Smoothed Particle Hydrodynamics for Real-Time Fluid Simulation for Unity

Yohib Hussain

Contents

Abstract

Realistically animated fluids can add substantial realism to interactive applications such as virtual surgery simulators or computer games. In this paper we try to implement a Fluid Simulation using Smooth Particle Hydrodynamics (SPH) to simulate fluids with free surfaces in Unity Game Engine. The method is an extension of the SPH-based technique by Desbrun to animate highly deformable bodies adapting it for interactive real-time use in Unity.

My implementation leverages compute shaders to effciently parallelize particle interactions on the GPU, enabling simulations with thousands of particles at interactive frame rates. Key physical effects such as pressure, velocity and external forces are incorporated, and rendering techniques are applied to visulize fluid dynamics.

Results demonstrate that the method produces visually plausible fluid motion with good performance across different particle counts. While not physically exact compared to high-resolution grid solvers, the approach balances realism and effciency, making it suitable for applications where interactivity is critical. Future improvements may include surface tension modeling, multi-phase fluids, and integration with rigid-body dynamics.

1. Introduction

1.1 Motivation

Fluids (i.e liquids and gases) play an important role in every day life. Examples of fluid phenomena are wind, weather, ocean waves, waves induced by ships or simply pouring of a glass of water. As simple and ordinary these phenomena may seem, as complex and difficult it is to simulate them. Even though Computational Fluid Dynamics (CFD) is a well established research area with a long history, there are still many open research problems in the field. The reason for the complexity of fluid behaviour is the complex interplay of various phenomena such as convection, diffusion, turbulence and surface tension. Fluid phenomena are typically simulated off-time and then visualized in a second step e.g. in aerodynamics or optimization of turbines or pipes with the goal of being accurate as possible.

Less accurate methods that allow the simulation of fluid effects in real-time open up a variety of new applications. In the fields mentioned above real-time methods help to test whether a certain concept is promising during the design phase. Other applications for real-time simulation techniques for fluids are medical simulators, computer games or any type of virtual environment. We will take a look into the computer game side of the fluid simulation

1.2 Topic

The focus of this thesis is Smoothed Particle Hydrodynamics (SPH), a particle-based method for simulating the flow of fluids. While SPH can be also applied to gases, in this work the emphasis will be place on liquid fluids only.

2. Smooth Particle Hydrodynamics

Although Smoothed Particle Hydrodynamics (SPH) was developed for the simulation of astrophysical problems, the method in general enough to be used in any kind of fluid simulation.

SPH is an interpolation method for particle systems. With SPH, fleid quantities that are only defined at the discrete particle locations can be evaluted anywhere in space. For this purpose, SPH distributes quantities in a local neighborhood of each particle using radial symmetrical smoothing kernels.

Accoring to SPH, a scalar quantity A is interpolated at location $\mathbf{r}$ by a weighted sum of contributions from all particles:

$$A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{1}$$

where j iterates over all the particles, m_j is the mass of the particle j, $\mathbf{r}_j$ its position, ρ_j is the density and A_j is the field quantity at $\mathbf{r}_j$.

The function $W(r, h)$ is called the smoothing kernel with core radius h. It defines the effective range of interaction in the simulation: We can use this to get the estimates on the density, pressure and viscosity of the given particle.

The particle mass and density appear in Eqn. (1) because each particle i represents a certain volume $V_i = \frac{m_i}{\rho_i}$. While the mass m_i is constant throughout the simulation and, in our case, the same for all the particles, the density ρ_i varies and needs to be evaluated at every time step. Through substitution into Eqn (1) we get for the density at location $\mathbf{r}$

$$\rho_S(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{2}$$

3. Modelling Fluids with Particles

In the Eulerian (grid based) formulation, isothermal fluids are described by a velocity field v, a density field ρ and a pressure field p. The evolution of these quentities over time is given by two equations. The first equation assures convservation of mass.

equation over here.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{3}$$

while the Navier-Stokes equation formulates conservation of momentum

equation over here.

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v} \tag{4}$$

where g is the external force density field and μ the viscosity of the fluid. Many forms of the Navier-Stokes equation appear in the literature. Eqn. (4) represents a simplified version for incompressible fluids.

In the SPH formulation, the accleration of a particle i is obtained from the Newton's second law:

$$\frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{F}_i}{m_i} \tag{5}$$

where $\mathbf{v}_i$ is the velocity of the particle and i and $\mathbf{f}_i$ and ρ_i are the force density field and the density field evaluated at the location of particle i, repectively.

This formulation is equivalent to the continuum expression

$$\frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{f}_i}{\rho_i} \tag{6}$$

where $\mathbf{f}_i$ denotes the force per unity volume and ρ_i the density, as often written in SPH literature.

Now we will see the force density terms using SPH.

3.1 Pressure

Application of the SPH rule described in Eqn (1) to the pressure term $-\nabla p$ yields

$$\mathbf{f}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{7}$$

Unfortunately, this force is not symmetric as can be seen when only two partciles interact. Since the gradient of the kernel is zero as its center, particle i only uses the pressure of particle j to compute its pressure force and vice versa. Because the pressures at the locations of the two particles are not equal in general, the pressure forces will not be symmertic. Different ways of symmertrization of Eqn. (7) have been proposed in the literature. But I have went with this

$$\mathbf{f}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_j m_j \frac{p_i + p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{8}$$

The so computed pressure force is symmetric because it uses the arithmetic mean of the pressures of interacting particles.

