

Real-time particle-based snow simulation on the GPU

게임공학 1조

강수한, 박경숙, 안해영, 원종서, 윤태웅

Index

- 01** 최종 결과물 소개
- 02** 구현한 기술
논문 기술 + 추가로 적용한 기술
- 03** 시행착오 과정
- 04** 중간발표 피드백 대응

01 최종 결과물

구현한 기술과 최종 결과물

1. UE4 + CUDA를 이용한 눈 시뮬레이션 기술
2. 캐릭터 충돌
3. Particle들을 시각화 하는
Screen Space Rendering 기술
4. 구역별 Simulation으로 최적화하는 기술



02 구현한 기술

Cohesion force, compression 커널 함수 + FRNN search 기술 개발

cohesion force와 compression은 눈 시뮬레이션의 핵심
논문에 서술된 알고리즘을 CUDA를 이용하여 코드화 하는데 성공

1. Cohesion Force

$$\vec{F}_n = \begin{cases} -\frac{E_j r_j + E_k r_k}{2} \delta \vec{n} \\ 0 \text{ and cohesion is broken,} \end{cases}$$

$$\text{if } -\frac{E_j r_j + E_k r_k}{2} \delta < 4 \frac{\sigma_{nj} r_j^2 + \sigma_{nk} r_k^2}{2} \\ \text{otherwise}$$

```
__global__  
void getCohesionForce(  
    float3* dev_pos,  
    float3* dev_vel,  
    float3* dev_force,  
    ParticleData* dev_pData,  
    const int* dev_PID,  
    const int* dev_BinID,  
    const int* dev_PBM,  
    float3* debug,  
    const unsigned int size,  
    const ParamSet Param  
) {
```

2. Compression

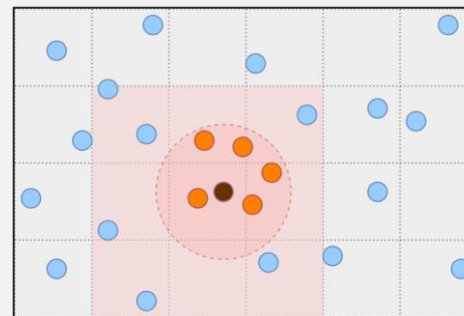
$$\vec{D}(\rho_i) = \vec{F}_{minW} + \left(\frac{e^{\frac{\rho_i}{100}} - 1 - 0.000335}{2980.96} \right) \vec{F}_{maxW}$$

$$d \leftarrow \begin{cases} d - k_q p_c, & \text{if } \vec{F}_c > \vec{D}(\rho_i) \\ d & \text{otherwise} \end{cases}$$

```
__global__  
void compression(  
    ParticleData* dev_pData,  
    float* dev_radius,  
    float3* debug,  
    const unsigned int size,  
    const ParamSet Param  
) {
```

3. FRNN search

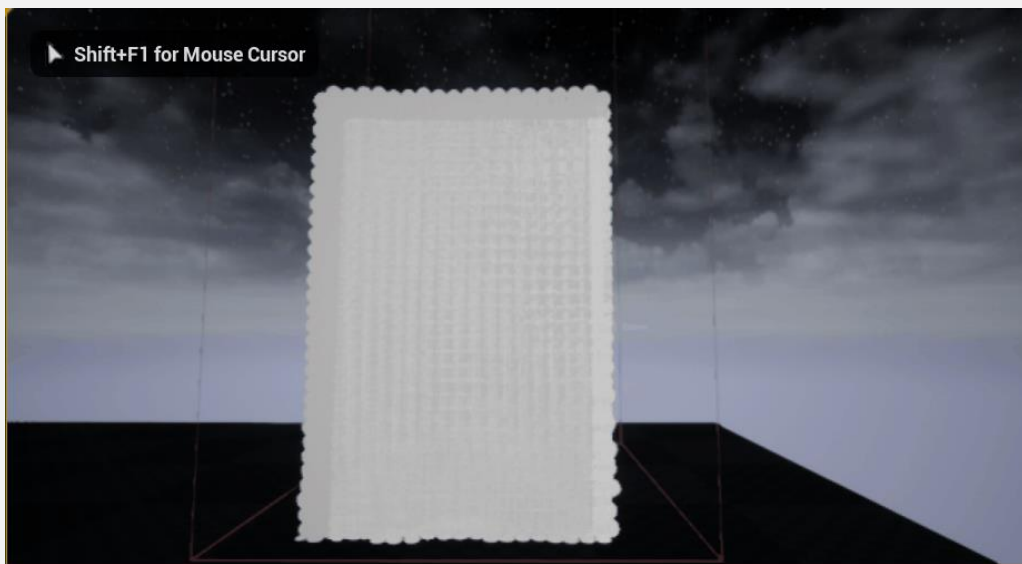
particle들의 응집력, 충돌을 계산할
때 주변 particle들을 $O(nk)$ 의 시간
복잡도로 찾아내는 기술



