한국공학대학교 게임공학과

포트폴리오

게임 클라이언트

오정훈

목차

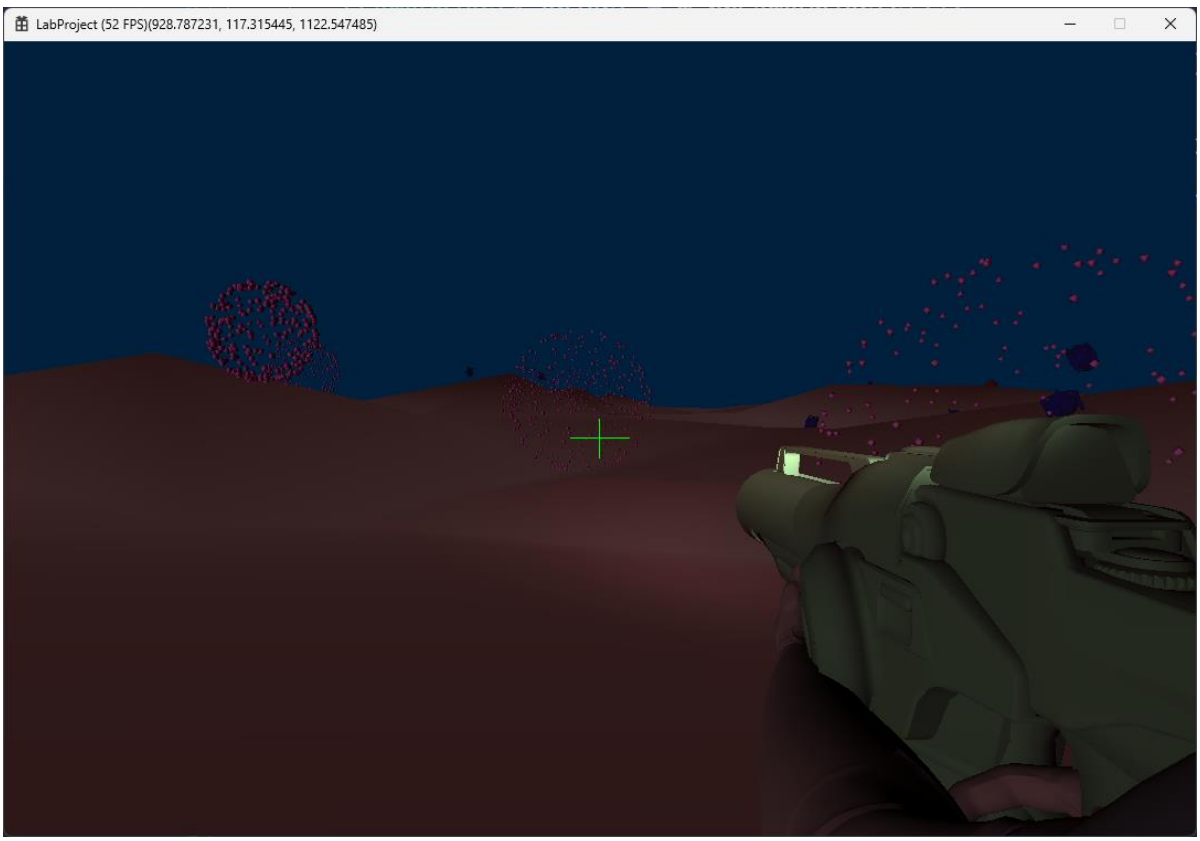
DX12: Drone Strike (3D 게임 프로그래밍 1 텀 프로젝트)

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

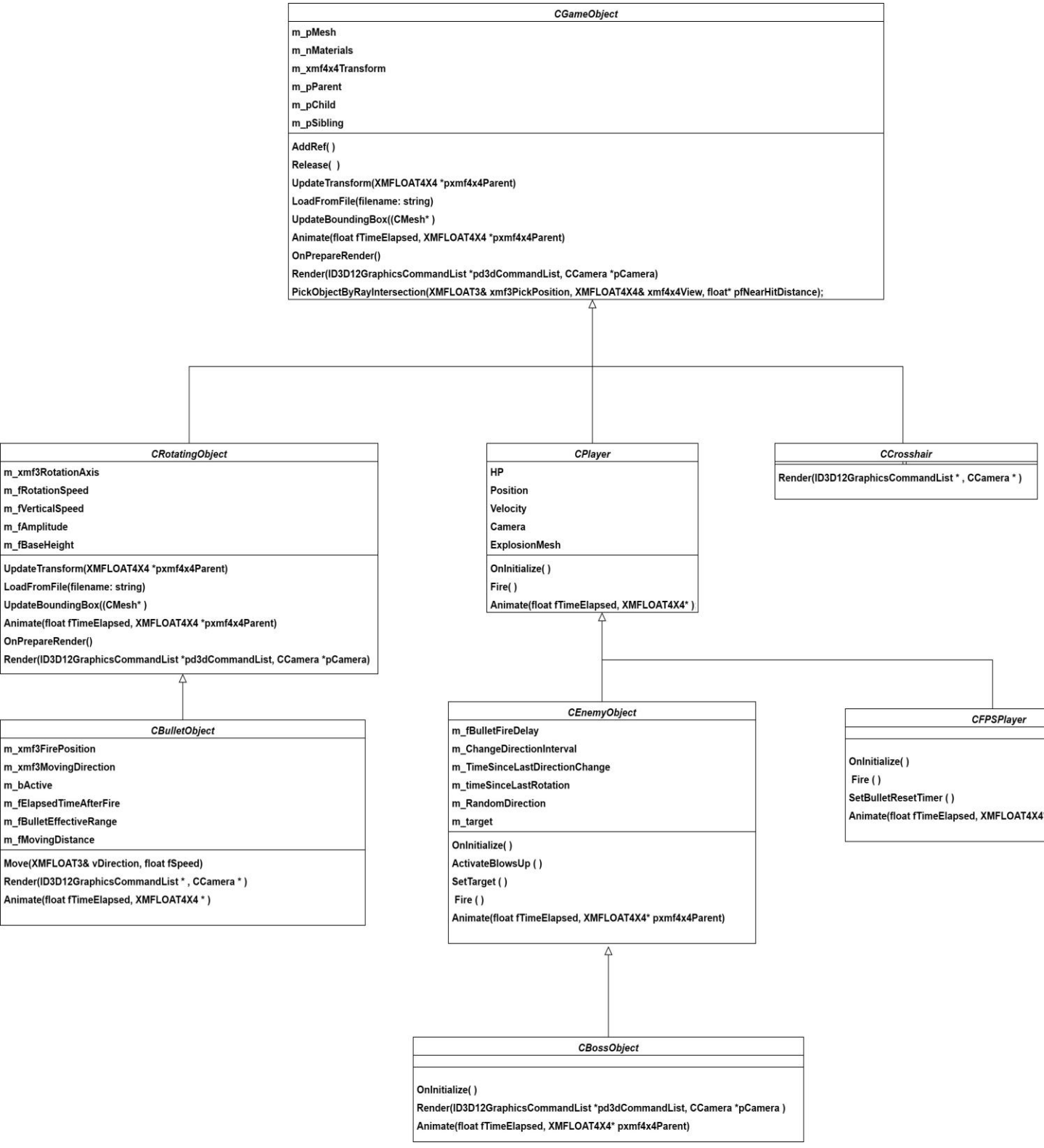
Git
https://github.com/ojh6507/DX12_LabProject/tree/main/LabProject07-9-1

