Fluid Simulation using SPH method

tspaghettit@gmail.com

김동현

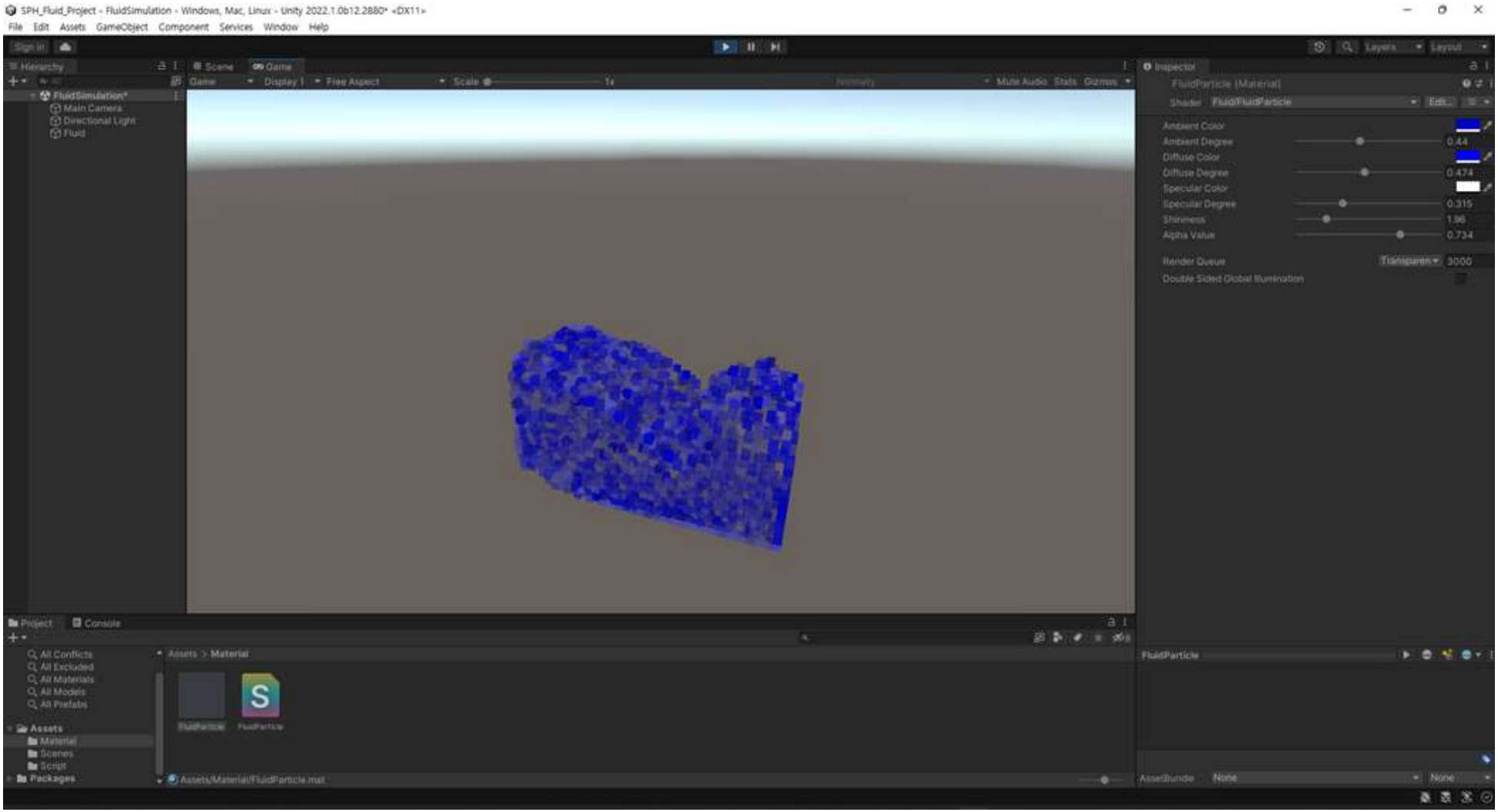
목차

1. 개요

- 3차원 공간에서 유체의 움직임을 구현한 프로젝트
- d’strict 사의 “WAVE” 작품을 보고 영감을 받음
- 엔진 자체 기능을 사용하지 않고, 그래픽스 프로그래밍을 통해 직접 구현하고자 했습니다