@ Improved GPU near neighbours performance for
multi-agent simulations | Robert Chisholm, Steve
Maddock, Paul Richmond

02 구현한 기술

Cohesion force, compression 커널 함수 + FRNN search 기술 개발



▲ 눈의 자연붕괴



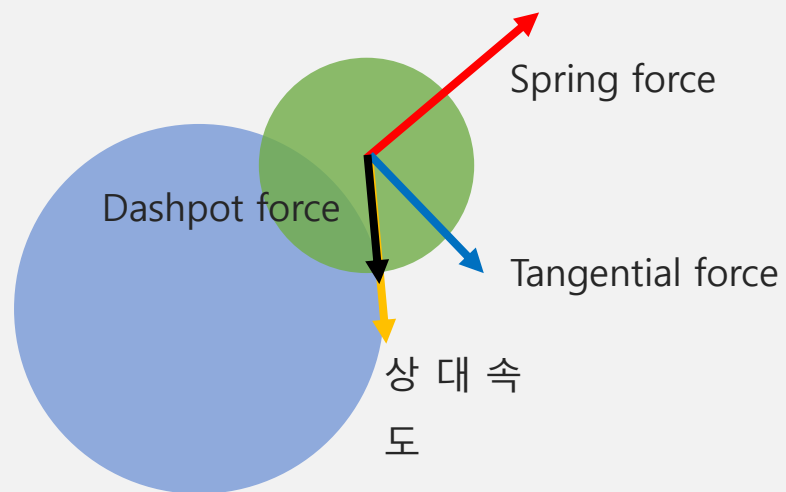
▲ 하늘에서 떨어져서 지면의 눈과 자연스럽게 합쳐지는 눈덩이

02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리

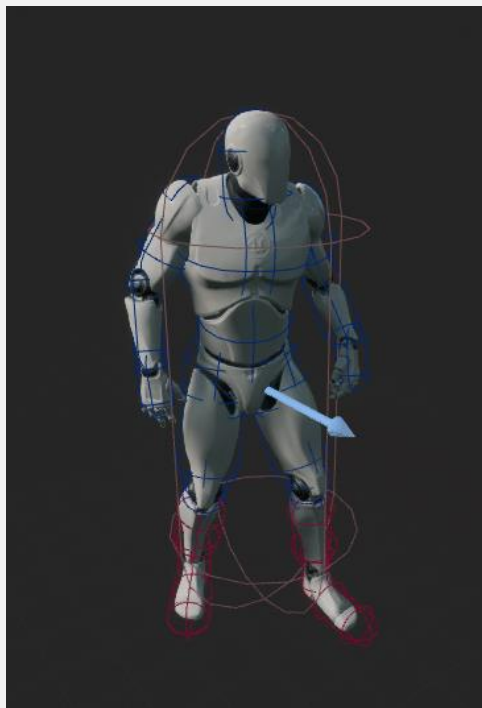
Dem 충돌 모델

Spring Force, Dashpot Force, Tangential Force를
이용한 구형 충돌체와 Particle간의 충돌 모델

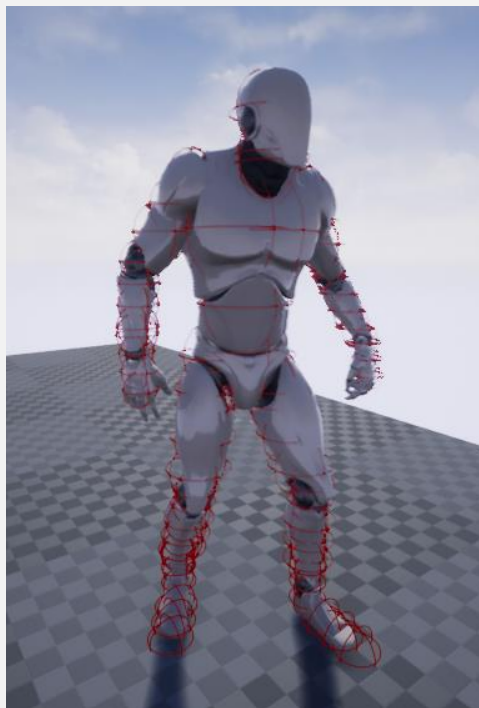


02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리



▲ Collision Component



▲ Sampling Result

평범한 캐릭터 Mesh에 Dem 충돌 모델을
적용하기 위해서는 캐릭터에 부착된
Collision Component를 작은 구체로
Sampling하는 과정이 필요함

02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리

Unreal의 collision Component는 Box, Sphere, Capsule 3가지 type으로 나뉨

각각의 Collision Type에 대해 Collision의 표면을 구 형태로 빈틈없이 채우는 코드 개발

내부의 구체는 충돌에 관여하지 않기 때문에 Collision의 내부는 Sampling하지 않음

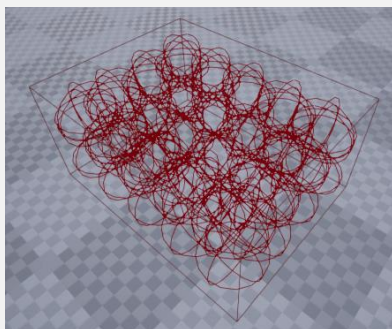