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 메서드:

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);
            }
        }

}
```

적 오브젝트 폭발 파티클 (cont'd)

```
void CEnemyObject::Render(ID3D12GraphicsCommandList* pd3dCommandList, CCamera* pCamera)
{
	if (m_bBlowingUp) {
		// 폭발이라면 폭발 파티클 큐브 렌더링
		for (int i = 0; i < EXPLOSION_DEBRISES; i++) {
		   // 각 파티클의 월드 변환 행렬을 설정
			m_exp[i]->m_xmf4x4World = m_pxmf4x4Transforms[i];
			m_exp[i]->Render(pd3dCommandList, pCamera); // 렌더링
		}
	}
	else {
		CPlayer::Render(pd3dCommandList, pCamera);
	}
	for (auto& obj : m_ppBullets) {
		if (obj->m_bActive)obj->Render(pd3dCommandList, pCamera);



	}
}
```

Android : Pixel Game (스마트폰 게임 프로그래밍)

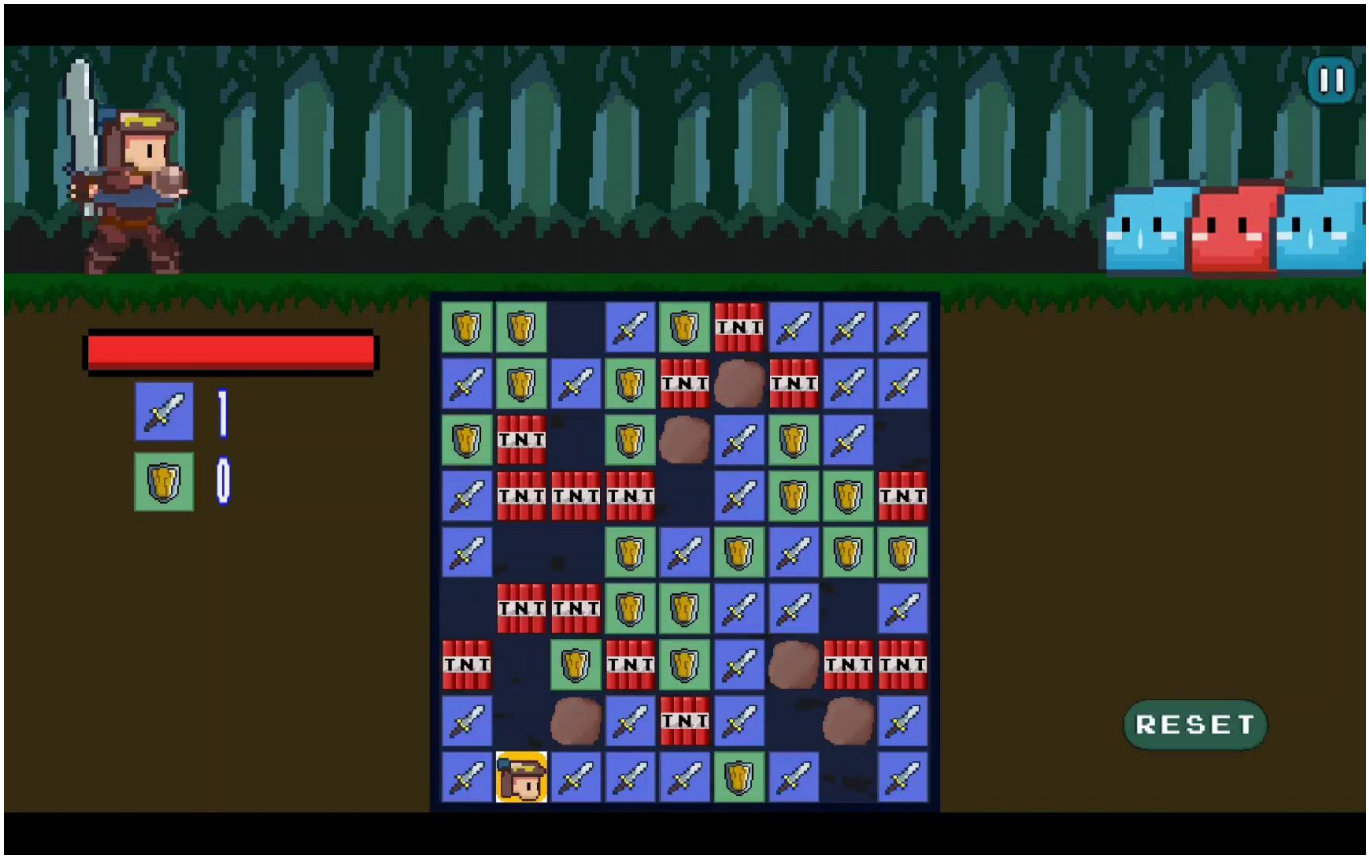
항목	내용
게임 이름	Pixel Game
게임 장르	인디 게임, 전략 게임
참여 인원	1 명
작업 기간	2024/04 ~ 2024/06
개발 툴	Android Studio / Aseprite
개발 환경	Windows 11 64bit
타겟 플랫폼	Android API 34
상태	완료

Git

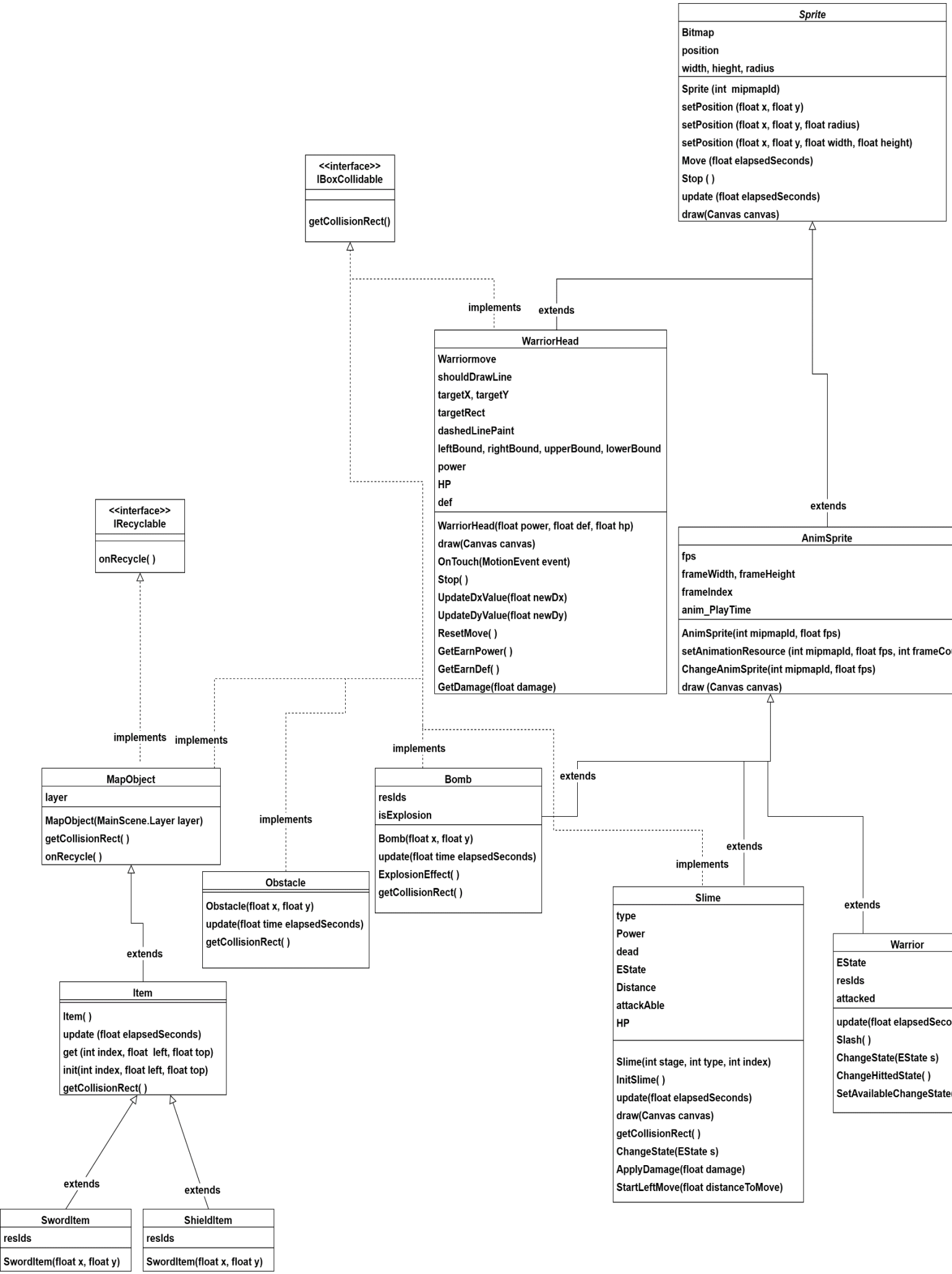
https://github.com/ojh6507/SPGTermProject

Youtube

https://youtu.be/dSSu7duatmk



주요 게임 오브젝트 구조도



충돌 후 반사각으로 이동