Since particles only carry the three quantities mass, position and velocity, the pressure at particle locations has to be evaluated first. This is done in two steps. Eqn (2) yields the density at the location of the particle. Then, the pressure can be computed via the ideal gas state equation

$$p = k\rho \tag{9}$$

where k is the gas constant that depends on the temperature. In my simulation I have used a modified version of Eqn (9)

$$p = k(\rho - \rho_0) \tag{10}$$

where ρ_0 is the rest density. Which I have given as a constant

3.2 Viscosity

Application of the SPH rule to the viscosity term $\mu\nabla^2\mathbf{v}$ again yields the asymmetric forces

$$\mathbf{f}_i^{viscosity} = \mu\nabla^2\mathbf{v}(\mathbf{r}_a) = \mu\sum_j m_j\frac{v_j}{\rho_j}\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{11}$$

because the velocity field varies from particle to particle. Since viscosity forces are only depedent on the velocity differences and not on absolute velocities, there is a natural way to symmetrize the viscosity forces by using the velocity differences:

$$\mathbf{f}_i^{viscosity} = \mu\sum_j m_j\frac{v_j - v_i}{\rho_j}\nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{12}$$

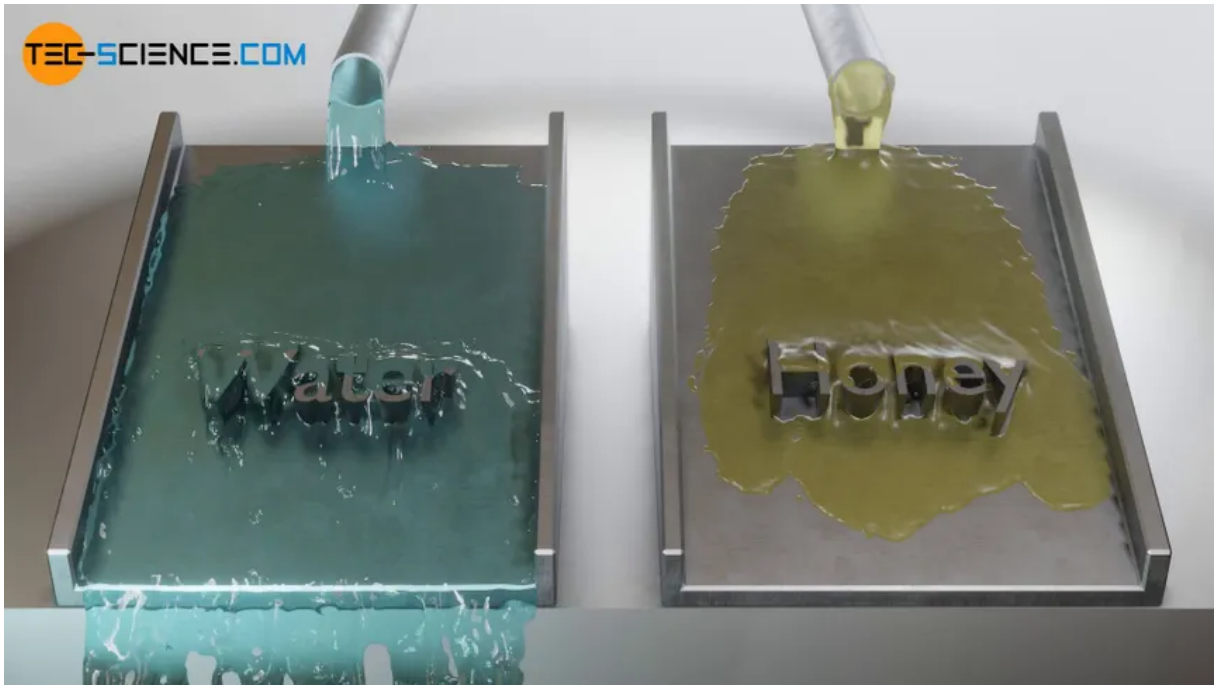

Figure 1: Difference viscosity makes to a fluid

3.3 External Forces

In the Unity Project which supports gravity, collision forces and forces caused by user interaction. These forces are applied directly to the particles without the use of SPH. When the particles collide with any solid object such as the blue box in my Unity Simulation, we simply push them out of the object and reflect the velocity component that is perpendicular to the object's surface.

4. Smoothing kernels

Smoothing kernels are fundamental in SPH as they determine how particle properties are interpolated over space. They provide stability, accuracy and computational effciency to the simulation, such as density estimation, pressure forces, and viscosity. I have used two types of smoothing kernels

- Poly6 Kernel: This kernel is used for density estimation. It's smooth shape ensures that conributions from nearby particles are weighted appropriately, avoiding sharp discontinuities.

$$W_{\text{poly6}}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

- Spiky Kernel: Used for calculating pressure and viscosity forces. Its gradient is sharper near the particle center, which improves force calculations and help maintain stability.