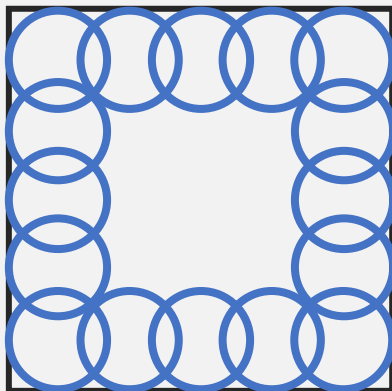
Sampling한 구체의 Radius가 작을 수록 저장해야 하는 구체의 수가 많기 때문에 최대한 큰 크기의 구체로 구성하여 구체의 수를 최소화 함

```
void UCollisionSamplingManager::CollisionSampling(
    TArray<UPrimitiveComponent*> collisionComponents
)
{
    FCollisionShape collisionShape;
    ColliderArr.Empty();
    for (UPrimitiveComponent* collider : collisionComponents)
    {
        collisionShape = collider->GetCollisionShape();
        if (collisionShape.IsBox())
        {
            BoxCollisionSampling(collider);
        }
        else if (collisionShape.IsSphere())
        {
            SphereCollisionSampling(collider);
        }
        else if (collisionShape.IsCapsule())
        {
            CapsuleCollisionSampling(collider);
        }
    }
}
```

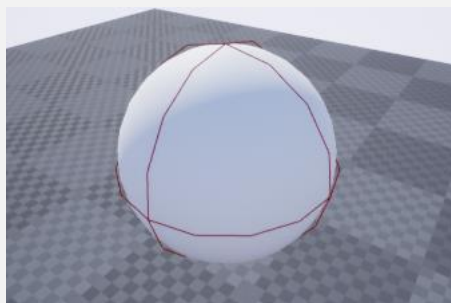
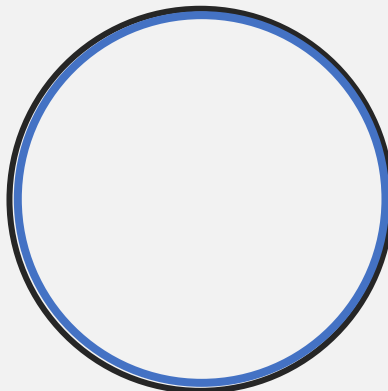

02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리

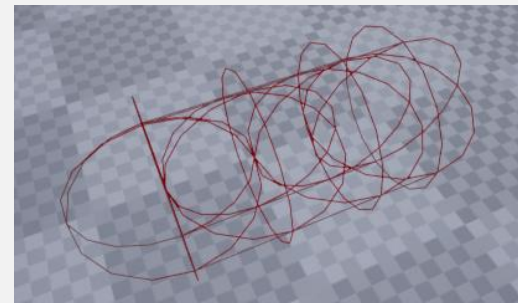
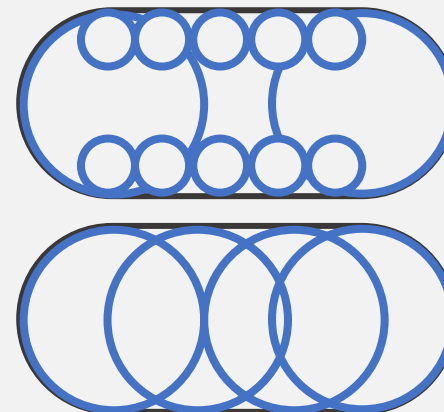
1. Box Collision



