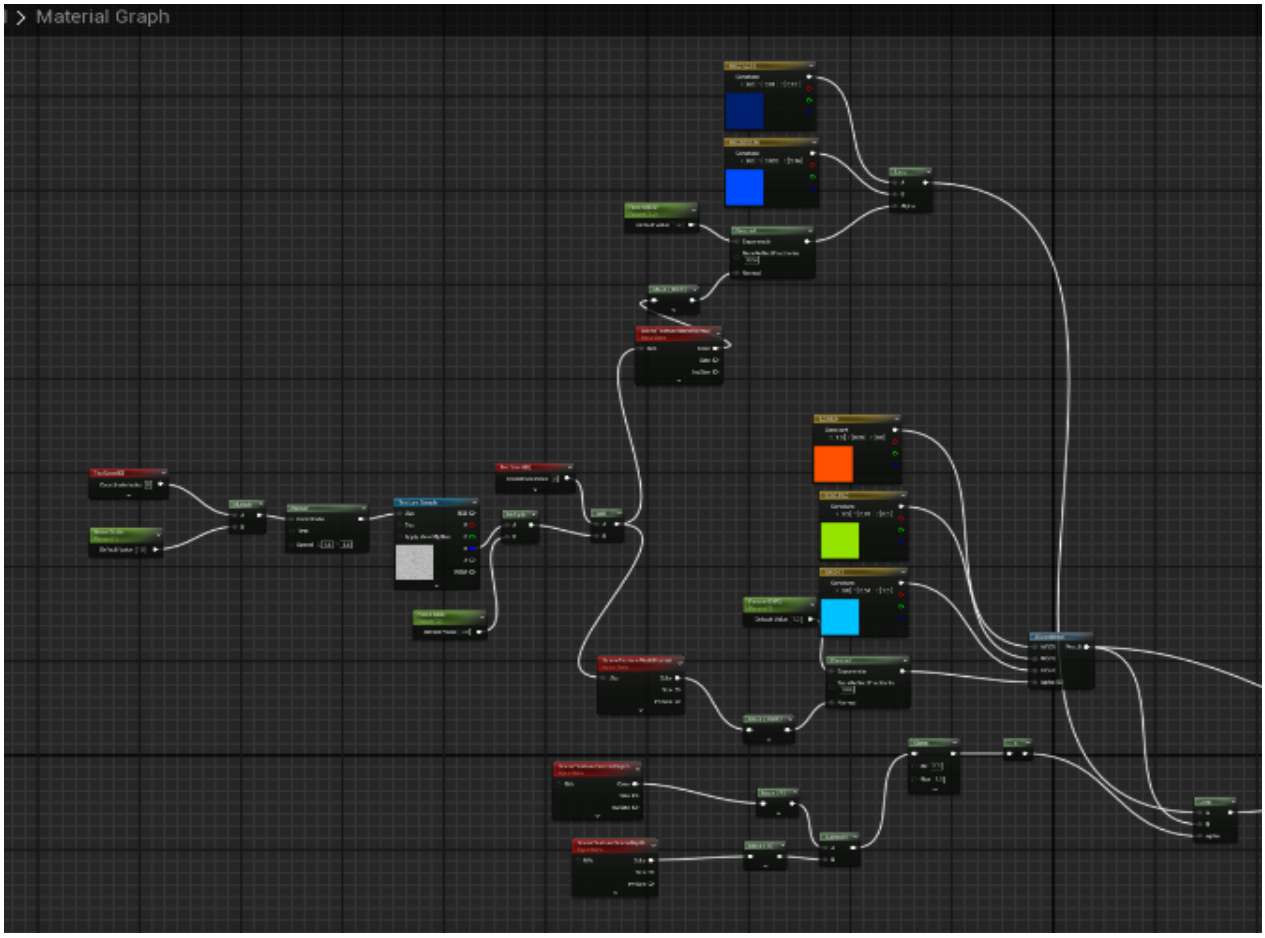
Flash Bomb Effect



포스트 프로세싱 머티리얼로 구현했습니다.
SceneTexture:PostProcessingInput0 노드로 현재 화면 텍스쳐를 가져오고 Vector4값 (흰색)과 Lerp 함수를 ₩
사용하여 alpha값에 따라 시야가 마비되는 효과를 구현했습니다.