```
//----------------- CollisionChecker.java -----------------
    public void updateDirectionAfterCollision(RectF obstacleRect, RectF headRect) {
        float incidentX = this.warriorHead.GetDx();
        float incidentY = this.warriorHead.GetDy();

        // 충돌 방향에 따른 법선 벡터 설정
        float normalX = 0;
        float normalY = 0;

        float deltaX = (headRect.left + headRect.width() / 2) -
                       (obstacleRect.left + obstacleRect.width() / 2);
        float deltaY = (headRect.top + headRect.height() / 2) -
                       (obstacleRect.top + obstacleRect.height() / 2);

        if (Math.abs(deltaX) > Math.abs(deltaY)) {  // 수평 충돌
            normalX = Math.signum(deltaX);
        } else {                                    // 수직 충돌
            normalY = Math.signum(deltaY);
        }

        // 내적 계산
        float dotProduct = incidentX * normalX + incidentY * normalY;

        // 반사 벡터 계산
        float reflectedX = incidentX - 2 * dotProduct * normalX;
        float reflectedY = incidentY - 2 * dotProduct * normalY;

        // 반사된 벡터로 방향 갱신
        this.warriorHead.UpdateDxValue(reflectedX);
        this.warriorHead.UpdateDyValue(reflectedY);

        // 충돌 후 위치 조정하기 위해 계산.
        float pushBackDistance = 0.05f;
        if (normalX != 0) {
            this.warriorHead.SetX(pushBackDistance * normalX);
        }
        if (normalY != 0) {
            this.warriorHead.SetY(pushBackDistance * normalY);
        }

    }
```

updateDirectionAfterCollision 는 캐릭터가 벽이나 장애물과 충돌했을 때 반사되어 움직이도록 처리하는 함수입니다.

먼저, GetDx와 GetDy 함수로 캐릭터의 현재 이동 방향을 가져옵니다.
캐릭터 중심과 장애물 중심 사이의 거리를 계산하여, 충돌이 수평인지 수직방향인지 판단하여 법선 벡터의 x 또는 y 성분을 구합니다.

법선 벡터를 계산하면, 입사 벡터와 법선 벡터의 내적을 계산하여 반사 벡터를 계산합니다.
계산된 반사 벡터를 이용해 캐릭터의 이동방향을 구합니다.

적 생성

```
//----------------- EnemyGenerator.java -----------------
private static final int[][] slimeTypePerStage = {
        {0,1,0},
        {2,1,1,2,1},
        {3,2,3,1,1,0,2,3,2,3}
}; //총 3가지 스테이지, 각 스테이지에 스폰될 몬스터 타입 입력.

private void generate(int stage) {
    if (scene == null) return;
    int numberOfSlimes = slimeCountPerStage[stage - 1];
    //stage는 1부터 시작, 배열 순회하며 mainScene에 slime 추가.
    for (int i = 0; i < numberOfSlimes; i++) {
       scene.add(MainScene.Layer.enemy,
                 Slime.get(stage,slimeTypePerStage[stage-1][i],i));
    }
}
```

```
//----------------- Slime.java -----------------

private Slime(int stage, int type, int index) {
    super(resIds[type][0], ANIM_FPS); //몬스터 타입에 맞는 리소스 지정
    this.level = stage;
    this.type = type;
    if(type == 2) RADIUS =0.9f; // 리소스 크기 차이로 인한 Radius조정
    else if(type == 3) RADIUS =1.4f; // 리소스 크기 차이로 인한 Radius조정
    setPosition(Metrics.width -RADIUS* 3.5f + index,
                Metrics.height/(3 * RADIUS + 2.2f), RADIUS);
    dead =false;
    InitSlime();
}