Public Media Art "WAVE" Full ver. by a'strict
https://www.youtube.com/watch?v=CZxKdgiisAU

실제 구현 내용 (https://youtu.be/w5LDP7CwzFk)



2. 이 프로젝트를 통해 얻고자 한 것

- 사실적인 그래픽을 만들어 보는 경험
- 컴퓨터 그래픽스에서 배운 지식을 직접 적용해 보며 개념들을 다시 공부
- 논문의 내용을 프로그램에 직접 적용해 보는 과정을 체험
- Unity 엔진에서 Shader 를 작성하는 방법에 대한 공부

3. 구현 내용과 구현 방법

A. 유체의 움직임

- Compute Shader 를 통해서, 유체를 이루는 각 Particle 의 정보 계산
- GPU Instancing 을 통해 수많은 Particle 을 한번에 렌더링

B. 유체의 색깔

- 유체의 바깥을 향하는 Normal Vector 를 이용해서 기본적인 블린 – 퐁 모델 적용
- Particle 이 유체의 경계에 가까운 정도에 따라서, 투명도를 조절

A. 유체의 움직임

- 유체의 움직임을 계산하기 위해서

  Matthias Müller, David Charypar and Markus Gross 의

  'Particle-Based Fluid Simulation for Interactive Applications'

  논문의 내용을 이용

  - https://matthias-research.github.io/pages/publications/sca03.pdf
  - 나비에-스토크스 방정식을 particle-based approach 중 하나인 SPH 방법을 이용하여 계산

- 수많은 particle 들의 값을 빠르게 계산하기 위해서

  Compute Shader 를 이용

A. 유체의 움직임

- SPH method
  - 한 particle 의 값이 다른 particle 들의 값에 의해 결정된다
  - 서로 떨어져 있는 거리에 따라서 가중치가 달라지는데, 이는 smoothing kernel 함수 값에 의해 결정
- SPH 방법을 통해 각 particle 의 밀도, 압력, 힘 등을 계산
- Compute Shader 계산 시, 한 thread 당 하나의 particle 을 할당
  - 병렬 계산을 실행
  - Particle 의 수가 많아져도 빠른 계산이 가능

```
//Fluid 를 이루는 Particle 각각이 가지는 데이터
참조 4개
private struct Particle
{
    // 위치
    public Vector3 position;
    // 속도
    public Vector3 velocity;
    //particle 에 가해지는 힘 => 최소 3가지 요소로 구성되어 있다
    //압력 + 점성 + 표면장력
    public Vector3 force;

    // 밀도
    public float density;
    // 압력
    public float pressure;

    //particle surface normal, 표면장력
    public Vector3 surfNormal;


    참조 1개
    public Particle(Vector3 pos)
    {
        position = pos;
        velocity = Vector3.zero;
        force = Vector3.zero;
        density = 0.0f;
        pressure = 0.0f;
        surfNormal = Vector3.zero;
    }

}
```

A–1. C# 스크립트 파일

- 유체를 구성하는 Particle 구조체

  밀도, 압력 등 유체의 값을 가지고 있으며

  유체 내부를 향하는 법선 벡터,
  Particle 에 가해진 알짜힘,
  현재 속도, 현재 위치를 저장하고 있다.