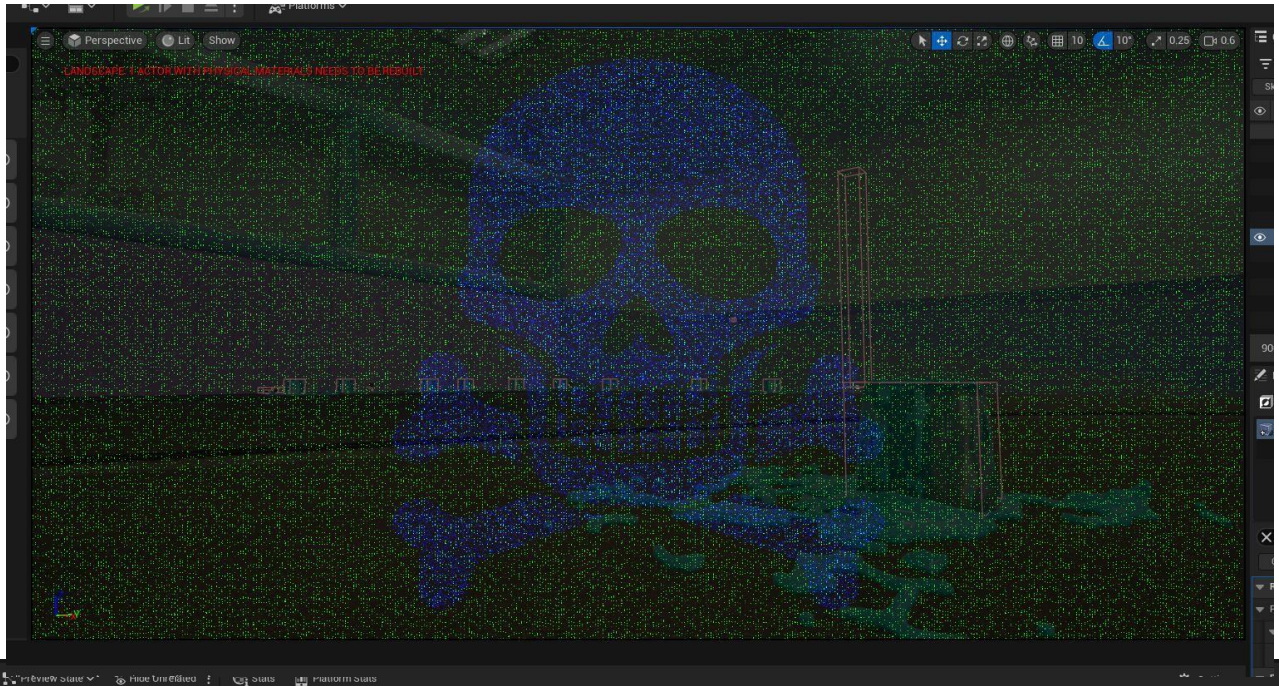
Screen Position의 uv와 Vector2(오프셋)를 더한 값을 SceneTexture:PostProcessingInput0 노드의 새로운 UV로 전달해서 화면이 두 겹으로 보이도록 했습니다.