public void InitSlime( ) {
// 몬스터 타입에 따라 hp, 공격력, 속도, 한번에 이동 가능 한 거리 설정.
    if(type == 0 ){
        HP = 30.f;
        fDistance = 3.f;
        SPEED = 3.f;
        fPower = 5.f;
    }
    else if(type == 1){
        HP = 60.f;
        fDistance = 2.f;
        SPEED = 2.f;
```

스테이지별로 생성될 타입과 수를 2차원 배열에 미리 정의하여,
각 스테이지에서 등장할 몬스터를 결정하고 생성합니다.

객체 생성 과정에서, 몬스터의 타입과 스테이지 정보를 매개변수로 전달하여,
InitSlime( )에서 몬스터 타입에 따라 체력, 이동 속도, 공격력 등을 설정합니다.

UE5: The Toys

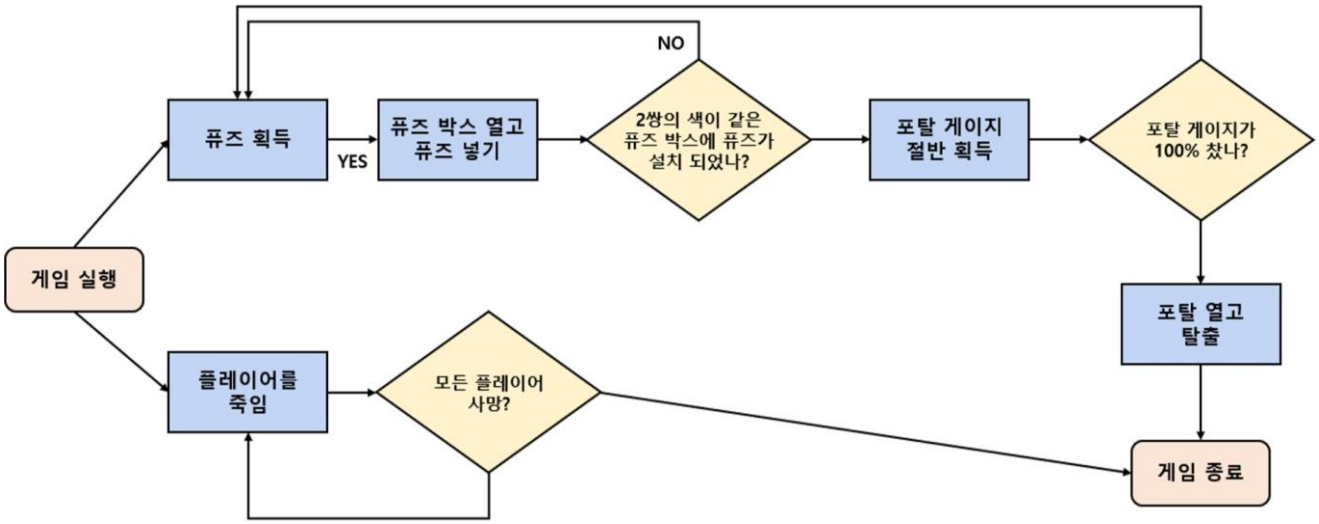
항목	내용
게임 이름	The Toys
게임 장르	비대칭 PvP
기획 의도	대중적인 비대칭 PvP 게임 만들기
참여 인원	3 명
작업 기간	2023/12 ~ 2024/07/29
개발 툴	Unreal Engine 5.3
개발 환경	Windows 11 64bit
나의 역할	게임 사운드 적용, 오브젝트 상호작용, 이펙트 구현, 캐릭터 스킬 구현
타겟 플랫폼	Windows 10/11 64 bit
상태	완료

Git

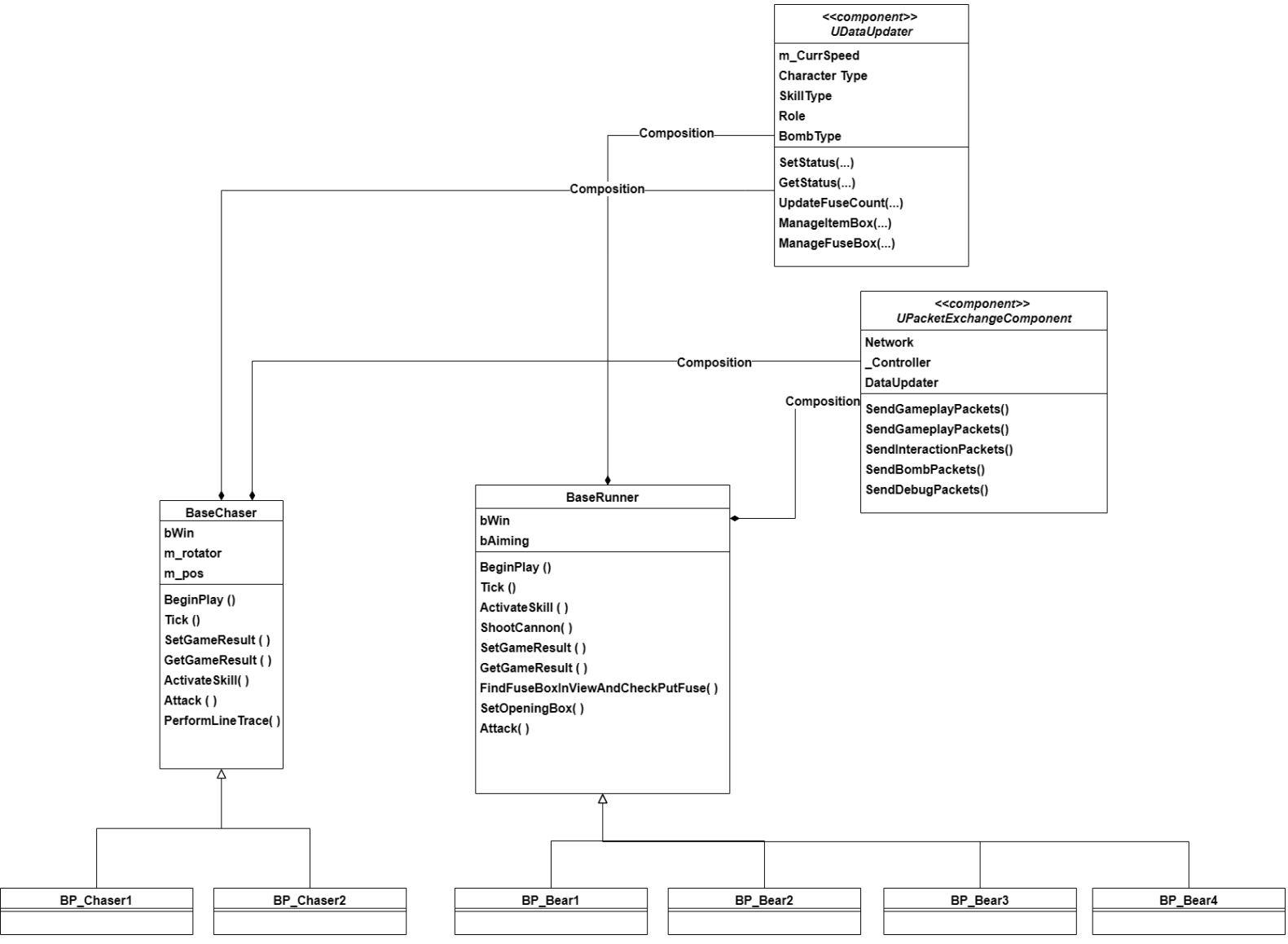
https://github.com/NewbieProgrammerCrew/graduation-project

< 인 게임 화면 >

< 게임 플로우 차트 >



< 캐릭터 구조도 >



< 구현 >

Bomb 관리

Bomb 획득

Bomb의 타입 정보는 서버로부터 "SC_PICKUP_BOMB_PACKET"으로 전달 받습니다.

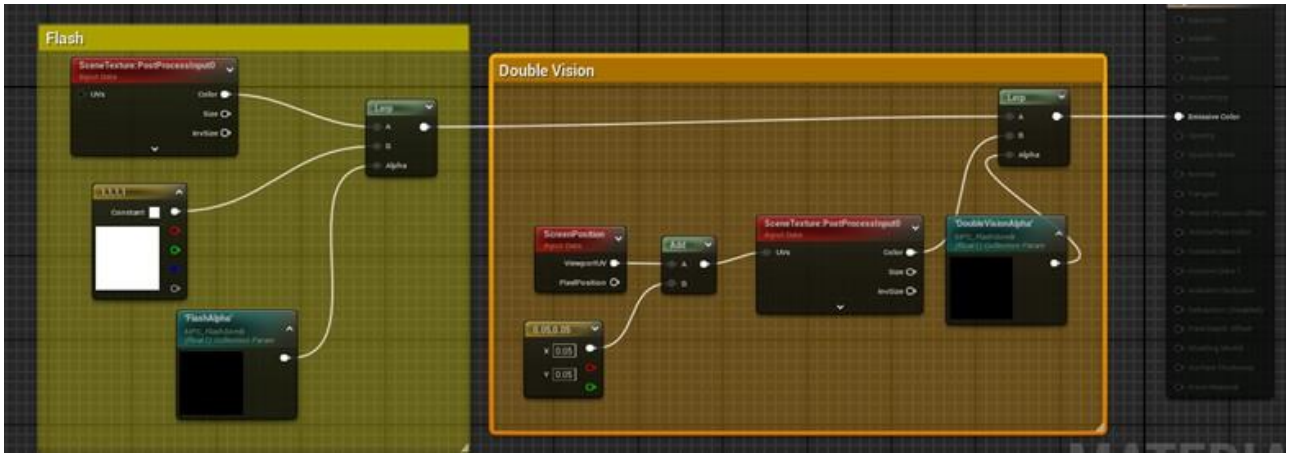
- 이전 코드 문제점: 월드에 스폰된 폭탄 객체를 캐릭터의 Cannon mesh에 붙이는 방식인데 기존 폭탄과 새로운 폭탄 교체를 빠르게 반복하면 폭탄이 "중복 스폰" 문제가 발생했습니다.
- 개선 사항: 폭탄 타입의 정보만 저장하고, Cannon 메시에 미리 배치한 Bomb Mesh를 Dynamic Material Instance로 폭탄 타입에 따라 색상을 변경하고 Visible를 true로 설정하였습니다. 폭탄 객체는 캐릭터가 발사할 때 생성되도록 수정했습니다.
- Bomb 획득 개선한 코드