  Particle 구조체를 버퍼를 통해 GPU 로 넘겨서 계산을 진행한다

```
// 2. 매 프레임마다 처리하는 일
Unity 메시지 | 참조 0개
void Update()
{
    //Wave 진행을 위해, 실행 후 지난 시간을 compute shader 에 넣어주어야 한다
    //float 의 표현 범위가 10^31 까지 가능하기 때문에, overflow 는 걱정하지 않아도 된다
    totalTime += deltaTime;
    shader.SetFloat("Time", totalTime);

    /*
    전체 프로세스

       매 프레임, 시간마다 각 particle 의 밀도가 변하므로 밀도를 계산해 주어야 한다
       이후 밀도로 인한 각 particle 의 압력을 계산해주어야 한다
       그 다음에는 각 particle 에 가해지는 힘을 구해야 한다

       밀도 -> 압력 -> Forces

       알짜힘을 구했다면, 힘에 의한 가속도를 구하여서 새로운 속도를 구하고 새로운 속도로 이동한 위치를 구한다
       이후 해당 위치가 정해진 범위를 벗어나지 않게 하면 된다

       가속도 구하고 속도 갱신 위치 갱신 -> 범위 확인
    */
    // 2-1. 각 particle 들의 밀도와 압력을 먼저 계산하고
    shader.Dispatch(kernelComputeDensityPressure, groupSize, 1, 1);

    // 2-2. 각 particle 에 가해지는 압력에 의한 힘 + 점성에 의한 힘 + Wave 에 의한 힘 계산 + 중력
    shader.Dispatch(kernelComputeForces, groupSize, 1, 1);

    // 2-2-5. surface force 를 구하는 커널
    shader.Dispatch(kernelComputeSurfaceForce, groupSize, 1, 1);


    // 마우스 입력을 확인한다
        if(Input.GetMouseButtonDown(0))...

        if(Input.GetMouseButtonUp(0))...
```

- 스크립트 내 Update 함수

  매 프레임마다 자동으로 호출되는 함수

  Compute Shader 내에 정의된 커널 함수들을 호출

  밀도와 압력을 계산,
  알짜힘 계산,
  이동 계산 등 일련의 과정을 진행

```
//모든 힘에 대한 처리를 끝마치고 나서 ~
// 2-3. 가속도를 구하여서 새로운 속도, 위치를 구한다
shader.Dispatch(kernelMakeMove, groupSize, 1, 1);

// 2-4. 정해진 범위를 벗어나지 않았는 지 확인한다
shader.Dispatch(kernelCheckLimit, groupSize, 1, 1);


//마지막으로 particle 들을 그려낸다 => GPU Instancing 이용
Graphics.DrawMeshInstancedIndirect(mesh, 0, material, bounds, argsBuffer);
```

이후, GPU Instancing 을 통해
수많은 Particle 들을 한번에 렌더링

Compute Shader 에서 계산한 값은
Buffer 를 통해서 바로
정점 셰이더에서 접근하여 처리한다

```
/*
    2-1. particle 의 상태 갱신하는 커널 함수

    가장 먼저 밀도를 계산하고
    얻은 밀도 값으로 압력 값을 계산한다
*/
[numthreads(ThreadCount, 1, 1)]
//SV_DispatchThreadID => 전체 그룹 관점에서, 해당 스레드 id 리턴
void ComputeDensityPressure (uint3 id : SV_DispatchThreadID)
{
    //일단 스레드가 담당하는 현재 particle 을 가져온다
    Particle curParticle = particles[id.x];

    //particle 의 밀도는 매 프레임마다 구해주어야 한다
    curParticle.density = 0; //초기화


    //밀도 -> 압력 순서로 계산
    /*
        SPH 기법 = 전체 다른 particle 의 값과 위치를 이용해서 보간해서 => 현재 값을 구한다
        그러니까, 전체 particles 에 대해서 조사를 진행한다
    */
    for(int j = 0; j < particleCount; j++)
    {
        //자신과 다른 particle 간의 위치 차이를 구한다
        //논문에서 주어진 대로, 상대방에서 나를 향하는 벡터를 사용한다
        float3 relativePos = curParticle.position - particles[j].position;

        float rSquare = dot(relativePos, relativePos);

        //서로 떨어져 있는 거리가 h 이하인 경우에만 영향을 끼친다 => h = smooth radius
        //이때 어차피 r^2 을 사용하므로, 거리 비교도 제곱 값을 이용한다
        //제곱하고자 하는 값이 벡터니까... 거리를 구하고 제곱하는 거 보다,
        //자체 내적을 이용하는 게 더 편하다 -> 그래서 내적 사용
        if(rSquare < hSquare)
        {
            //SPH 방법을 이용하여 밀도를 구한다
            //W_Poly6 Smooth Kernel
            curParticle.density += particleMass * W_Poly6(rSquare);
        }

    }
    //혹시 모르니까, 밀도 값은 음수 값을 가질 수 없다
    curParticle.density = max(curParticle.density, 0.001f);
```