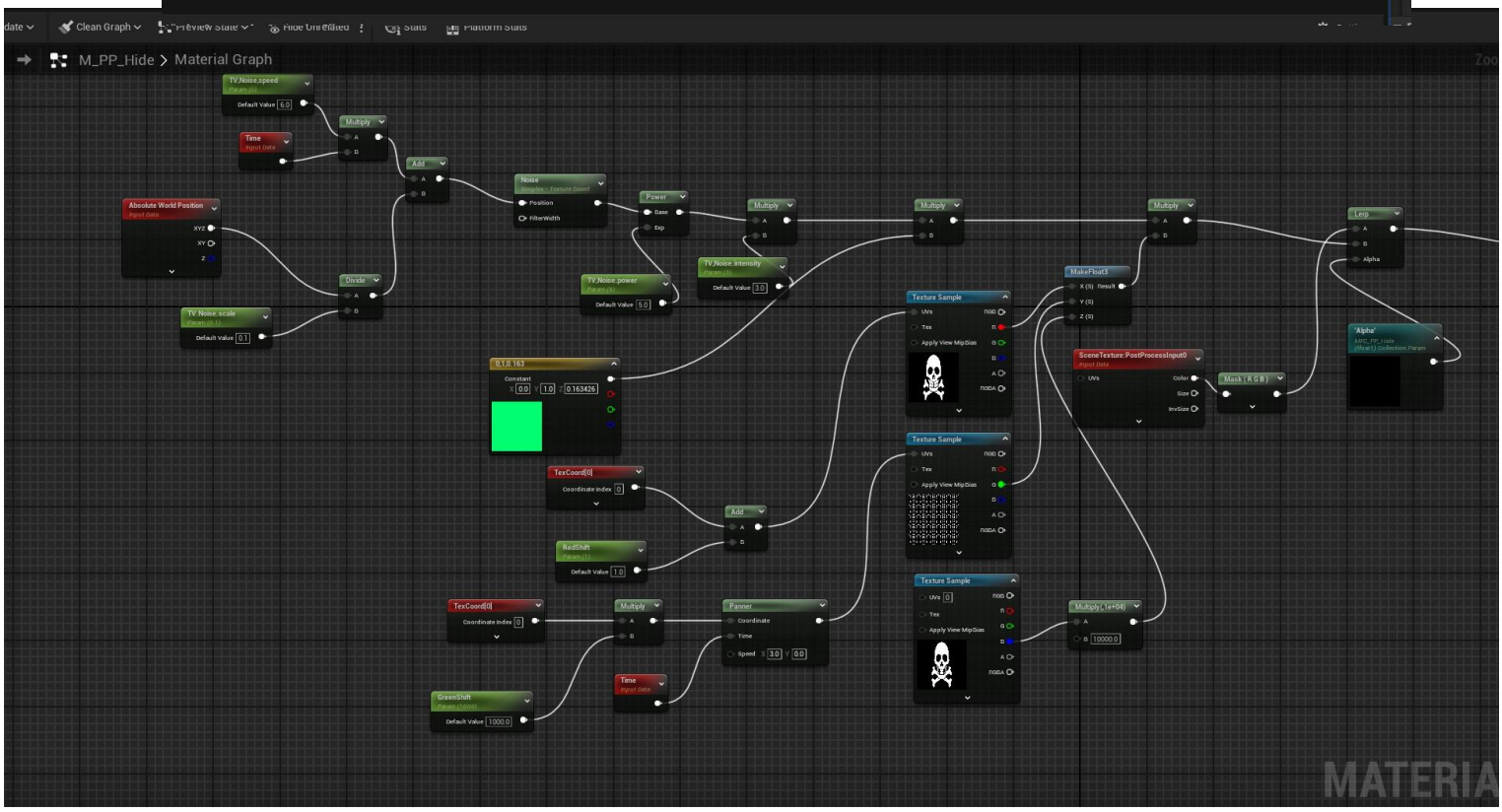
열화상 카메라



캐릭터의 CustomDepth Pass를 활성화하여 CustomDepth 값을 기반으로
빨강, 초록, 파랑 값을 보간하여 캐릭터의 색상을 동적으로 결정하도록 구현했습니다.
또한, 스텐실 버퍼를 사용하여 캐릭터가 벽 뒤에 있을 때도 시각적으로 표시되도록 했습니다.

술래2 스킬 : 모든 도망자 시야를 일시적으로 가리기



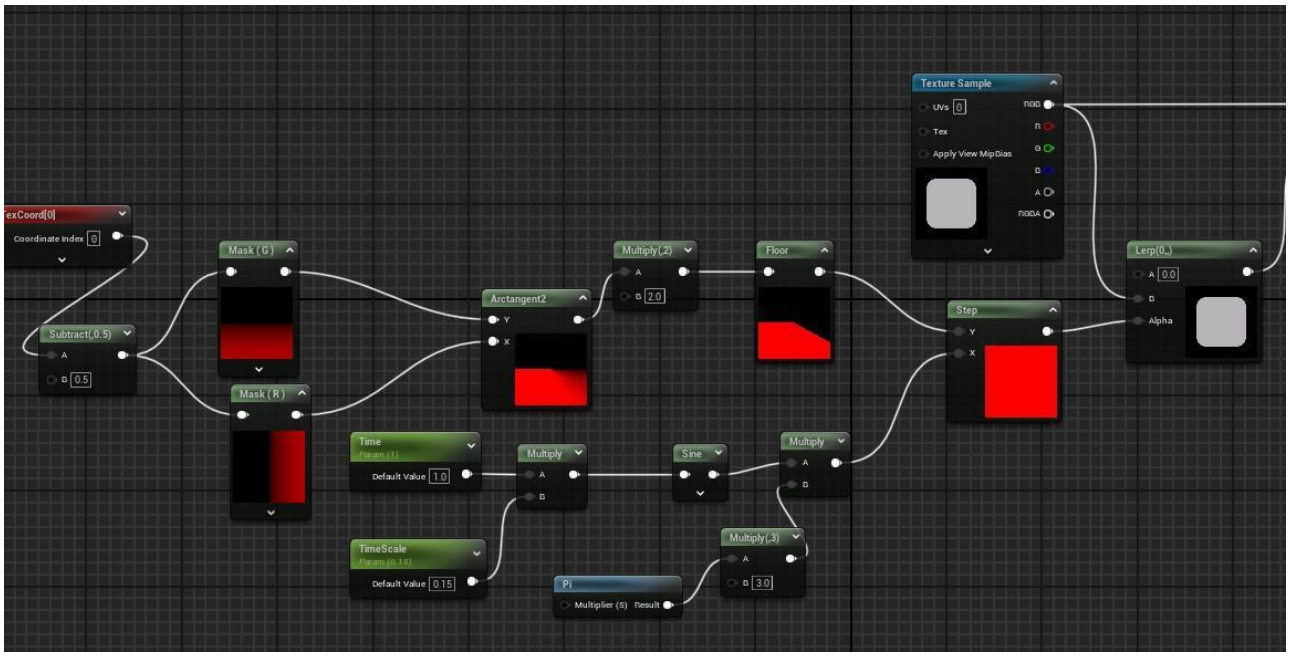


포스트 프로세싱 머티리얼을 사용하여 술래2 스킬을 구현했습니다. TV
의 노이즈 효과를 구현하기 위해서 노이즈 패턴을 생성해야 합니다.
따라서 Absolute World Position 노드를 활용하여 월드 좌표를 기반으로 노이즈 패턴을 생성했습니다.
이를 통해 카메라 움직임과 관계없이 패턴이 월드 공간에 고정되도록 구현하였습니다

그리고 중앙에 해골 텍스처 이미지를 추가하였고 배경은 해골 이미지의 uv좌표를 변경하여 작은 패턴들로 표현했습니다. 그리고 Panner노드로 좌->우로 이동하도록 구현했습니다.