```
void APlayerManager::Player_Bomb_Pickup (SC_PICKUP_BOMB_PACKET   item_pickup_player)
{
// 패킷을 받았을 때 폭탄 객체 생성이 아닌 타입 정보만 캐릭터 클래스에 저장하도록 수정
    //.. 코드 생략..
      ABaseRunner* runnerInstance = Cast<ABaseRunner>(playerInstance); if (runnerInstance) {
            runnerInstance->EquipBomb(BombType(item_pickup_player.bombType));
      }

}
```

Flash Bomb Effect

- **구현 과정**

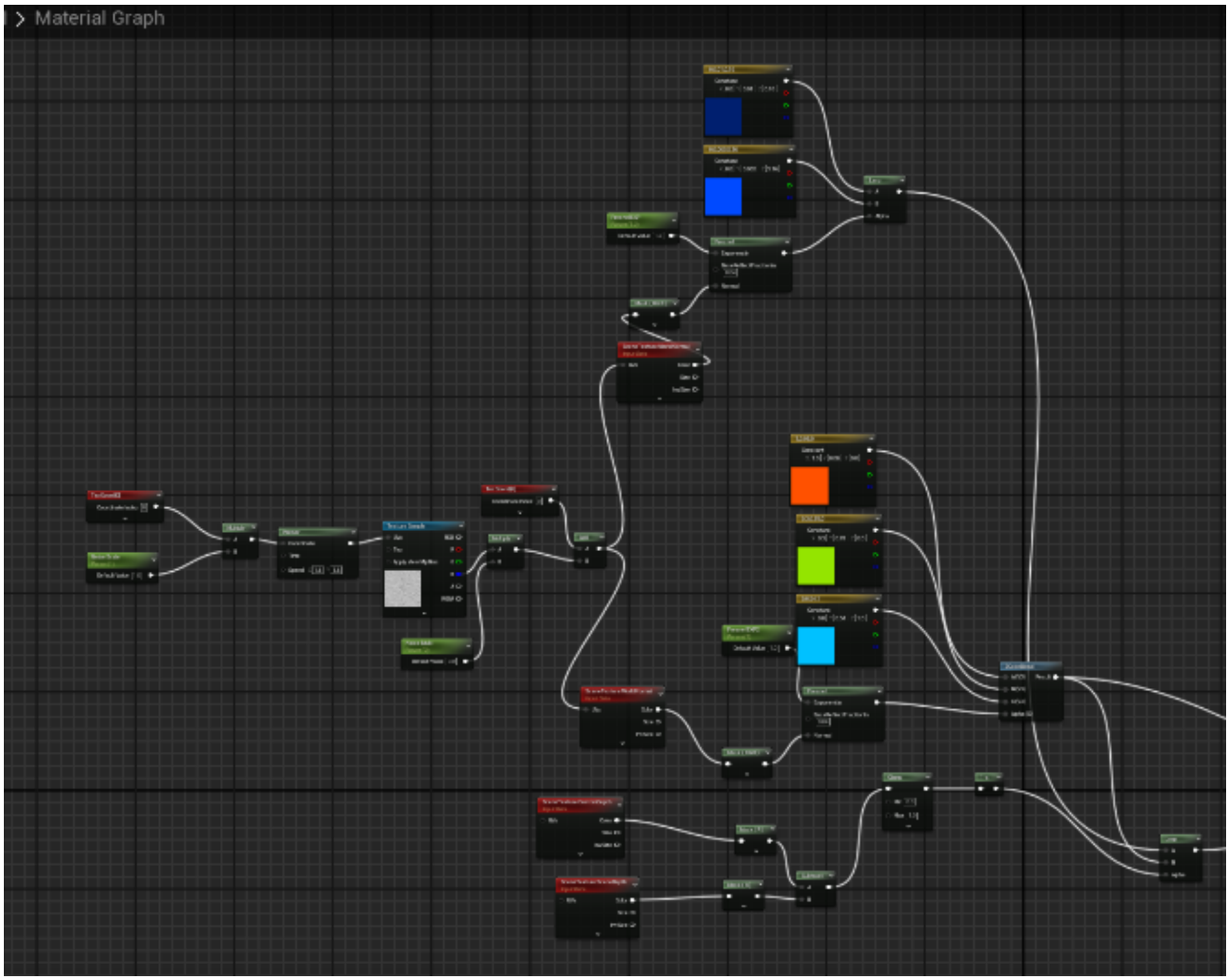


포스트 프로세싱 머티리얼로 구현했습니다. SceneTexture:PostProcessingInput0 노드로 현재 화면 텍스쳐를 가져오고 Vector4값 (흰색)과 Lerp 함수를 사용하여 alpha값에 따라 시야가 멀어지는 효과를 구현했습니다.