$$W_{\text{spiky}}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

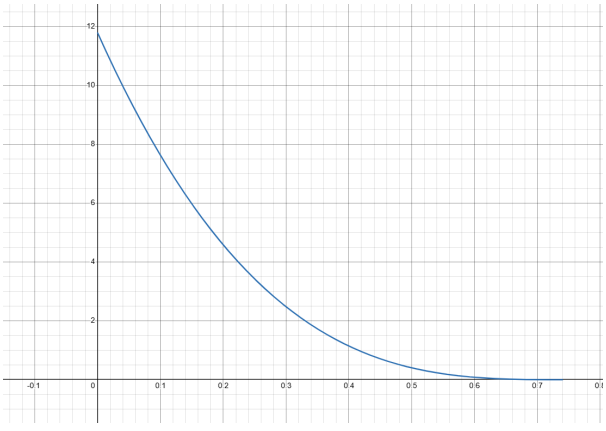

Figure 2: Spiky kernel

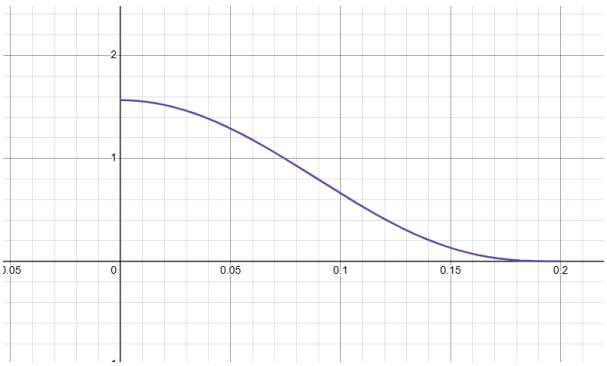

Figure 3: Poly6 kernel

4.1 Derivation of Kernel Functions

4.1.1 Poly6 Kernel Derivation

$$W(r,h) = \frac{315}{64\pi h^9}(h^2 - r^2)^3 \tag{15}$$
$$= \frac{315}{64\pi h^9}\Big(h^2(1 - \frac{r^2}{h^2})\Big)^3 \tag{16}$$
$$= \frac{315}{64\pi h^9} h^6 \left(1 - \frac{r^2}{h^2}\right)^3 \tag{17}$$
$$= \frac{315}{64\pi h^3} \left(1 - \frac{r^2}{h^2}\right)^3 \tag{18}$$
$$= \frac{315}{64\pi h^3} x^3, \quad \text{with } x = 1 - \frac{r^2}{h^2} \tag{19}$$

4.1.2 Spiky Kernel First Derivate

$$\frac{dW}{dr} = -\frac{45}{\pi h^6}(h - r)^2 \tag{20}$$
$$= -\frac{45}{\pi h^6}(h^2(1 - r/h)^2) \tag{21}$$
$$= -\frac{45}{\pi h^4}(1 - r/h)^2 \tag{22}$$
$$= -\frac{45}{\pi h^4} x^2, \quad \text{with } x = 1 - r/h \tag{23}$$

4.1.3 Spiky Kernel Second Derivate

$$\frac{dW}{dr} = \frac{d}{dr}\left[-\frac{45}{\pi h^6}(h - r)^2\right] \tag{24}$$
$$= -\frac{45}{\pi h^6} \cdot 2(h - r) \cdot \frac{d}{dr}(h - r) \tag{25}$$
$$= -\frac{45}{\pi h^6} \cdot 2(h - r) \cdot (-1) \tag{26}$$
$$= \frac{90}{\pi h^6}(h - r) \tag{27}$$
$$= \frac{90}{\pi h^6}(h(1 - r/h)) \tag{28}$$
$$= \frac{90}{\pi h^6} h\, x \quad \text{with } x = 1 - r/h \tag{29}$$
$$= \frac{90}{\pi h^5} x \tag{30}$$

5. Surface Rendering via Ray Marching

5.1 The Rendering Problem

The SPH produces a set of discrete particles, each representing a point within the fluid volume. A native rendering of these particles as individual spheres or points fails to convey the impression of continuous fluid with a coherent surface. Therefore, a method is required to reconstruct and render a visually plausible surface from this particle data in real-time.

5.2 The Different techniques of Rendering Methods

Two primary techniques exist for this task: polygonization and screen-space rendering.

- Polygonization methods, such as the popular Marching Cubes algorithm, first convert the particle data into a density field on a 3D grid (a process known as voxelization). The algorithm then traverses this grid to generate a triangle mesh representing the fluid's surface, which can be rendered with standard techniques. While capable of producing high-quuality meshes, this approach was deemed unsuitable for this project due to tw main drawbacks: the high computational cost of voxelization and mesh generation, and the performance bottleneck associated with transferring a potentially large, dynamic mesh from the GPU(where the simulation runs) to the CPU for rendering in Unity.
- Screen-space methods, by contrast, operate directly on the GPU for each pixel on the screen. The choosen method for this project is Ray Marching an implicit surface, which avoids the creation of any intermediate geometry and is exceptionally well-suited for the parallel architecture of the GPU.

5.3 Theory of Ray Marching

The core of the rendering technique is to define the fluid surface not with triangles, but as an implicit surface. The surface is defined as the set of all points p in space where a function $\mathbf{F(p)}$ equals zero. Specifically, we use a Signed Distance Field (SDF), a special type of implicit function that, for any point p, returns the shortest distance to the surface. The sign of the distance is positive outside the fluid and negative inside.

5.4 Constructing the Fluid SDF

The SDF for the entire fluid is built by combining the SDFs of the individual particles. A single is presented by a sphere, whose SDF is given by:

equation here

To combine the fields from all particles into a single, smooth "blobby" surface, a simple minimum operation is insufficient as it would create sharp creases. Instead, a polynomial smooth minimum(*smin*) function is used. This smoothly bends the distance fields of nearby particles.