머티리얼 파라미터 콜렉션을 사용하여 lerp의 alpha값을 캐릭터가 조정 가능하도록 구현했습니다. 술래2가 스킬 사용했을 때 타임라인이 호출되어 alpha값을 증가되어 스킬 사용되도록 구현했습니다.

스킬 쿨타임 UI 애니메이션



UV 좌표를 atan2 노드의 인풋 값으로 전달하고, floor 노드를 사용하여 UV 값 모두 내림 처리 했습니다.

그 값을 Step노드로 전달했습니다.

Time 값을 TimeScale로 곱한 후 sin 함수의 입력 값으로 전달하고, pi를 곱한 값을 step 함수의 x 입력 값 으로 사용했습니다.

Step 노드에서 x값이 y값보다 크다면 0, y값이 더 크면 1이 나오게 됩니다. 이 값을 lerp의 알파 값으로 전 달하여 mask부분을 설정해줬습니다.

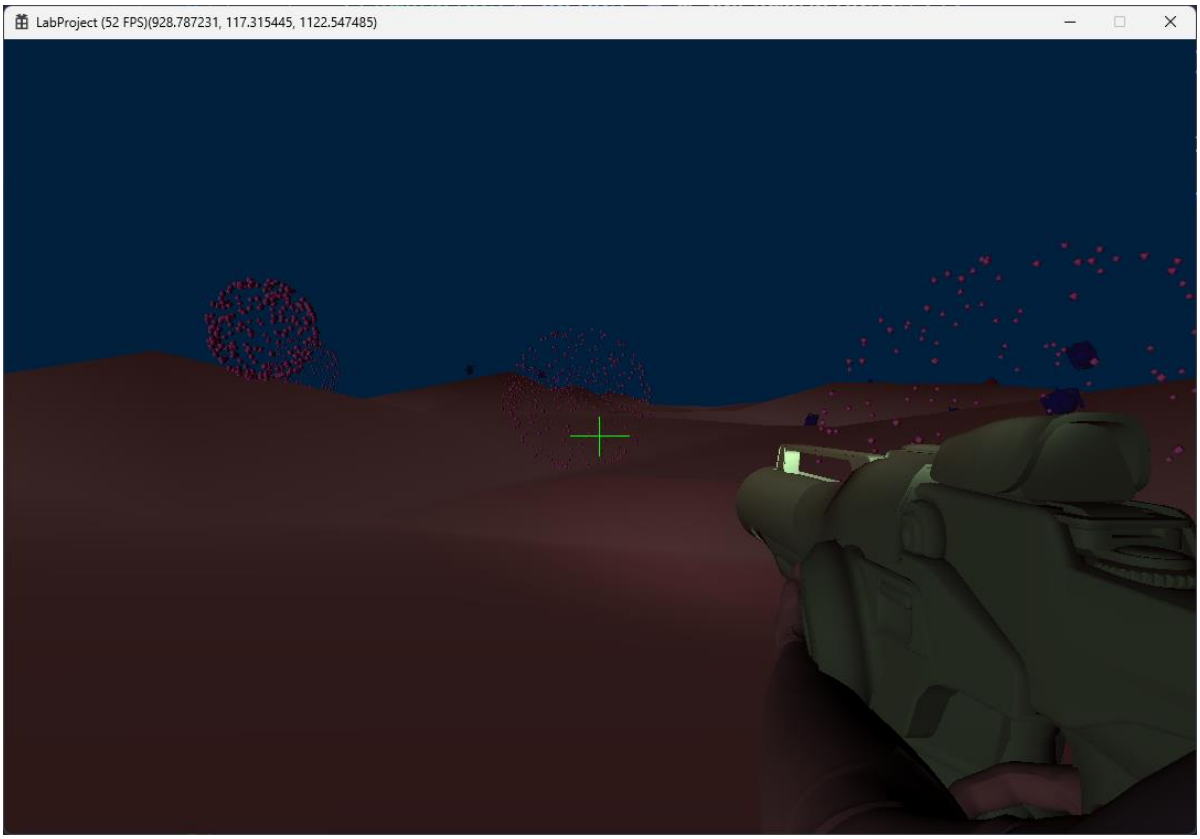
DX12: Drone Strike (3D 게임 프로그래밍 1 텀 프로젝트)

항목	내용
게임 이름	Drone Strike
게임 장르	sc-fi FPS
개발 인원	1 명 (클라 1)
작업 기간	2024/05/31 ~ 2024/06/07
개발 툴	DirectX 12
개발 환경	Windows 11 64bit
타겟 플랫폼	Windows 10/11 64bit