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

열화상 카메라

- **구현 과정**

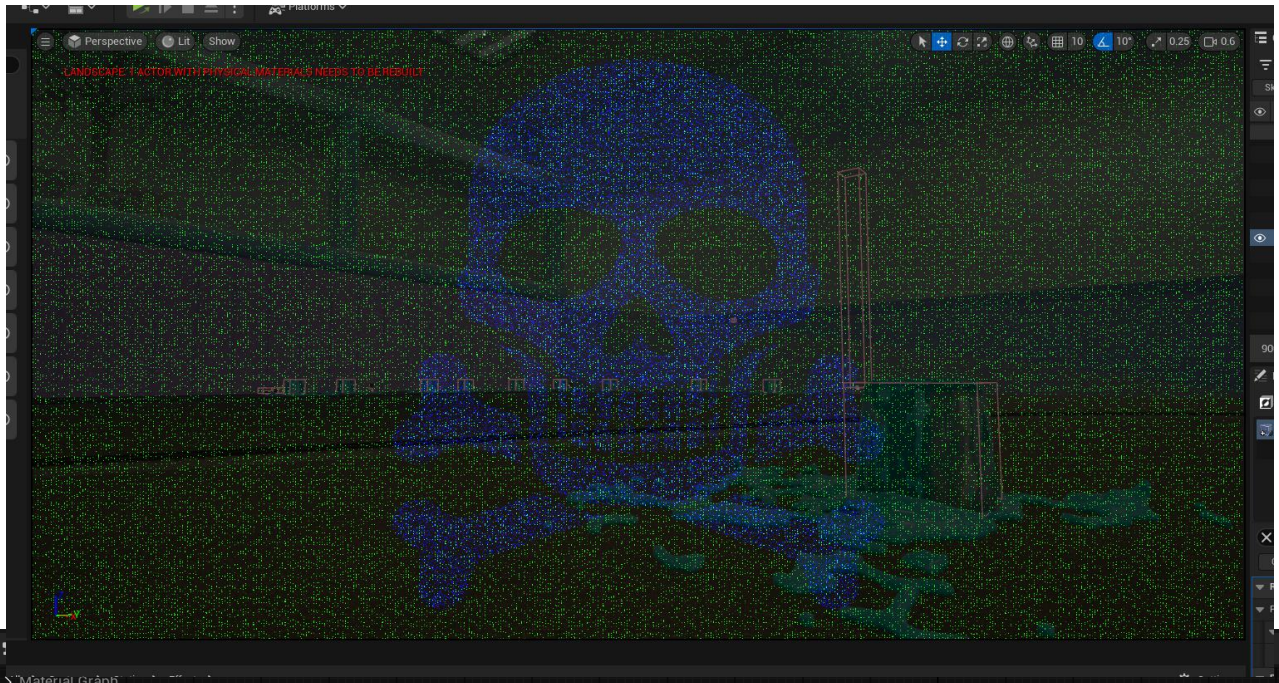


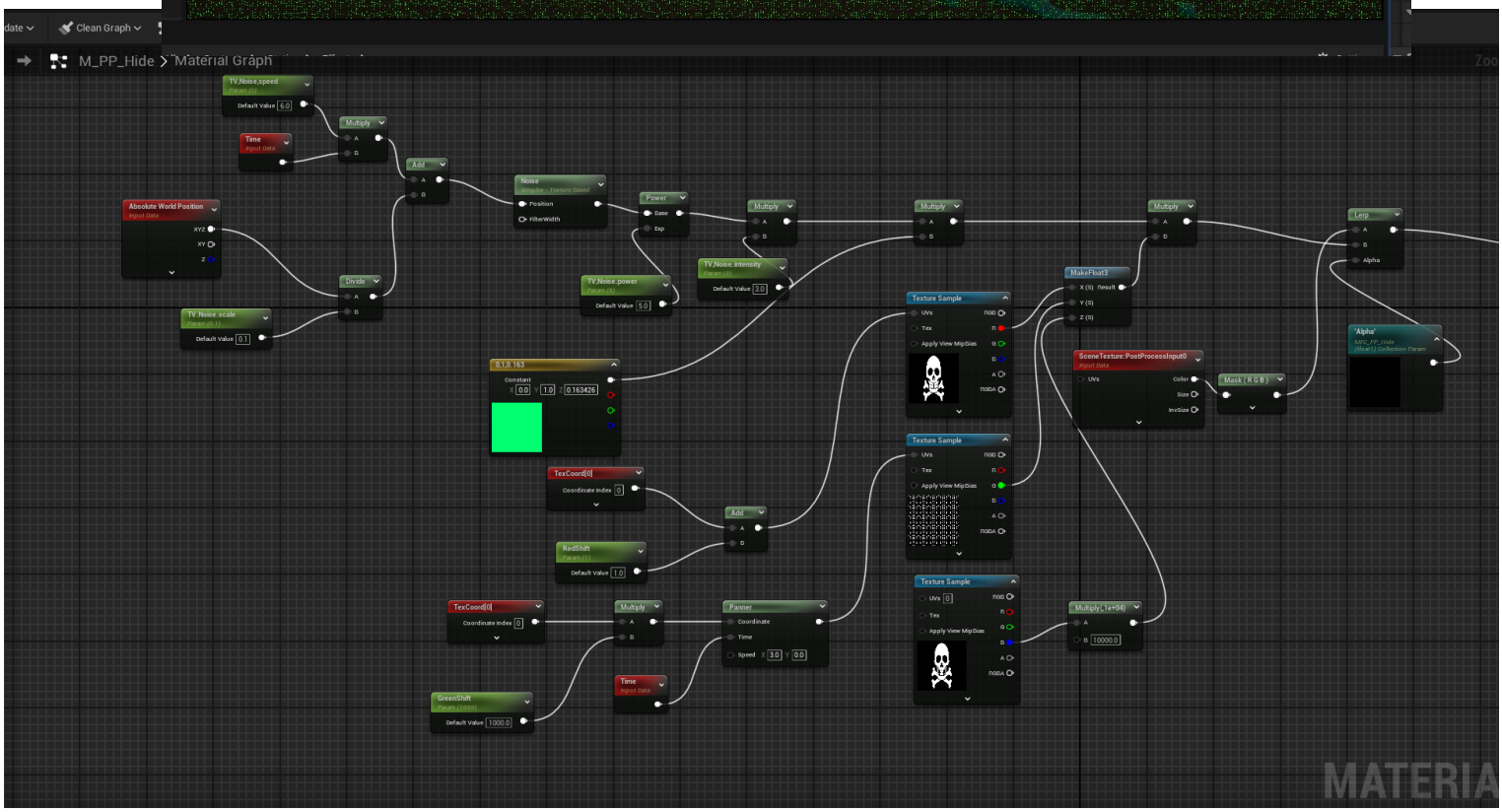
캐릭터의 CustomDepth pass를 활성화하여 캐릭터는 빨갛게 보이도록 구현했습니다.

또한 스텐실 버퍼를 사용하여 술래가 벽 뒤에 있어도 보이게 했습니다.

술래2 스킬

- **구현 과정**





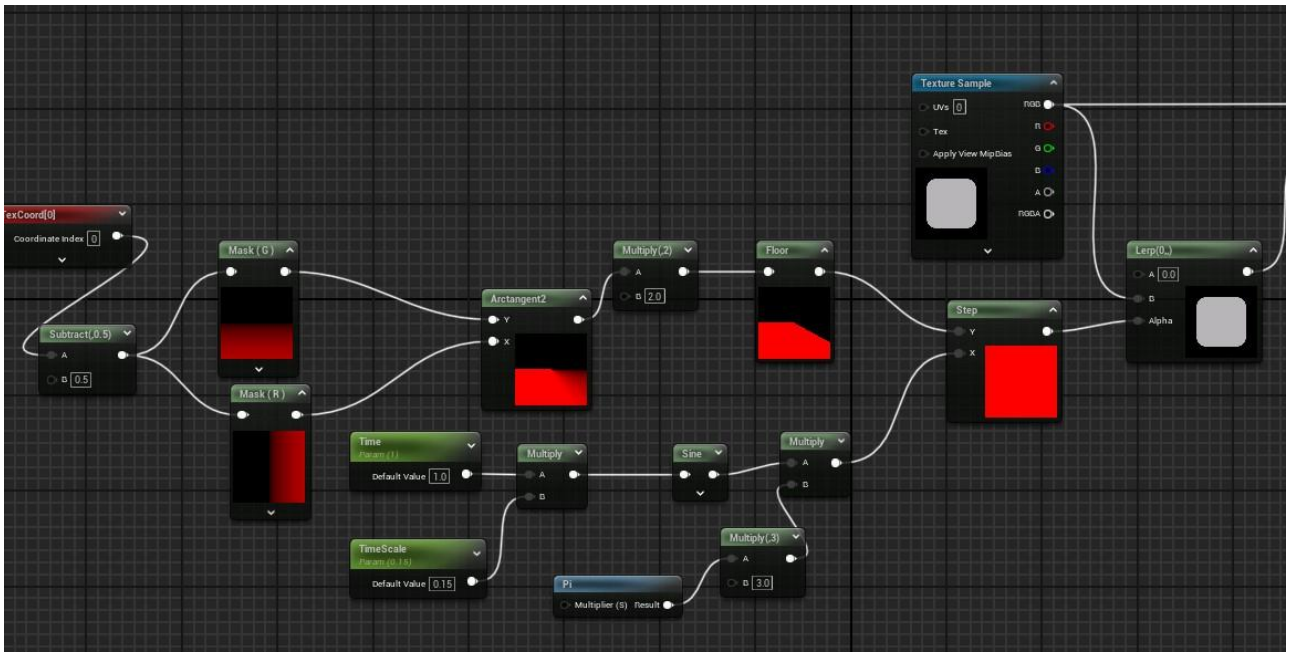
포스트 프로세싱 머티리얼을 사용하여 술래2 스킬을 구현했습니다.
TV의 노이즈 효과를 구현하기 위해서 노이즈 패턴을 생성해야 합니다.
따라서 Absolute World Position 노드를 활용하여 월드 좌표를 기반으로 노이즈 패턴을 생성했습니다.
이를 통해 카메라 움직임과 관계없이 패턴이 월드 공간에 고정되도록 구현하였습니다

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

- **구현 과정**