5.5 The Ray Marching Algorithm

With the SDF defined, the surface can be rendered using a ray marching algorithm, also known as sphere tracing. For each pixel on the screen, a ray is cast from the camera. The algorithm proceeds iteratively:

1. From the ray's current position, the SDF is evaluated to find the distance d to the surface.
2. The SDF guarantees that we can safely "march" the ray forward along its direction by the distance d without passing through the surface.
3. This process is repeated. If d becomes smaller than a small threshold (epsilon), the ray has hit the surface. If the ray travels too far, it is considered a miss.

This process is highly efficient as it takes the largest possible safe steps through empty space.

6. Implementation and Methods

6.1 Project Setup

A simple object scene was prepared in order for the simulation to be tested. There is a simple Ground cube and a few test objects which are the sphere and cube here.

We then create a SPH.cs script that initialize the particles that we are going to use.

6.2 SPH.cs Script implementation

6.2.1 Particle Sturcture

The First step in the script is to define a data structure that represents a single particle in the simulation. Each particle needs to store its physical properties (pressure, density), its current motion (velocity, force), and its position in the world. To make sure this structure is compatible between C# and the compute shader, it is laid out in memory sequentially.

```
[System.Serializable]
[StructLayout(LayoutKind.Sequential, Size=44)]
public struct Particle {
    public float pressure; // 4 bytes
    public float density; // 8 bytes
    public Vector3 currentForce; // 20 bytes
    public Vector3 velocity; // 32 bytes
    public Vector3 position; // 44 total bytes
}
```

Listing 1: Particle struct

This structure is only 44 bytes, which is small enough to handle thousands of particles efficiently on the GPU. Each particle is initialized with default values in the script, and later updated every frame by the compute shaders (for density, pressure, force, and integration).

6.2.2 Settings for SPH Class

We will then create a **SPH class** which will contain all the settings for the SPH Simulation:

- **General Settings**
  - `collisionSphere`: We can assign our test sphere which can interact with the fluid.
  - `showSpheres`: Toggle to show the particles in the scene (useful for debugging).
  - `numToSpawn`: Number of particles to spawn along each axis (X,Y,Z).
  - `totalParticles`: The number of particles to spawn (product of the per-axis count).
  - `boxSize`: The bounding box of the scene.
  - `spawnCenter`: The spawn center of the particle grid in the scene.
  - `particleRadius`: Radius of each particle.
  - `spawnJitter`: Adds randomness so the particle grid is not perfectly uniform.
- **Particle Rendering Settings**
  - `particleMesh`: Mesh of the particle.
  - `particleRenderSize`: Size in which the particle will render.
  - `material`: Material for the particles.
- **Compute Settings**
  - `shader`: Our compute shader.
  - `particles`: List that stores all the particles in the scene.
- **Fluid Constants**
  - `boundDamping`: Value used when particles collide with the boundary (reflects velocity and scales it down to simulate energy loss).
  - `viscosity`: Viscosity of the fluid.
  - `particleMass`: Mass of an individual particle.
  - `gasConstant`: The gas constant.
  - `restDensity`: Rest density of the fluid.
  - `timestep`: Simulation timestep.
- **Private Compute Shader Buffer Variables**
  - `_argsBuffer`: Arguments related to objects in the scene (sent to compute shader).
  - `_particlesBuffer`: Total amount of particles sent to the compute shader.
  - `integrateKernel`: The integrate function in the compute shader.
  - `computeKernel`: The compute function in the compute shader.
  - `densityKernel`: The density function in the compute shader.

```
public class SPH : MonoBehaviour
{
    [Header("General")]
    public Transform collisionSphere;
    public bool showSpheres = true;
    public Vector3Int numToSpawn = new Vector3Int(10,10,10);
    private int totalParticles {
        get {
            return numToSpawn.x*numToSpawn.y*numToSpawn.z;
        }
    }
    public Vector3 boxSize = new Vector3(4,10,3);
    public Vector3 spawnCenter;
    public float particleRadius = 0.1f;
    public float spawnJitter = 0.2f;

    [Header("Particle Rendering")]
    public Mesh particleMesh;
    public float particleRenderSize = 8f;
    public Material material;

    [Header("Compute")]
    public ComputeShader shader;
    public Particle[] particles;

    [Header("Fluid Constants")]
    public float boundDamping = -0.3f;
    public float viscosity = -0.003f;
    public float particleMass = 1f;
    public float gasConstant = 2f;
    public float restingDensity = 1f;
    public float timestep = 0.007f;

    // Private Variables
    private ComputeBuffer _argsBuffer;
    public ComputeBuffer _particlesBuffer;
    private int integrateKernel;
    private int computeKernel;
    private int densityPressureKernel;

}
```

Listing 2: SPH Class and Settings

6.2.3 Gizmos Function to Draw the Boxes

I am using the `OnDrawGizmos` function to the boundary boxes and the `spawnCenter` (for debugging).

```
private void OnDrawGizmos() {


        // Draw simulation bounding box
        Gizmos.color = Color.blue;
        Gizmos.DrawWireCube(Vector3.zero, boxSize);

        // Draw spawn center (only in editor, not while running)
        if (!Application.isPlaying) {
            Gizmos.color = Color.cyan;
            Gizmos.DrawWireSphere(spawnCenter, 0.1f);
        }

}
```