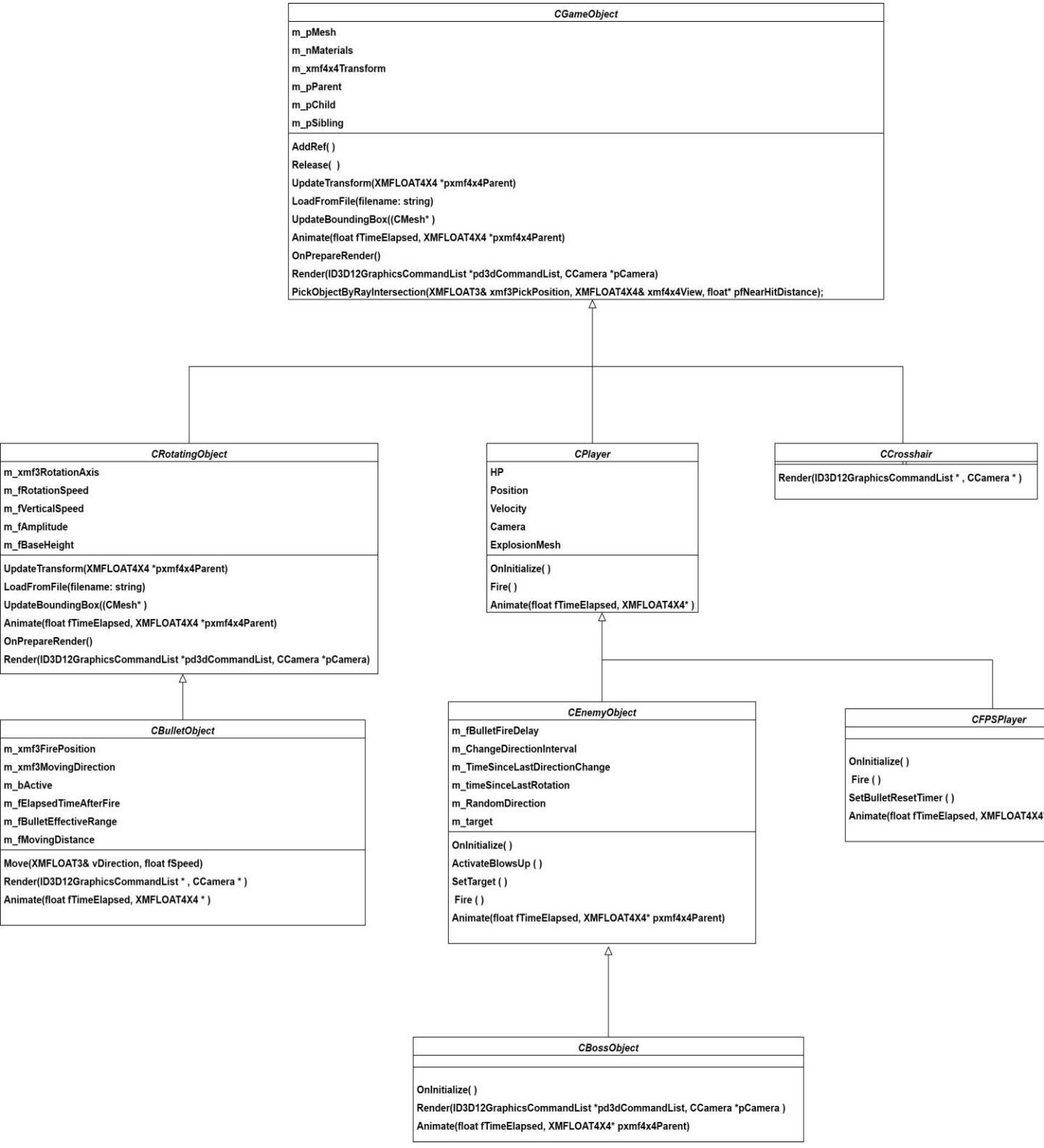
Git
https://github.com/ojh6507/DX12_LabProject/tree/main/LabProject07-9-1

Youtube
https://youtu.be/x7n0MSsS6vg

< 인 게임 화면 >



주요 게임 오브젝트 구조도



크로스헤어

프로젝트 조건으로 텍스처 맵핑을 사용할 수 없었기 때문에

Shader.hlsl에 십자가 모양인 투영 좌표계를 입력했습니다.