A–2. Compute Shader

- SPH 방법으로
  Compute Shader 에서 계산하는 방법

1. SV_DispatchThreadID 를 이용

   전체 스레드 그룹 중 현재 커널 함수를 실행하는 스레드 id 에 접근

2. 전체 Particle 들에 대해서 접근

   다른 Particle 이 영향을 주는 거리 내에 존재하는 지 파악
   영향을 주는 거리보다 멀리 있는 Particle 에 대해서는 계산하지 않는다

```
    // 새로 구한 밀도를 바탕으로, particle 이 가지는 압력을 구한다
    // 논문에 의해서, 수정된 이상기체 방정식을 사용하도록 한다
    curParticle.pressure = gasCoeffi * max(curParticle.density - restDensity, 0.0f);

    /*
        다른 모든 스레드들이 공유 메모리에 접근하는 것을 기다리게 하면서, 동기화를 하고 싶었지만?
        하나의 그룹 내의 스레드들만 동기화가 된다
        즉, 어차피 전체 그룹 내 전체 스레드의 동기화는 불가능
        그냥 하지말자

        GroupMemoryBarrierWithGroupSync();
    */

    // 계산한 particle 정보를 버퍼에 저장한다
    particles[id.x] = curParticle;
}
```

3. 다른 Particle 이 영향을 주는 영역내에 존재한다면, 구하려는 값을 계산

    위의 예에서는 압력만 계산하고 있지만,
    압력에 의한 힘, 점성에 의한 힘 등 유체 Particle 간의 힘을 구하는 경우
    적절한 Smoothing Kernel 값을 가져와서, 거리에 따른 가중치를 계산한다

4. 계산된 값을 버퍼에 저장

```
// 2-4. Particle 이 정해진 영역을 벗어나는 지 확인
[numthreads(ThreadCount, 1, 1)]
void CheckLimit (uint3 id : SV_DispatchThreadID)
{
    // 동작을 확인하고자 하는 particle
    Particle particle = particles[id.x];

    // 각 경계면에서 내부를 향하는, 노멀벡터를 설정한다
    float3 norVec = float3(0, 0, 0);

    // 경계로부터 떨어진 거리
    float overLength = 0;



    //벗어난 경계면의 종류에 따라 내부로 향해야 하는, 노멀 벡터를 구해준다
    if(particle.position.x < -limitRange.x)      norVec.x += -limitRange.x - particle.position.x;
    else if(particle.position.x > limitRange.x)  norVec.x += limitRange.x - particle.position.x;

    if(particle.position.y < -limitRange.y)      norVec.y += -limitRange.y - particle.position.y;
    else if(particle.position.y > limitRange.y)  norVec.y += (limitRange.y) - particle.position.y;

    if(particle.position.z < -limitRange.z)      norVec.z += -limitRange.z - particle.position.z;
    else if(particle.position.z > limitRange.z)  norVec.z += limitRange.z - particle.position.z;



    //정해진 영역으로 들어가는 노멀벡터가 0벡터 이라면, 범위를 벗어나지 않은 거니까, 그냥 끝낸다
    if(dot(norVec, norVec) == 0) return;

    // 아니라면, 경계로부터 떨어진 거리를 구하고 노멀벡터를 정규화한다
    else {
        // norVec 의 길이가 경계로부터 떨어진 거리다
        overLength = length(norVec);

        // 정규화
        norVec = normalize(norVec);
    }
```