2. Sphere Collision



3. Capsule Collision



02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리

GPU를 이용한 충돌 처리 과정

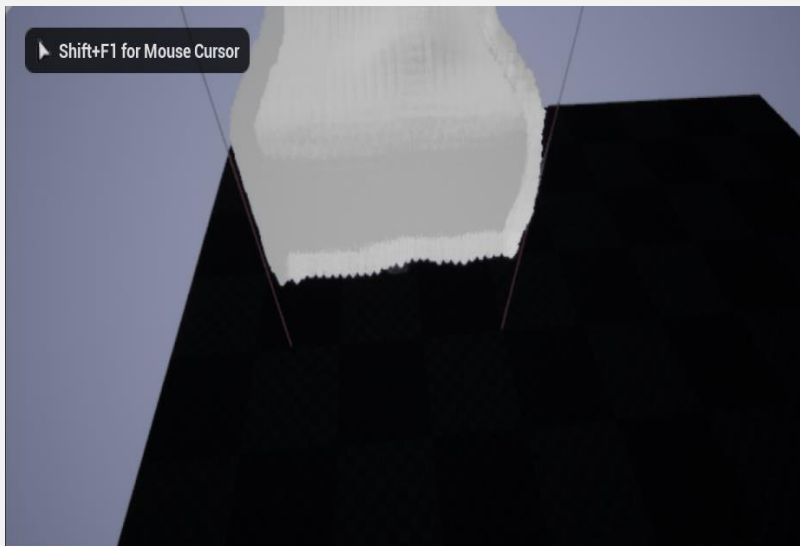
1. 매 프레임 Sampling한 충돌체들의 위치정보를 GPU로 메모리 복사
2. 충돌 계산을 위해 필요한 충돌체의 속도 정보는 이전 프레임의 위치, 현재 프레임의 위치를 이용해 속도를 병렬로 계산하는 Kernel 함수로 계산
3. GPU메모리 상에 존재하는 Particle 정보, 충돌체 정보를 바탕으로 충돌 Kernel함수를 particle마다 병렬 실행하여 Particle이 충돌의 결과로 받는 힘을 계산
4. 합력에 의한 각 Particle의 새로운 위치 계산

```
__global__  
void calcColliderVelocity(  
    float3* dev_colPos,  
    float3* dev_preColPos,  
    Collider* dev_colliderNotInteracting,  
    ParamSet Param,  
    float dt){
```

```
__global__  
void collide(float3* dev_pos,  
    float3* dev_vel,  
    float3* dev_force,  
    Collider* dev_colliderNotInteracting,  
    Collider* dev_colliderInteracting,  
    float3* dev_colPos,  
    float3* dev_colInterPos,  
    ParticleData* dev_pData,  
    const unsigned int size,  
    ParamSet Param)  
{
```