PSO를 하나 더 생성하고(쉐이더 추가했기 때문에) 쉐이더의 Input값은 null값을 전달했습니다.

```
//크로스헤어 hlsl 코드
VS_OUTPUT VSProjection(uint vertexID : SV_VertexID)
{
	VS_OUTPUT output;

	float3 positions[4] =
	{
		float3(-0.05f, 0.0f, 0.0f),
		float3(0.05f, 0.0f, 0.0f),
		float3(0.0f, -0.05f, 0.0f),
		float3(0.0f, 0.05f, 0.0f),

	}; //화면 중앙에 위치하는 십자가 좌표 입력

	output.position = float4(positions[vertexID], 1.0f); return output;
}

// Pixel Shader
float4 PSProjection(VS_OUTPUT input) : SV_TARGET
{
	return float4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

```
//크로스헤어 PSO 설정
void CCrosshairShader::CreateShader(ID3D12Device* pd3dDevice,
                                    ID3D12GraphicsCommandList* pd3dCommandList,
                                    ID3D12RootSignature* pd3dGraphicsRootSignature)
{
	m_nPipelineStates = 1;
	m_ppd3dPipelineStates = new ID3D12PipelineState * [m_nPipelineStates];
	// 생략
	.
	.
	.
	m_d3dPipelineStateDesc.InputLayout = { nullptr, 0 }; // 입력 레이아웃 생략
	m_d3dPipelineStateDesc.PrimitiveTopologyType =
D3D12_PRIMITIVE_TOPOLOGY_TYPE_LINE; // 토폴로지는 line으로 설정
	// 생략
	.
	.
	.
}
```

적 오브젝트

주요 기능으로 타겟(플레이어) 추적, 충돌 회피가 있습니다.

적 오브젝트 피격 처리

기존: 플레이어가 발사한 총알의 bounding box와 적 오브젝트의 bounding box의 intersect 비교했었습니다.

문제점: 플레이어를 항상 화면 앞에 렌더링 되도록 구현했습니다.
그러나 적이 플레이어와 가까이 있을 때 충돌 처리가 안되는 버그가 발생했습니다.

개선: 화면 중앙에서 시작하는 Ray와 콜리전 검사로 개선했습니다.

```
//피킹 로직 사용
bool CScene::PickObjectPointedByCursor(float xClient, float yClient, CCamera*
                                       pCamera)
 {
      if (!pCamera) return false;

      // 카메라 행렬과 뷰포트 정보 가져오기
      XMFLOAT4X4 xmf4x4View = pCamera->GetViewMatrix();
      XMFLOAT4X4 xmf4x4Projection = pCamera->GetProjectionMatrix(); D3D12_VIEWPORT
      d3dViewport = pCamera->GetViewport();

      // 화면 좌표를 3D 공간 좌표로 변환
      XMFLOAT3 xmf3PickPosition;
      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; CGameObject*
      pNearestObject = nullptr;

      if (type == InGame) {
           //인 게임 오브젝트와 보스 오브젝트 검사
           for (auto& objList : { m_ppGameObjects, m_ppBossObjects }) {
                for (auto& obj : objList) {
                  // 시야에 보이는 오브젝트만 검사 (바운딩 박스와 카메라 절두체로 검사)
                  if (!obj->IsVisible(m_pCamera)) continue;

                     // 레이 캐스팅을 통한 오브젝트 충돌 검사
                     nIntersected = obj->PickObjectByRayIntersection(xmf3PickPosition, xmf4x4View,
                                                                     &fHitDistance);
                     // 가장 가까운 오브젝트 선택
                     if ((nIntersected > 0) && (fHitDistance < fNearestHitDistance)) {
                          fNearestHitDistance = fHitDistance;
                          pNearestObject = obj;
                          // 이미 폭발 중인 오브젝트는 스킵
                          if (pNearestObject->m_bBlowingUp) continue;
                           if (objList == m_ppBossObjects) {
                                pNearestObject->m_bBlowingUp = true;
                          }
                     }
                }
           }
           // 생략
```

적 오브젝트 피격 처리 (cont'd)

```
// 화면 좌표계의 킹 위치를 월드 좌표계 ray로 변환하는 함수
void CGameObject::GenerateRayForPicking(XMFLOAT3& xmf3PickPosition, XMFLOAT4X4& xmf4x4View,
                                        XMFLOAT3* pxmf3PickRayOrigin,
                                        XMFLOAT3* pxmf3PickRayDirection)
{
    // 월드-뷰 변환 행렬 계산
    XMFLOAT4X4 xmf4x4WorldView = Matrix4x4::Multiply(m_xmf4x4World, xmf4x4View);
    // 월드-뷰 변환 행렬의 역행렬 계산
    XMFLOAT4X4 xmf4x4Inverse = Matrix4x4::Inverse(xmf4x4WorldView);

    //레이의 시작점을 카메라 원점으로 정의하기 위해
    XMFLOAT3 xmf3CameraOrigin(0.0f, 0.0f, 0.0f);
    // 카메라 좌표계의 원점을 모델 좌표계로 변환
    *pxmf3PickRayOrigin = Vector3::TransformCoord(xmf3CameraOrigin, xmf4x4Inverse);
    // 피킹 위치를 모델 좌표계로 변환
    *pxmf3PickRayDirection = Vector3::TransformCoord(xmf3PickPosition, xmf4x4Inverse);
    // 레이의 방향 벡터 계산 및 정규화
    *pxmf3PickRayDirection = Vector3::Normalize(Vector3::Subtract(*pxmf3PickRayDirection,
                                                                  *pxmf3PickRayOrigin));
}
```