- 이동한 Particle 이 지정된 영역을 벗어나지 않게 조정하는 커널 함수

1. 현재 스레드가 담당하는 Particle 에 접근

2. 해당 Particle 이 정해진 영역을 벗어났는 지 판단

   벗어났다면, 다시 돌아가기 위한 방향벡터를 계산

3. 만약 Particle 이 벗어났지만 다시 돌아오는 방향이라면

   그대로 종료

```
/*
    정해진 영역을 벗어났으니, 들어오는 방향으로 particle 의 속도를 바꾸어야 한다

    영역으로 들어오는 - 노멀 벡터를 속도로 가지도록 한다
    그리고 경계와 부딪혔으니, damping 값 만큼 속도가 느려지게 한다
*/
particle.velocity = norVec * (1 - damping);


/*
    그리로 영역을 벗어난만큼, particle 을 직접 이동 시킨다
    원하는 영역 안으로 직접 이동시킨다
*/
particle.position += overLength * (1.0001f) * norVec;



// 수정한 내용을 버퍼에 저장한다
particles[id.x] = particle;
```

4. 정해진 영역을 벗어나는 방향으로 진행 중 이였다면

   정해진 영역으로 돌아오도록 방향을 바꾸어 준다

   떨어진 만큼, 위치를 강제로 옮겨서 확실히 정해진 영역에 들어오도록 한다

B. 유체의 색깔

- 기본적인 블린-퐁 모델 구현
- 유체의 투명함을 표현하기 위해서, 알파 블렌딩 사용
  - Unity 엔진의 ShaderLab 에서 기본적으로 제공하는 명령어 사용
- 논문의 내용을 따르면, 각 Particle 에서 유체를 향하는 Normal Vector 를 구할 수 있다
  - 해당 Normal Vector 를 사용하여, 라이팅 계산
  - 처음 계산된 Normal Vector 의 길이 값을 이용하면 그 Particle 이 얼마나 표면에 존재하는 지 알 수 있다

B. 유체의 색깔

- 임의의 Threshold 값을 지정해서, 어떤 Particle 이 표면인지 내부인지를 판단
- 내부라 판단되면, fragment shader 에서 아무것도 하지 않는다
  - 투명한 유체 내부를 표현
- 외부라 판단되면, 표면에 가까울 수록 불투명하게 표현
  - 표면에서 내부로 향할수록 투명해지는 걸 표현

```
//Mesh 를 이루는 정점에 대한 처리
v2f vert (appdata v, uint instanceID : SV_InstanceID)
{
    v2f o;

    //현재 정점이 속한 인스턴스인 Particle 의, 월드 공간에서의 위치를 가져온다
    float3 pos = particles[instanceID].position;


    // 월드 좌표계 => 모델 좌표계 월드 좌표, 일치 시키는 변환을
    // 각 모델 정점에 적용 => 정점의 월드 좌표계를 얻을 수 있다
    // 모델 좌표계의 정점을, 인스턴스가 위치한 위치로 보내는 이동변환을 만든다
    // 그리고 주어진 radius 만큼 크기가 변하도록 확대변환도 추가한다
    // 이동변환 + 확대변환
    float4x4 affinMat = float4x4 (
                                    particleRadius, 0, 0, pos.x,
                                    0, particleRadius, 0, pos.y,
                                    0, 0, particleRadius, pos.z,
                                    0,0,0,1
                                );

    //모델의 정점에 이동변환을 적용한다 => 해당 위치에 Particle Instance 가 생기도록 한다
    v.vertex = mul(affinMat, v.vertex);


    /*
        정점의 위치가 변했으니, 노멀벡터도 변환을 해주어야 한다

        원래 노멀 벡터는, 노멀 정점과 다른 변환을 적용
        [L | t] 변환 중,
        linear transform 인 L 변환에 대해서 => (L ^ -1) ^ T 를 노멀 벡터에 적용
        inverse transform 적용 => 후 normalize

        근데 여기서는 모델 => 월드 변환 시, 회전이 없다 + uniform scaling
        따라서 노멀 벡터를 변환하지 않고, 그냥 있던 것을 사용하면 된다
    */
    /*
        근데 particle.surfNormal = 바깥에서 fluid 를 향하는 벡터 방향
        흔히 정점의 노멀벡터라면 바깥을 향하는 방향
        따라서 방향을 바꾸어 주어야 한다
    */
    float3 normalWorld = -particles[instanceID].surfNormal;
```