02 구현한 기술

Collision Sampling 기술을 이용한 충돌처리



▲ 떨어지는 눈과 캐릭터의 상호작용



▲ 걷는 캐릭터와 눈의 상호작용

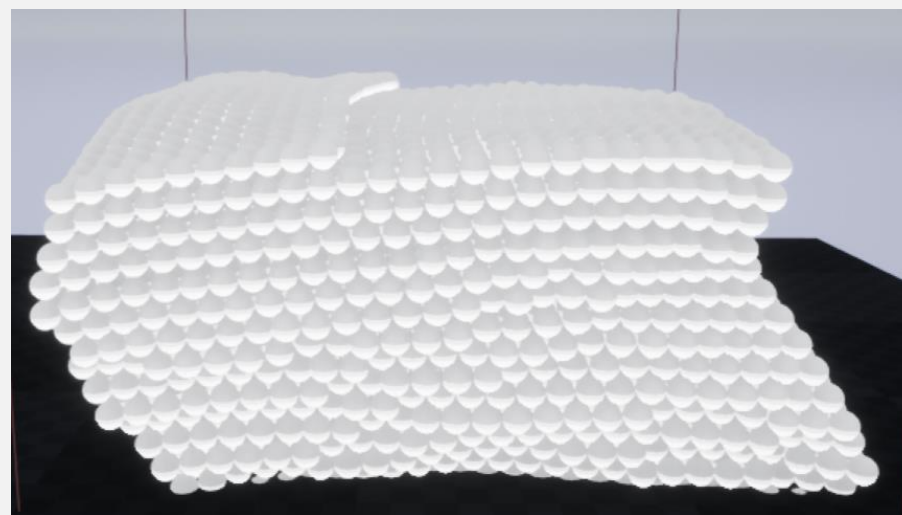
02 구현한 기술

Screen Space Rendering

Particle기반 물리 시뮬레이션은 결과로 나오는 위치정보들을 적절하게 시각화 하는 것이 중요함

Marching Cube Algorithm을 통한 시각화를 계획했지만, 중간 발표 때 Marching Cube Algorithm은 real time에 실행하기에는 적절하지 못하다는 피드백을 받음

따라서, Marching Cube Algorithm만큼 디테일을 살리면서 좀 더 좋은 성능을 보여주는 Screen Space Rendering Algorithm을 적용하기로 결정



▲ Particle을 Sphere Mesh로 렌더링한 경우

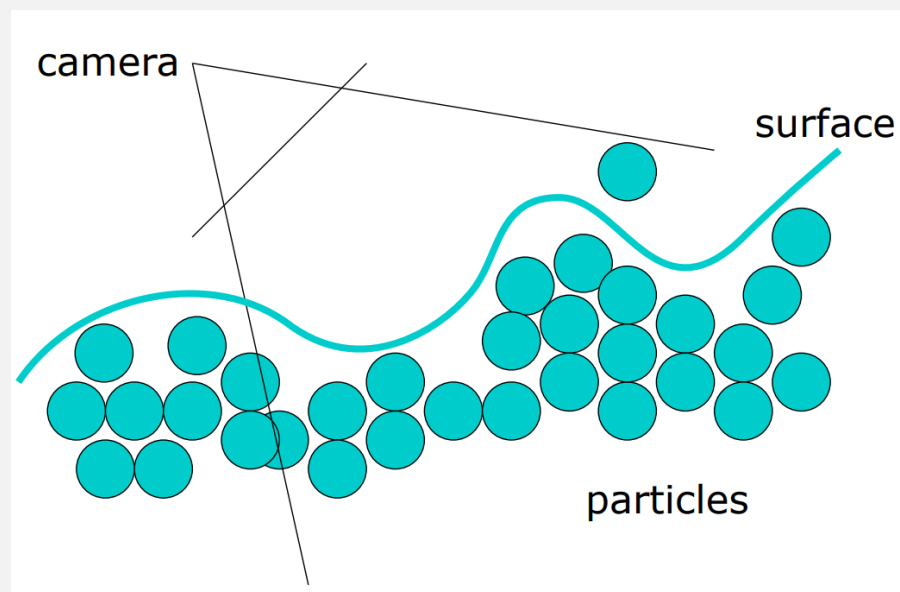
02 구현한 기술

Screen Space Rendering

Screen Space Rendering?