```
// 주어진 레이와 메시의 바운딩 박스 간의 교차 검사
int CMesh::CheckRayIntersection(XMFLOAT3& xmf3RayOrigin, XMFLOAT3& xmf3RayDirection,
                                float* pfNearHitDistance)
{
    XMVECTOR xmRayOrigin = XMLoadFloat3(&xmf3RayOrigin);
    XMVECTOR xmRayDirection = XMLoadFloat3(&xmf3RayDirection);

    // 바운딩 박스와 레이의 교차 검사
    bool bIntersected = bbox.Intersects(xmRayOrigin, xmRayDirection, *pfNearHitDistance);
    return bIntersected;
}
```

적 오브젝트 Movement

적 움직임 상태:

- Random Patrol (idle)
- 플레이어 추적
- 일정거리에서 플레이어 공격

문제점 : 플레이어 추적 로직만 추가했을 때 모든 적 오브젝트가 같은 지점에 모이는 문제와 플레이어 위치에 겹쳐지는 문제가 발생했습니다.

개선: 회피 벡터를 추가했습니다. 회피 벡터 적용 후 적 오브젝트들이 동일한 경로로 몰리지 않으며, 플레이어 주위에 분포하게 되었습니다.

주요 함수:

CalculateAvoidanceVector:

- 타겟과의 겹쳐지는 상황을 막기 위해 회피 벡터를 계산합니다.
- 타겟과의 거리에 따라 회피 강도를 조절합니다.

CalculateFinalDirection:

- 타겟 방향과 회피 벡터를 더하여 최종 이동 방향을 정합니다.
- 'separationFactor'를 통해 회피 행동의 강도를 조정할 수 있습니다.

FollowTarget:

- 타겟 추적, 거리에 따른 속도 조절, 회전 시선 처리를 수행합니다.

PerformRandomPatrol:

- 타겟이 없을 때 일정 시간마다 새로운 랜덤 방향을 생성하고 그 방향으로 이동합니다.
- 부드러운 회전을 통해 자연스럽게 움직입니다.

적 오브젝트 Movement

```
    // target: 플레이어
    // 적 객체가 플레이어와 너무 가까워졌을 때 겹쳐질 경우를 방지하기 위한 회피 벡터 계산
    XMFLOAT3 CEnemyObject::CalculateAvoidanceVector(const XMFLOAT3& targetDirection,
                                                    float targetDistance, float avoidanceRadius)
    {
        // 타겟과의 거리가 회피 반경보다 작을 때
        if (targetDistance < avoidanceRadius) {
            //현재 위치 - 타겟 위치 연산으로 타겟 위치에서 현재위치로 가는 벡터를 구합니다.
            XMFLOAT3 avoidance = Vector3::Subtract(GetPosition(), m_target->GetPosition());
            //정규화하고 거리에 반비례하는 크기로 조정
            // 가까울 수록 회피 벡터가 커져야하기 때문에
            return Vector3::ScalarProduct(Vector3::Normalize(avoidance), 1 / targetDistance);
        }
        return XMFLOAT3(0, 0, 0);
    }

    // 타겟 방향 + 회피 방향 = 최종 이동방향 결정
    XMFLOAT3 CEnemyObject::CalculateFinalDirection(const XMFLOAT3& targetDirection,
                                                   const XMFLOAT3& avoidanceVector,
                                                   float separationFactor)
{
    //회피 벡터에 separationFactor가중치 적용, 타겟의 방향과 회피벡터를 더합니다.
    XMFLOAT3 finalDir = Vector3::Add(targetDirection,
                                     Vector3::ScalarProduct(avoidanceVector,  separationFactor));
    // 방향만 필요하기 때문에 정규화
    return Vector3::Normalize(finalDir);
}

   //적이 플레이어를 추적하는 움직임 구현
void CEnemyObject::FollowTarget(const XMFLOAT3& direction, float distance, const XMFLOAT3& newLook,
                                float followDistance, float deadZone, float frameTime)
{
    //distance: 현재 위치에서 타겟까지의 거리
    //newLook: 새로 계산된 시선 방향 (부드러운 회전처리를 위함)
    //followDistance: 타겟 추적 가능한 거리

    //타겟과의 거리에 따른 속도 조절 speedFactor 계산
    //멀어질수록 속도 증가, deadZone에 가까워질수록 속도 감소
    float speedFactor = (distance - (followDistance - deadZone)) / (deadZone + 1);
    float adjustedSpeed = m_fMovingSpeed * speedFactor * frameTime; //fps 고려
    XMFLOAT3 shift = Vector3::ScalarProduct(direction, adjustedSpeed); // 최종 이동 벡터 계산

    LookAt(m_target->GetPosition(), newLook); //타겟 방향으로 시선 회전
    AdjustMovementToTerrain(shift); //터레인에 맞춰 이동 벡터 조정
    Move(shift, false);
}
```

적 오브젝트 Movement (cont'd)