Listing 3: OndrawGizmos() function

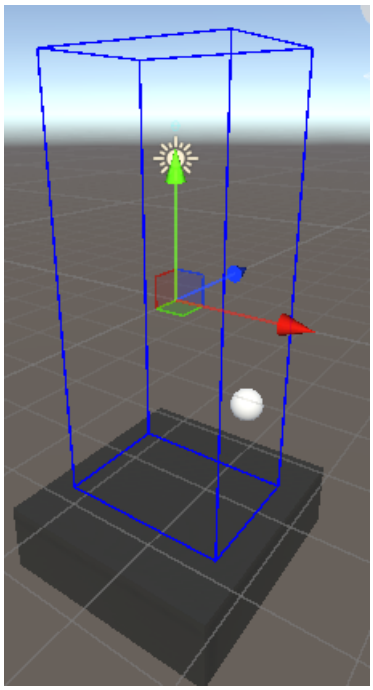

Figure 4: Gizmos Box

6.2.4 Spawning the Particles in a Grid

The SpawnParticleInBox() function takes the `spawnCenter` and stores it in the variable `spawnPoint` and creates a new List called _particles, It it loops and goes in a grid like fashion across the three axis (`x`,`y`, and `z`), using `numToSpawn` to determine the number of particles per axis. Each particle is positioned relative to the `spawnCenter` with spacing determined by twice the particle radius.

To avoid the grid being a perfectly uniform grid which will lead to simulation accuracy, a small random offset is added to each position using `Random.onUnitSphere * particleRadius * spawnJitter`. This creates a slight offset in the particles that should give more accurate results.

```
1  private void SpawnParticlesInBox() {
2
3          Vector3 spawnPoint = spawnCenter;
4          List<Particle> _particles = new List<Particle>();
5
6          for (int x = 0; x < numToSpawn.x; x++) {
7              for (int y = 0; y < numToSpawn.y; y++) {
8                  for (int z = 0; z < numToSpawn.z; z++) {
9
10                     Vector3 spawnPos = spawnPoint + new Vector3(x*
                           particleRadius*2, y*particleRadius*2, z*particleRadius
                           *2);
11
12                     // Randomize spawning position a little bit for more
                           convincing simulation
13                     spawnPos += Random.onUnitSphere * particleRadius *
                           spawnJitter;
14
15                     Particle p = new Particle {
16                         position = spawnPos
17                     };
18
19                     _particles.Add(p);
20                 }
21             }
22         }
23
24         particles = _particles.ToArray();
25
26 }
```

Listing 4: SpawnParticleInBox() function

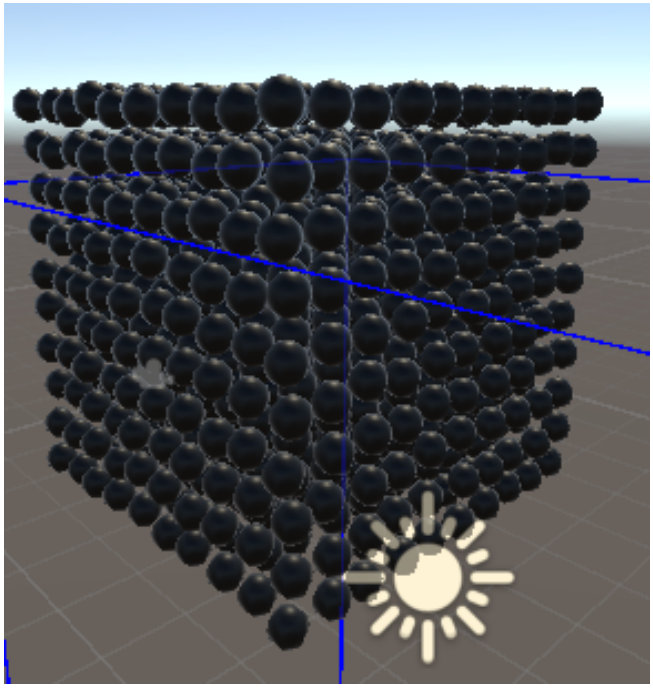

Figure 5: Particle in a grid box

6.2.5 Initalizing the Simulation

The Awake function is called when the simulation object is initalized, It first calls `SpawnParticlesInBox()` to generate the initial particle distribution. Afterwards, it sets up the data structures required for the GPU compute shaders.

And Finally it calls the `SetupComputeBuffers()`.

```
private void Awake() {

        SpawnParticlesInBox(); // Spawn Particles

        // Setup Args for Instanced Particle Rendering
        uint[] args = {
            particleMesh.GetIndexCount(0),
            (uint)totalParticles,
            particleMesh.GetIndexStart(0),
            particleMesh.GetBaseVertex(0),
            0
        };

        _argsBuffer = new ComputeBuffer(1,args.Length * sizeof(uint),
            ComputeBufferType.IndirectArguments);
        _argsBuffer.SetData(args);

        // Setup Particle Buffer
        _particlesBuffer = new ComputeBuffer(totalParticles,44);
        _particlesBuffer.SetData(particles);

        SetupComputeBuffers();

}
```

Listing 5: Awake Function

6.2.6 Setting up Compute Buffers

The function `SetupComputeBuffers` sets up all the Buffers from the compute shader and also passes the values which are calculated from the `SPH.cs` script to the compute shader.

```
private void SetupComputeBuffers() {

        integrateKernel = shader.FindKernel("Integrate");
        computeKernel = shader.FindKernel("ComputeForces");
        densityPressureKernel = shader.FindKernel("ComputeDensityPressure");

        shader.SetInt("particleLength", totalParticles);
        shader.SetFloat("particleMass", particleMass);
        shader.SetFloat("viscosity", viscosity);
        shader.SetFloat("gasConstant", gasConstant);
        shader.SetFloat("restDensity", restingDensity);
        shader.SetFloat("boundDamping", boundDamping);
        shader.SetFloat("pi", Mathf.PI);
        shader.SetVector("boxSize", boxSize);

        shader.SetFloat("radius", particleRadius);
        shader.SetFloat("radius2", particleRadius * particleRadius);
        shader.SetFloat("radius3", particleRadius * particleRadius *
           particleRadius);
        shader.SetFloat("radius4", particleRadius * particleRadius *
           particleRadius * particleRadius);
        shader.SetFloat("radius5", particleRadius * particleRadius *
           particleRadius * particleRadius * particleRadius);

        shader.SetBuffer(integrateKernel, "_particles", _particlesBuffer);
        shader.SetBuffer(computeKernel, "_particles", _particlesBuffer);
        shader.SetBuffer(densityPressureKernel, "_particles", _particlesBuffer
           );

}
```