Camera를 기준으로 Particle들을 관측했을 때,
Particle들이 구성하는 surface를 계산하고
Rendering하는 알고리즘

Screen에 보이는 Surface만 계산하면 되기에 전체
Particle을 Mesh화 하는 Marching Cube보다 연산
복잡도가 적음

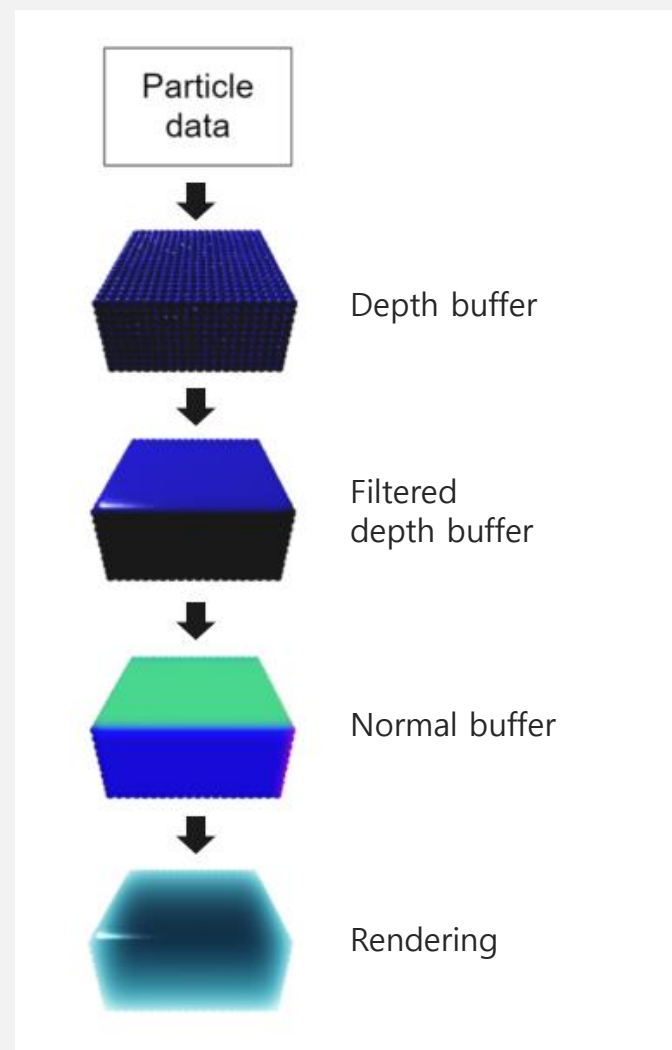


02 구현한 기술

Screen Space Rendering

Rendering 과정

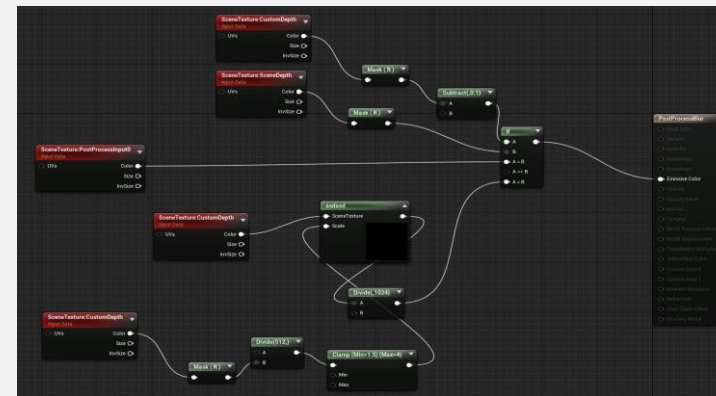
1. Screen기준 Particle들의 Depth image 생성
2. Depth image에 Gaussian blur를 적용
3. Gaussian blur가 적용된 Depth image의 x, y 변화량을 기반으로 Pixel Normal을 계산
4. Normal에 Lighting효과 적용하여 렌더링



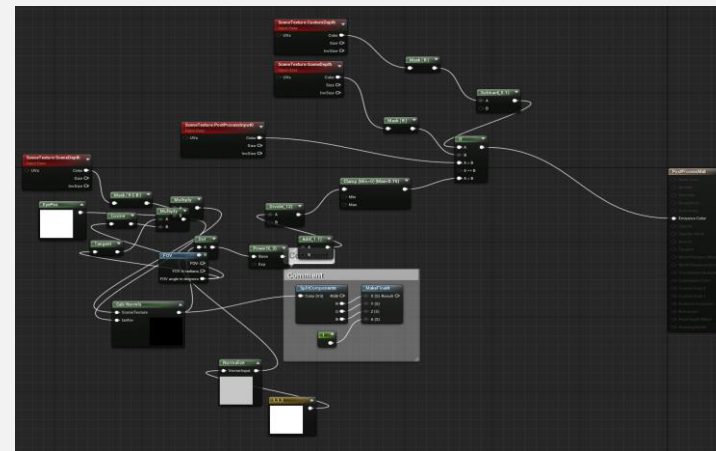
02 구현한 기술

Screen Space Rendering

엔리얼 엔진이 기본으로 제공하는 Post Process Volume을
이용해 Particle들에 한해서 Screen Space Rendering을 하도
록 구현함