B–1. Vertex Shader

1. SV_InstanceID 를 통해서, Instancing 을 통해 그려지는 instance 에 접근

   해당 instance id 와
   Compute Shader 에서 사용한 스레드 id 로 동일한 버퍼에 접근한다

   따라서 동일한 ID 가 같은 Mesh 에 접근한다고 생각할 수 있다

2. 계산된 위치에 맞춰서 모델 변환 실행

```
    //v2f o 를 리턴한다
    o.vertexWorld = v.vertex;
    o.vertexClip = UnityWorldToClipPos(v.vertex);

    /*
        노멀 벡터의 크기는 원래 라이팅에 중요하지 않다
        하지만 여기서는 계산된 노멀벡터의 크기 => 경계 여부를 판단
        따라서 크기를 저장해둔다
    */
    o.surfaceNormalWorld = normalWorld;
    o.surfaceNormalLength = length(normalWorld);



    return o;
}
```

4. Lighting에 필요한 월드 좌표계에서의 값을 fragment shader 에 넘긴다

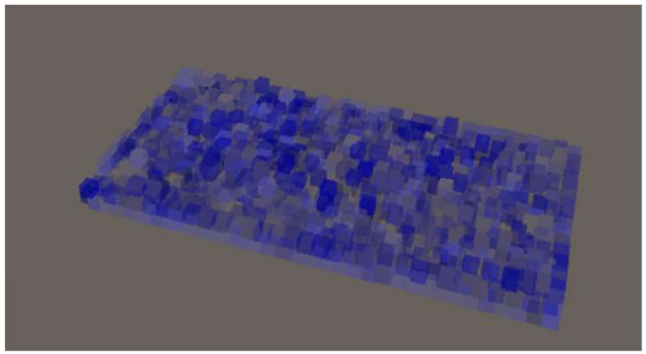
```
//Fragment Shader
fixed4 frag (v2f i) : SV_Target
{
    // 계산된 노멀 벡터의 크기 판단
    // 그냥 표면 파티클이 아니면 그리지 않는다
    if(i.surfaceNormalLength < surfTrackThreshold) return fixed4(0, 0, 0, 0);


// 1. Ambient Color
    float3 AmbientTerm = (_AmbientColor.xyz) * _AmbientDegree;

// 2. Diffuse Color
    //프레그먼트의 월드 공간 좌표계에서 surface normal 을 구한다
    float3 norVec = normalize(i.surfaceNormalWorld);

    //빛 벡터 = 해당 프레그먼트에서 광원을 바라보는 벡터
    //월드 공간에서 빛 벡터를 구한다
    float3 lightVec = normalize(UnityWorldSpaceLightDir(i.vertexWorld));

    float3 DiffuseTerm = max(dot(norVec, lightVec), 0) * (_DiffuseColor.xyz) * _DiffuseDegree;

// 3. Specular Color

    //뷰 벡터 = 해당 프레그먼트에서 카메라를 바라보는 벡터
    //뷰 벡터를 월드 공간에서 구한다 (프레그먼트의 월드 공간 좌표를 넣어서 구할 수 있다)
    float3 viewVec = normalize(UnityWorldSpaceViewDir(i.vertexWorld));

    // 빛 반사 벡터 => 프레그먼트에서 반사된 빛이 나아가는 방향
    float3 reflectVec = reflect(lightVec, norVec);

    // 눈에 들어오는 빛의 양
    float lightAmountOnEye = max(0, dot(viewVec, reflectVec));

    // 매끄러움 계산
    float specular = pow(lightAmountOnEye, _Shininess);

    // 정반사 빛
    float3 SpecularTerm = specular * (_SpecColor.xyz) * _SpecularDegree;

    // 최종 색깔
    float4 TotalColor = float4(AmbientTerm + DiffuseTerm + SpecularTerm, 0);
```

```
// 4. 투명도 결정
//
    //표면의 정도를 나타내는 변수를 만들자
    //이 값이 0 에 가까울수록 내부이며, 클수록 표면이다
    float surfCoef = i.surfaceNormalLength - surfTrackThreshold;

    //표면일수록 불투명한 값을 가지게 하자
    TotalColor.w = surfCoef * _AlphaValue;

    return TotalColor;
}
```