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.

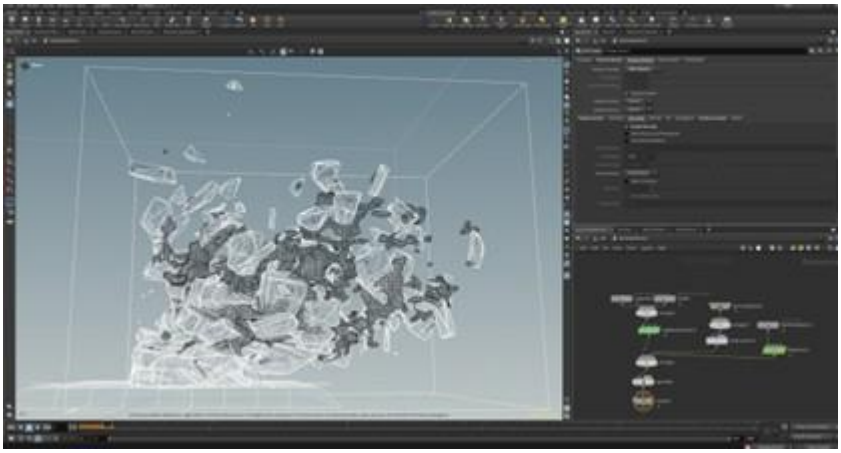
젤리 폭발 이펙트

- **구현 과정**

1. 첫 번째 접근: Niagara fluid와 chaos destruction을 사용

- 문제점: 젤리가 아닌 사탕이 폭발하는 느낌이었습니다. 그리고 niagara fluid 파티클이 생성 될 때 일시적으로 프레임 드랍 발생했습니다.

2. 두 번째 접근 : Houdini 시뮬레이션



강체 없이 유체만 사용했을 때 중력에 의해 젤리 모양을 유지하지 못했고 충돌 지점에서만 splash 가 발생했습니다.

따라서 rigid body 시뮬레이션과 fluid 시뮬레이션을 결합하여 액체를 담은 젤리 형태의 강체와 bullet 오브젝트를

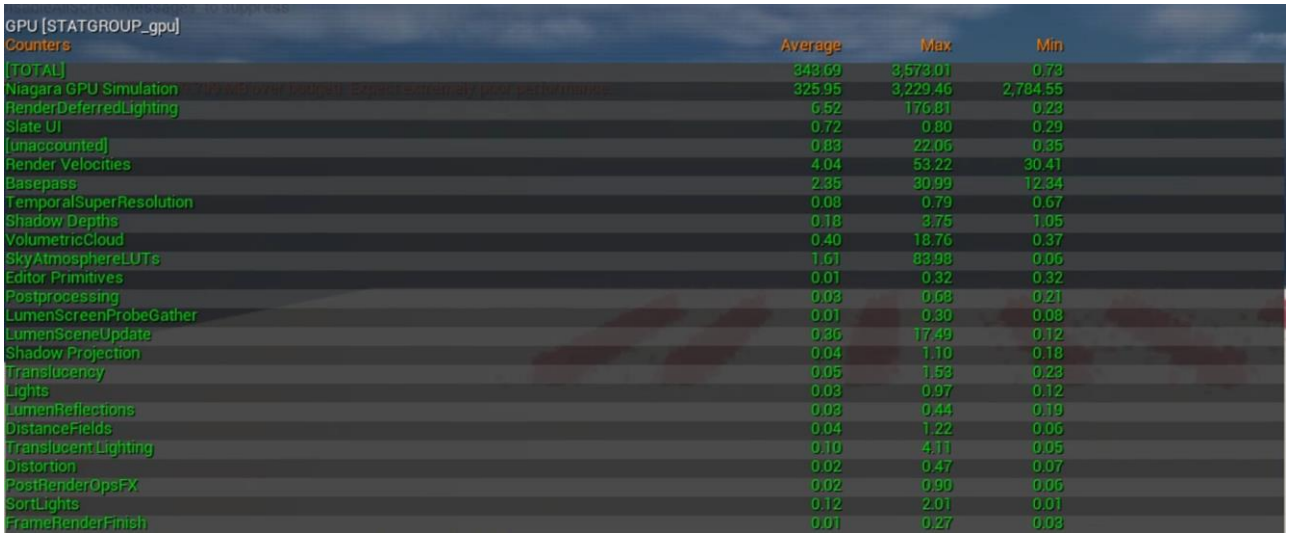
충돌 시뮬레이션했습니다. 그리고 flip 시뮬레이션 결과를 텍스처로 bake하여 저장했습니다.

언리얼 엔진에 사용하기 위해 alembic 파일을 사용해봤습니다,

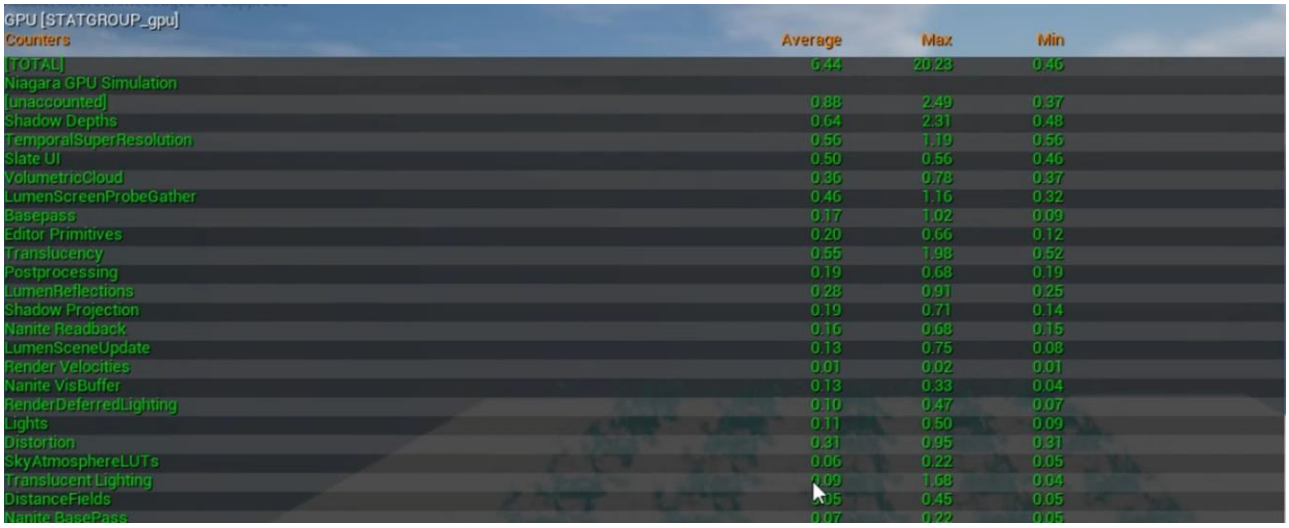