▲ Calculate Gaussian Blur



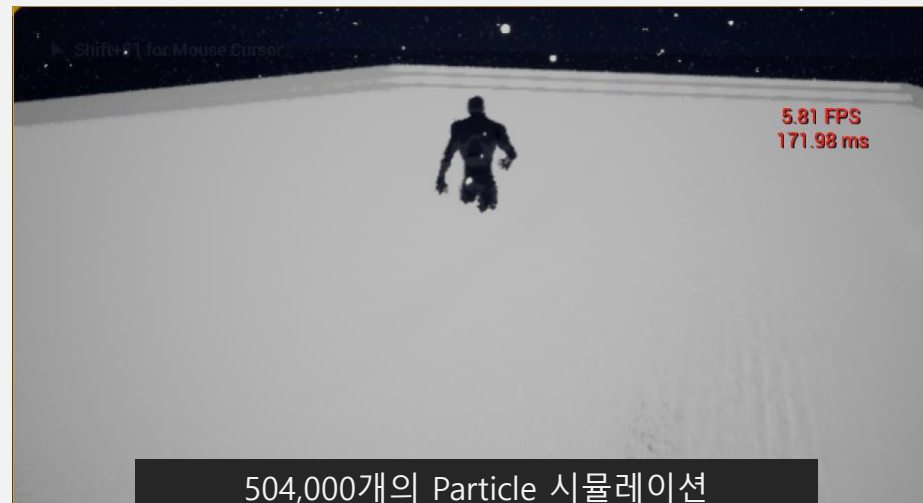
- ▲ Calculate Normal From Depth + Diffuse Shading

02 구현한 기술

구역별 Simulation을 통한 최적화

중간 발표 때 Particle기반 물리 시뮬레이션은 Particle개수가 많아 질수록 연산 복잡도가 크게 증가하기에 이를 해결하겠다고 말한 바가 있음

플레이어가 위치한 지역만 Particle 물리 시뮬레이션을 적용시키고 중요하지 않은 나머지 영역은 상대적으로 연산 복잡도가 낮은 방식으로 표현하는 방식을 사용함
대표적인 게임 최적화 기법인 LOD와 비슷한 아이디어