```
//randomPatrol
void CEnemyObject::PerformRandomPatrol(float elapsedTime, const XMFLOAT3& newLook, float frameTime)
{
	// 마지막 방향 변경 이후 시간 누적
	m_TimeSinceLastDirectionChange += elapsedTime;

	// 설정된 간격마다 새로운 랜덤 방향 생성
	if (m_TimeSinceLastDirectionChange >= m_ChangeDirectionInterval)
	{
		m_RandomDirection = GenerateRandomDirection();
		m_TimeSinceLastDirectionChange = 0.0f;
	}
	// 현재 방향과 속도를 기반으로 이동량 계산
	XMFLOAT3 shift = Vector3::ScalarProduct(m_RandomDirection, m_fMovingSpeed * frameTime);
	// 새로운 위치 계산
	XMFLOAT3 newpos = Vector3::Add(XMFLOAT3(m_xmf4x4World._41,
	                                        m_xmf4x4World._42,
	                                        m_xmf4x4World._43),
	                               shift);

	// 위치가 변경되었다면 새 방향을 향해 회전
	if (!XMVector3Equal(XMLoadFloat3(&m_xmf3Position), XMLoadFloat3(&newpos))) {
		LookAt(newpos, newLook);
	}

	// 현재 보는 방향을 랜덤 방향으로 설정
	m_xmf3Look = m_RandomDirection;

	AdjustMovementToTerrain(shift);
	Move(shift, false);
}
```

적 오브젝트 폭발 파티클

```
void CPlayer::PrepareExplosion(ID3D12Device* pd3dDevice,
                               ID3D12GraphicsCommandList*pd3dCommandList)
{
    //폭발 파티클 메쉬 설정과 구 표면 상의 분포된 랜덤한 단위 벡터를 생성
    for (int i = 0; i < EXPLOSION_DEBRISES; i++) XMStoreFloat3(&m_pxmf3SphereVectors[i],
                                                    Random::RandomUnitVectorOnSphere());
    m_pExplosionMesh = new CCubeMesh(pd3dDevice, pd3dCommandList, 3.f, 3.f, 3.f);
}

void CEnemyObject::Animate(float fTimeElapsed, XMFLOAT4X4* pxmf4x4Parent)
{
    CPlayer::OnPrepareRender();
    if (m_bBlowingUpAvailable) {
      // 폭발 지연
        m_fElapsedBlowupTimes += fTimeElapsed; if
        (m_fElapsedBlowupTimes > m_fDelay) {
            m_bBlowingUp = true; //폭발 활성화
        }
    }
    if (m_bBlowingUp) {
        m_fElapsedBlowupTimes = 0;
        m_bBlowingUpAvailable = false;

      // 폭발 진행 시간 누적
        m_fElapsedTimes += fTimeElapsed;
        if (m_fElapsedTimes <= m_fDuration) { // 폭발 진행 중
            XMFLOAT3 xmf3Position = GetPosition();
            for (int i = 0; i < EXPLOSION_DEBRISES; i++) {
                // 각 파티클의 변환 행렬 초기화
                m_pxmf4x4Transforms[i] = Matrix4x4::Identity();
                // 파티클의 새로운 위치 계산
                // m_pxmf3SphereVectors는 미리 계산된 무작위 방향 벡터
                // m_fExplosionSpeed로 폭발의 확산 속도 조절
                m_pxmf4x4Transforms[i]._41 = xmf3Position.x + m_pxmf3SphereVectors[i].x *
                m_fExplosionSpeed * m_fElapsedTimes;
                m_pxmf4x4Transforms[i]._42 = xmf3Position.y + m_pxmf3SphereVectors[i].y *
                                             m_fExplosionSpeed * m_fElapsedTimes;
                m_pxmf4x4Transforms[i]._43 = xmf3Position.z + m_pxmf3SphereVectors[i].z *
                                            m_fExplosionSpeed * m_fElapsedTimes;

                // 파티클 회전 적용
                XMVECTOR axis = XMLoadFloat3(&m_pxmf3SphereVectors[i]);

                // m_fExplosionRotation으로 회전 속도 조절
                XMMATRIX rotationMatrix = XMMatrixRotationAxis(axis, m_fExplosionRotation
                                                               * m_fElapsedTimes);
                XMMATRIX transformMatrix =XMLoadFloat4x4(&m_pxmf4x4Transforms[i]);
                transformMatrix = XMMatrixMultiply(rotationMatrix, transformMatrix);
                XMStoreFloat4x4(&m_pxmf4x4Transforms[i],  transformMatrix);
            }
        }

}
```

Android : Pixel Game (스마트폰 게임 프로그래밍)

항목	내용
게임 이름	Pixel Game
게임 장르	인디 게임, 전략 게임
개발 인원	1 명 (클라 1)
작업 기간	2024/04 ~ 2024/06
개발 툴	Android Studio / Aseprite
개발 환경	Windows 11 64bit
타겟 플랫폼	Android API 34

Git
https://github.com/ojh6507/SPGTermProject

Youtube
https://youtu.be/dSSu7duatmk