Listing 6: SetupComputeBuffers Function

6.2.7 Rendering the particles

The `Update` function is responsible for just rendering the particles that we are initilizing which are rendered with the `GridParticle.shader`

```
private static readonly int SizeProperty = Shader.PropertyToID("_size");
private static readonly int ParticlesBufferProperty = Shader.PropertyToID("
    _particlesBuffer");

private void Update() {

    // Render the particles
    material.SetFloat(SizeProperty, particleRenderSize);
    material.SetBuffer(ParticlesBufferProperty, _particlesBuffer);

    if (showSpheres)
        Graphics.DrawMeshInstancedIndirect (
            particleMesh,
            0,
            material,
            new Bounds(Vector3.zero, boxSize),
            _argsBuffer,
            castShadows: UnityEngine.Rendering.ShadowCastingMode.Off
        );


}
```

Listing 7: Update Function

6.2.8 Calling the Compute Shader

The `FixedUpdate` is called every physics frame. This function passes all the parameters to the compute shader and dispatches the individual kernels and divides them by 100 because the kernel we are using uses 100 threads.

```
private void FixedUpdate() {

        shader.SetVector("boxSize", boxSize);
        shader.SetFloat("timestep", timestep);
        shader.SetVector("spherePos", collisionSphere.transform.position);
        shader.SetFloat("sphereRadius", collisionSphere.transform.localScale.x
            /2);

        // Total Particles has to be divisible by 100
        shader.Dispatch(densityPressureKernel, totalParticles / 100, 1, 1);
        shader.Dispatch(computeKernel, totalParticles / 100, 1, 1);
        shader.Dispatch(integrateKernel, totalParticles / 100, 1, 1);
}
```

Listing 8: FixedUpdate Function

6.3 SPH Compute compute shader

The SPH Compute function is the function that computes the forces for each particles to neighbouring particles and sends the data to Unity to use.

6.3.1 Initialization of the Kernels

- **Kernels**
  - `Integrate`: The `Integrate` kernel updates each particle's position and velocity based on the computed forces.
  - `ComputeForces`: The `ComputeForces` kernel calculates forces between particles, such as pressure and viscosity interactions.
  - `ComputeDensityPressure`: The `ComputeDensityPressure` kernel computes each particle's density and pressure.

We are implementing the respective kernels to Integrate the timestep which the `Integrate` kernel does, we compute the forces between the particles using `ComputeForces` kernel, and finally we are use the `ComputeDensityPressure` kernel to compute the density and pressure of each of the particle.

```
#pragma kernel Integrate // Use the force of each particle to move particle
#pragma kernel ComputeForces // Compute forces for each particle
#pragma kernel ComputeDensityPressure // Compute density/pressure for each
    particle

struct Particle
{
    float pressure;
    float density;
    float3 currentForce;
    float3 velocity;
    float3 position;
};
```

Listing 9: Initilize block of the shader

In the particle struct, scalar properties (e.g., pressure, density) are represented by float types, whereas vector properties (e.g., position, velocity, currentForce) are represented by float3 types. The reason they are multiple different radius `radius2`, `radius3`, etc are for kernel optimizations

6.3.2 Setting up the variables for the Compute shader

We set a `RWStructedBuffer` and also set all of the different variables which are passed in the `SPH.cs` script

```
RWStructuredBuffer<Particle> _particles;

float particleMass;
float viscosity;
float gasConstant;
float restDensity;
float boundDamping;
float radius;
float radius3;
float radius2;
float radius4;
float radius5;
float pi;
float timestep;
float3 boxSize;
float3 spherePos;
float sphereRadius;

int particleLength;
```