그러나 1 frame을 그리는 시간이 평균 59ms가 발생했고,

Vertex Animation을 사용하여 1 frame을 렌더링 평균 시간은 6.44ms로 최적화 했습니다.

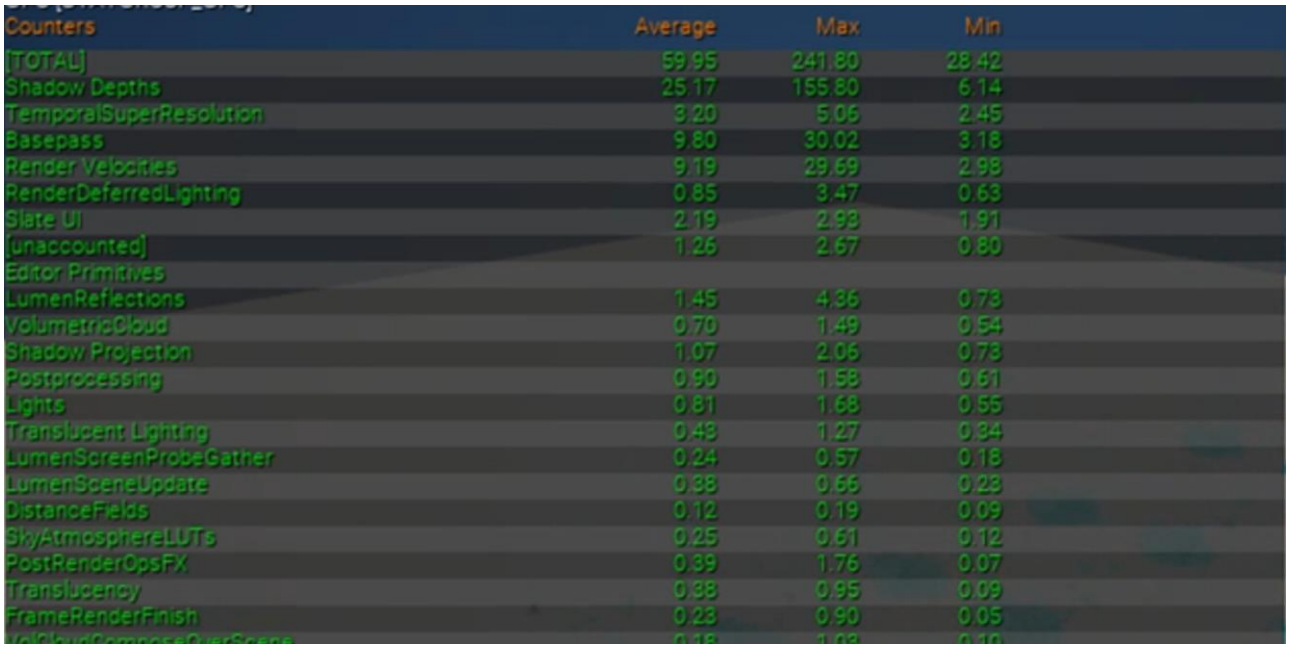
GPU 프로파일 결과



*niagara fluid + chaos destruction : 30*개 폭발



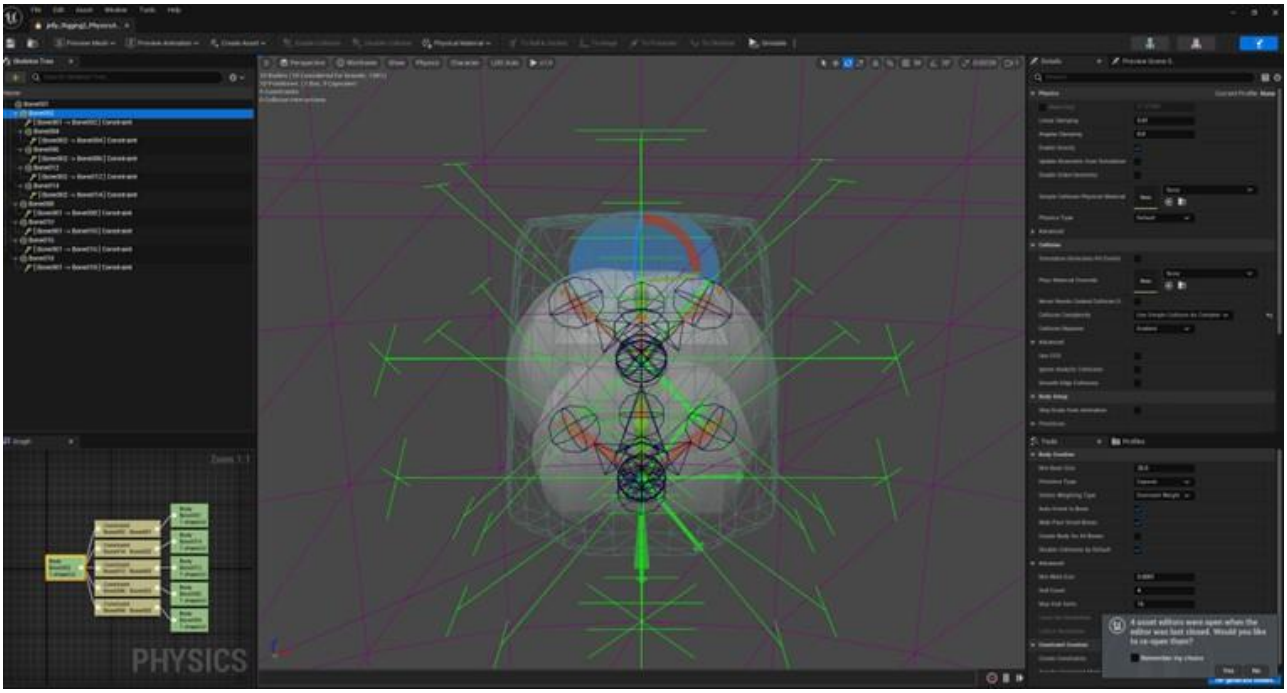
*Vertex Animation : 30*개 폭발



alembic 파일 *: 30*개 폭발

Jiggle Physics

- 구현 과정



Physics Asset에 Shape과 constraint 설정

- **문제점 :** 젤리와 플레이어가 충돌하면 젤리mesh는 Simulation Physics가 활성화가 되어 있어서 캐릭터와 충돌시 젤리가 멀리 날라가는 문제가 발생했습니다.
- **해결방안:** 젤리 하단 본 위치를 월드 좌표에 고정해서 해결했습니다. 이를 위해 젤리 액터 블루프린트에서 PhysicsConstraint 컴포넌트를 추가하여 젤리의 하단 bone을 월드에 고정해줬습니다.