504,000개의 Particle 시뮬레이션
+ Screen Space Rendering = 4~5fps

02 구현한 기술

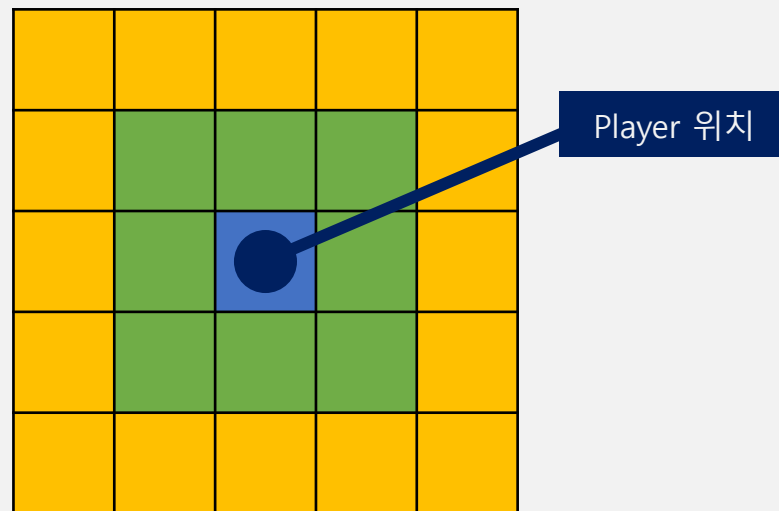
구역별 Simulation을 통한 최적화



구역 별 시뮬레이션 최적화 과정

1. 맵을 로드할 때, 구역별로 나눠서 Particle정보를 GPU메모리에 할당

2. 플레이어가 속한 구역, 플레이어와 1칸 떨어진 구역, 플레이어와 2칸 이상 떨어진 구역으로 분류하여 각각 다른 방식 적용

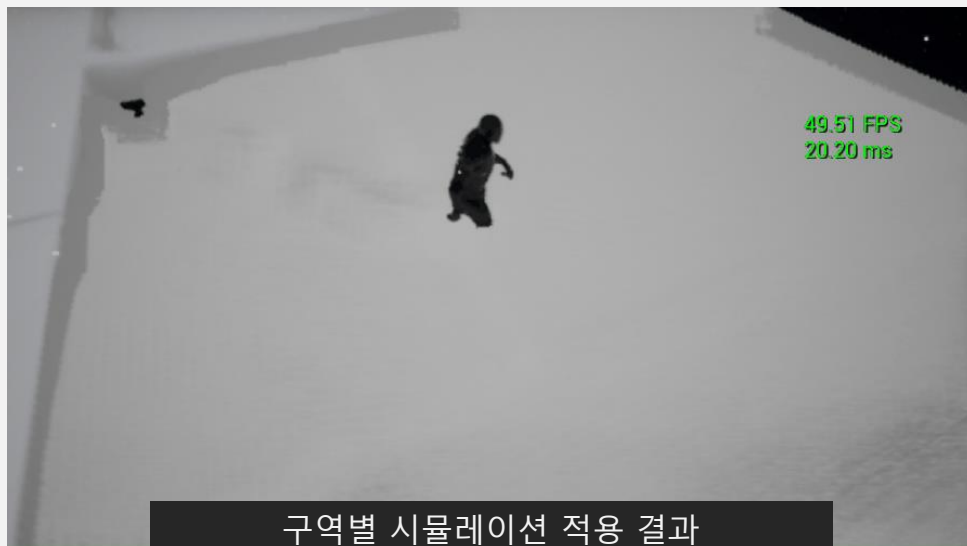
눈 Particle은 건드리지 않으면 정적인 동작을 하기 때문에 이러한 방식이 적용될 수 있음



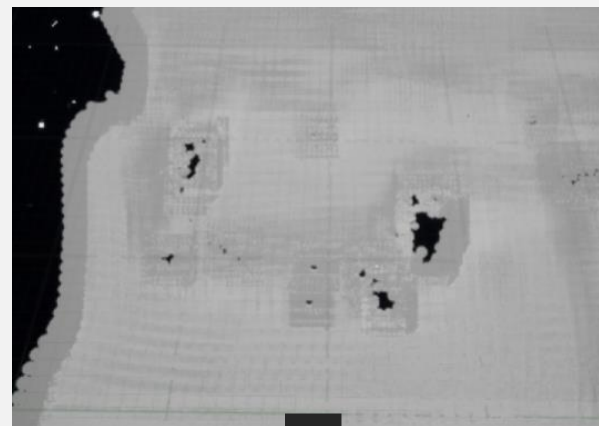
-  시뮬레이션 O + Screen Space Rendering
-  시뮬레이션 X + Screen Space Rendering
-  시뮬레이션 X + Height Map 기반, 고정된 Mesh Rendering

02 구현한 기술들과 노력들

구역별 Simulation을 통한 최적화



구역별 시뮬레이션 적용 결과
5fps -> 52fps, 10배 성능 향상



Height Map 기반, Voxel Mesh화

03 시행착오 과정

Delta Time 문제

Particle들을 매 프레임 각각의 합력을 구한 후 가속도, 속도, 위치 순으로 적분 계산을 함

이때 사용되는 시간 단위인 DeltaTime에 언리얼 Tick함수의 DeltaTime을 그대로 적용했더니 눈이 폭발하는 것처럼 보이는 문제 발생

추후에 논문을 다시 정독 후 time step을 0.001로 설정했다는 문구를 발견 후 수정, 오류 해결

Time step을 0.001로 수정한 결과, 시뮬레이션이 슬로우 모션과 같이 보이는 문제가 발생, 따라서 매 프레임 적용해주는 병렬계산 과정을 한 프레임에 여러 번 적용해주는 방식으로 문제 해결



03 중간발표 피드백에 대한 대응

피드백 대응

리얼타임이 중요한 게임보다 영화 쪽에서 더 잘 사용할 수 있는 기술

GPU 연산을 최대한 활용하고 구역별로 시뮬레이션을 다르게 적용하는 알고리즘을 개발해서 실시간성을 최대한 확보함



03 중간발표 피드백에 대한 대응

피드백 대응

리얼타임이 중요한 게임보다 영화 쪽에서 더 잘 사용할 수 있는 기술

GPU 연산을 최대한 활용하고 구역별로 시뮬레이션을 다르게 적용하는 알고리즘을 개발해서 실시간성을 최대한 확보함

실제 Mesh화를 하는 것이 핵심 Overhead인데, 이 부분에 대한 고려가 미비

Screen Space Rendering 방식을 적용하여 Overhead를 효과적으로 해결함

감사합니다

게임공학 1조
강수한, 박경숙, 안해영, 원종서, 윤태웅