Listing 10: Block that contains variables of the shader

6.3.3 The Integrate Kernel Function

- The first two `float3` variables define the boundaries for the particles so that they do not move outside the simulation box.
- The `float3 vel` is updated according to Newton's law of motion: $F = m \cdot a$, where the acceleration is `currentForce / particleMass` for the given particle.
- The position is updated using the distance–speed–time formula: $v = \frac{d}{t}$.
- Boundary conditions are enforced by checking if a particle exceeds the domain along any axis. If it does, the velocity is damped by `boundDamping` and the position is corrected to remain inside the box.
- Below that, the boundary box checks are applied for all axes.

```
[numthreads(100,1,1)]
void Integrate (uint3 id: SV_DISPATCHTHREADID)
{
    float3 topRight = boxSize / 2;
    float3 bottomLeft = -boxSize /2;

    float3 vel = _particles[id.x].velocity + ((_particles[id.x].currentForce/
       particleMass) * timestep);
    _particles[id.x].position += vel * timestep;


    // Minimum Enforcements

    if (_particles[id.x].position.x - radius < bottomLeft.x) {
       vel.x *= boundDamping;
        _particles[id.x].position.x = bottomLeft.x + radius;
    }

    if (_particles[id.x].position.y - radius < bottomLeft.y) {
       vel.y *= boundDamping;
        _particles[id.x].position.y = bottomLeft.y + radius;
    }

    if (_particles[id.x].position.z - radius < bottomLeft.z) {
       vel.z *= boundDamping;
        _particles[id.x].position.z = bottomLeft.z + radius;
    }

    // Maximum Enforcements

    if (_particles[id.x].position.x + radius > topRight.x) {
       vel.x *= boundDamping;
        _particles[id.x].position.x = topRight.x - radius;
    }

    if (_particles[id.x].position.y + radius > topRight.y) {
       vel.y *= boundDamping;
        _particles[id.x].position.y = topRight.y - radius;
    }

    if (_particles[id.x].position.z + radius > topRight.z) {
       vel.z *= boundDamping;
        _particles[id.x].position.z = topRight.z - radius;
    }


    _particles[id.x].velocity = vel;
}
```

Listing 11: The Integrate kernel

6.3.4 Kernel Equations

These four functions are the kernel implementations in the compute shader.

- **StdKernel**
  - This is the Standard Kernel also called the Poly6 kernel.
  - It takes the distance which is called `distanceSquared` to be faithful to the formula.
  - The formula in the function is modified for optimization purposes from Eqn. (11)
- **SpikyKernelFirstDerivative**
  - This is the First Derivation of the Spiky Kernel.
  - We compute the first derivate of the Spiky kernel because we want to know how strongly each neighbouring particle exerts pressure. In mathematical terms we are seeing the vector between the two points which is distance and radius.
  - The formula in the function has also been modified for optimization purposes from Eqn. (12)
- **SpikyKernelSecondDerivative**
  - This is the Second Derivation of the Spiky Kernel.
  - We compute the second derivate of the Spiky kernel because we want to the viscosity which involves velocity of the particles. In mathematical terms we are measuring how curved the kernel function is around the particle, a larger curvature means stronger smoothing / diffusion effect.
  - Again this forumla is also modified for optimization purposes from Eqn. (12)
- **SpikyKernelGradient**
  - This is the Gradient of the Spiky Kernel.
  - We need this because it gives us the magnitude of the vector of how strong the vector of the particles are is which we can use in the pressure computation.
  - This just returns the Spiky Kernels First Derivative and multiples that by the distance factor.

```
float StdKernel (float distanceSquared){
    // 1 - r^2/h^2
    float x = 1.0f - distanceSquared / radius2;
    return 315.f/ (64.f * pi * radius3) * x * x * x;
}
// Smoothing Function for Compute Forces
float SpikyKernelFirstDerivative(float distance){
    // 1 - r/h
    float x = 1.0f - distance/radius;
    return -45.f/(pi*radius4)*x*x;
}
float SpikyKernelSecondDerivative(float distance){
    float x = 1.0f - distance/radius;
    return 90.f / (pi*radius5) *x;
}
float3 SpikyKernelGradient(float distance, float3 direction){
    return SpikyKernelFirstDerivative(distance) *direction;
}
```

Listing 12: The Integrate kernel

6.3.5 Calculating Pressure

- This `ComputeDensityPressure` kernel is calculating the pressure of each particle by taking the particles position as the origin and looping through all the particles in the simulation and calculating its difference and getting the distance from it.
- It will then check if the particle is within a certain radius then it will apply the smoothing kernel the sum, then it will multiply it by the mass which we will get the density from.
- Then to calculate pressure we can just multiply the difference of the densities with the `gasConstant` to get the pressure of the particle.

```
[numthreads(100,1,1)]
void ComputeDensityPressure(uint3 id: SV_DISPATCHTHREADID){

    float3 origin = _particles[id.x].position;
    float sum = 0;

    for (int i = 0;i < particleLength; i++){
        float3 diff = origin - _particles[i].position;
        float distanceSquared = dot(diff, diff);

        if (radius2*0.004 >= distanceSquared*0.004){
            sum += StdKernel(distanceSquared*0.004); // Apply Smoothing kernel
        }

    }

    _particles[id.x].density = sum * particleMass + 0.000001f;
    _particles[id.x].pressure = gasConstant * (_particles[id.x].density -
       restDensity);

}
```