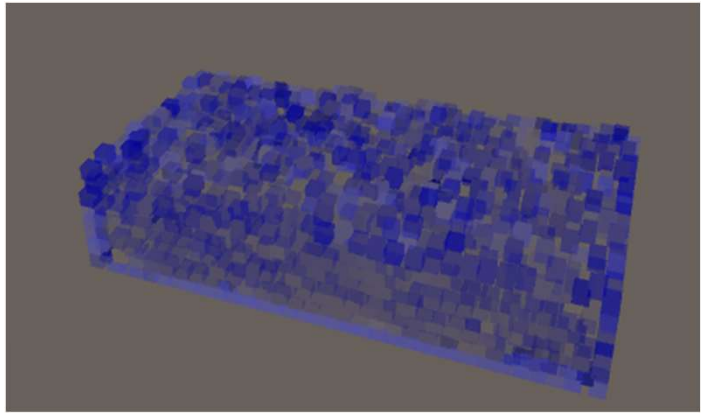
B–2. Fragment Shader

1. Ambient, Diffuse, Specular 라이팅 구현
2. 내부 Particle 은 그리지 않는다
3. 표면에 가까울수록 불투명한 값을 가지게 한다

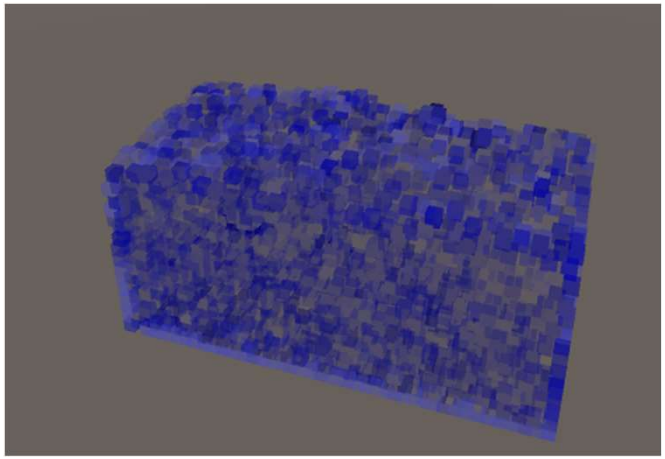
4. 결과



10 * 10 * 10
particles



20 * 20 * 10
particles

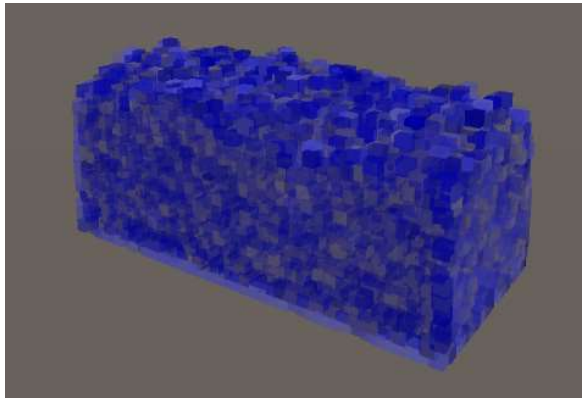


30 * 30 * 10
particles

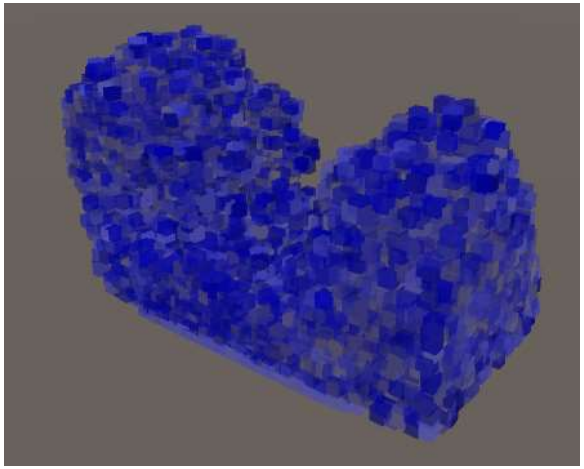
Compute Shader 덕분에 Particles 가 늘어나도 빠르게 연산이 가능함을 알 수 있었다

단, 10000 개를 넘어가는 순간 부터는, 끊김이 발생해서 매끄러운 실행이 어려웠다

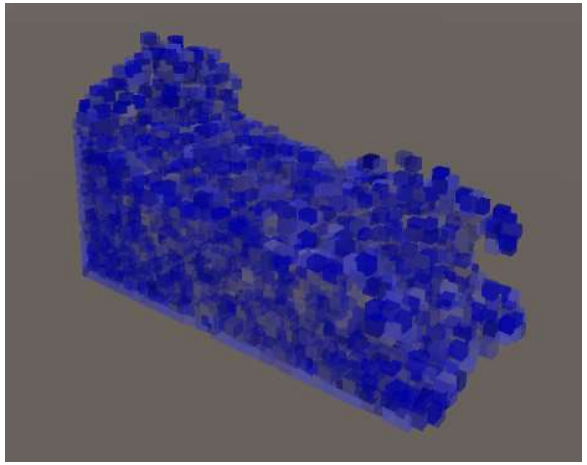
또한 유체의 움직임이 잘 나타나는 것을 확인할 수 있었다
(https://youtu.be/w5LDP7CwzFk)



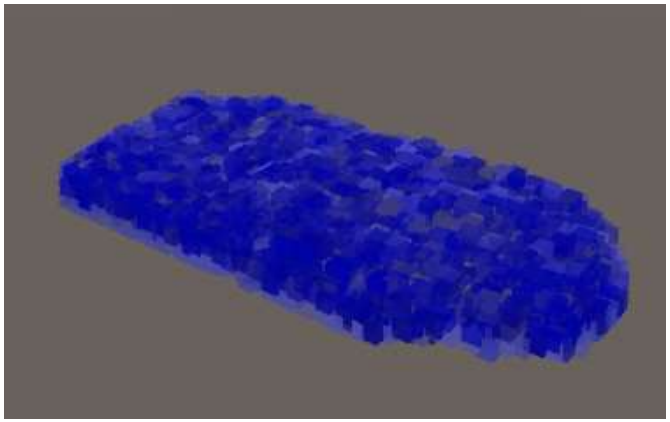
외력이 없을 때



Wave 가 적용될 때



외력을 받았을 때



표면장력이 크게 작용할 때

5. 느낀 점

- 그래픽스 파이프라인에 대해 더 자세히 알 수 있어서 뿌듯했다
- 사실적인 그래픽을 표현하는 작업이 즐거웠다
- 논문의 내용을 공부해보면서, 더 많은 그래픽 이론에 대해 궁금해졌다
- Unity 엔진의 도움덕에 많은 것을 쉽게 진행할 수 있었지만, 또 불편한 점들도 많았다. 그래픽스 API 공부에 대한 필요성을 느낄 수 있었다