Listing 13: Compute Density and Pressure Kernel

6.3.6 Calculating the Force

- This `ComputeForces` kernel is then calulating the force of the particle by taking the `i` particle's position as the origin, then it loops through all the particles.
- We also do a check where we don't compute the force if the particle position is the same i.e we are checking ourselves.
- We then check the distance between the particles if the distance is twice of the radius which is the smoothing radius to remove computational overhead of calculating all the forces of the particles because we know that when the particle is far enough it's force to the other particle is non-existant.
- We then get the `_pressureGradientDirection` which is the pressure gradient direction by normalizing the vector between the particles.
- We then calculate the total pressure contribution which is stored in `_pressureContribution` which is twice the mass (the reason we are able to do this is because the mass of all the particles are same) which we multiply by the spiky kernel gradient, then we multiply thatt to the pressure formula
- For calculating viscosity we are able to use the formula and multiply with the viscosity of the fluid, we are also calculating the difference in the velocities of the particles and divide by the density which is what the formula says
- Then we finally multiply the viscosity to the Spiky Kernel Second Derivative.

```
[numthreads(100,1,1)]
void ComputeForces(uint3 id: SV_DISPATCHTHREADID){

    float3 origin = _particles[id.x].position;
    float density2 = _particles[id.x].density * _particles[id.x].density;
    float mass2 = particleMass * particleMass;
    float3 pressure = float3(0,0,0);
    float3 visc = float3(0,0,0);

    for (int i = 0;i < particleLength; i++){
        if (origin.x == _particles[i].position.x && origin.y == _particles[i].
            position.y && origin.z == _particles[i].position.z){
            continue;
        }


        float dist = distance(_particles[i].position, origin);
        if(dist < radius*2){
            float3 pressureGradientDirection = normalize(_particles[id.x].
                position - _particles[i].position);

            float3 _pressureContribution = mass2 * SpikyKernelGradient(dist,
                pressureGradientDirection);
            _pressureContribution *= (_particles[id.x].pressure / density2 +
                _particles[i].pressure / (_particles[i].density * _particles[i
                ].density));

            float3 _viscosityContribution = viscosity * mass2 * (_particles[i
                ].velocity - _particles[id.x].velocity) / _particles[i].
                density;
            _viscosityContribution *= SpikyKernelSecondDerivative(dist);

            pressure += _pressureContribution;
            visc += _viscosityContribution;

        }

    }

    _particles[id.x].currentForce = float3(0,-9.81*particleMass,0) - pressure
        + visc;

    float3 colDir = _particles[id.x].position - spherePos;
    if (length(colDir) < sphereRadius){
        _particles[id.x].currentForce += colDir * 300;
    }

}
```

Listing 14: Compute Forces Kernel

6.4 Fluid Ray Marching

6.4.1 Initializing and Setting the Render Texture

The `FluidRayMarching` script serves as the central component responsible for managing the ray marching rendering effect. Its primary function is to perform a comprehensive initialization of all the necessary data, which includes configuring the shader parameters and setting up the required textures.

A critical part of this setup is handled by the `InitRenderTexture` method, which is dedicated specifically to initializing the main render texture that the camera will use as an output target. Once this entire initialization phase is complete and every parameter is correctly established, the script's final role is to send all of this configured information to the Raymarching.compute shader, which then utilizes the data to perform the fluid rendering calculations.

```
1  public class FluidRayMarching : MonoBehaviour
2  {
3      public ComputeShader raymarching;
4      public Camera cam;
5      List<ComputeBuffer> buffersToDispose = new List<ComputeBuffer>();
6      public SPH sph;
7      RenderTexture target;
8      [Header("Params")]
9      public float viewRadius;
10     public float blendStrength;
11     public Color waterColor;
12     public Color ambientLight;
13     public Light lightSource;
14
15     void InitRenderTexture()
16     {
17         if (target == null || target.width != cam.pixelWidth || target.height
               != cam.pixelHeight)
18         {
19             if (target != null)
20             {
21                 target.Release();
22             }
23
24             cam.depthTextureMode = DepthTextureMode.Depth;
25
26             target = new RenderTexture(cam.pixelWidth, cam.pixelHeight, 0,
                   RenderTextureFormat.ARGBFloat, RenderTextureReadWrite.Linear);
27             target.enableRandomWrite = true;
28             target.Create();
29         }
30     }
31     private bool render = false;
32     public ComputeBuffer _particlesBuffer;
33 }
```

Listing 15: FluidRayMarching class Initialize

6.4.2 Passing data to the GPU for the Ray Marching Compute Shader.

The `Begin` function it calls the `InitRenderTexture` function and then passes all the parameters to the `Raymarching` shader and sets its rendering to true.

The `OnRenderImage` function it checks if the render is false it calls the `Begin` method, when render is called it passes all the parameters to the group and combines divides the threads by 8 because that was how many threads we provided in the `Raymarching` shader and dispatches the shader.

```
public void Begin()
{
        InitRenderTexture();
        raymarching.SetBuffer(0, "particles", sph._particlesBuffer);
        raymarching.SetInt("numParticles", sph.particles.Length);
        raymarching.SetFloat("particleRadius", viewRadius);
        raymarching.SetFloat("blendStrength", blendStrength);
        raymarching.SetVector("waterColor", waterColor);
        raymarching.SetVector("_AmbientLight", ambientLight);
        raymarching.SetTextureFromGlobal(0, "_DepthTexture", "
            _CameraDepthTexture");
        render = true;
}
    void OnRenderImage(RenderTexture source, RenderTexture destination)
    {

        if (!render)
        {
            Begin();
        }

        if (render)
        {

            raymarching.SetVector("_Light", lightSource.transform.forward);

            raymarching.SetTexture(0, "Source", source);
            raymarching.SetTexture(0, "Destination", target);
            raymarching.SetVector("_CameraPos", cam.transform.position);
            raymarching.SetMatrix("_CameraToWorld", cam.cameraToWorldMatrix);
            raymarching.SetMatrix("_CameraInverseProjection", cam.
               projectionMatrix.inverse);

            int threadGroupsX = Mathf.CeilToInt(cam.pixelWidth / 8.0f);
            int threadGroupsY = Mathf.CeilToInt(cam.pixelHeight / 8.0f);
            raymarching.Dispatch(0, threadGroupsX, threadGroupsY, 1);

            Graphics.Blit(target, destination);
        }
}
```

Listing 16: Begin Function and OnRenderImage Function

6.5 The Ray Marching Shader

The `RayMarching` compute shader will have the Ray Marching shader code.

6.5.1 Initializing the Shader Parameters

The following code block initializes all the parameters required for the shader, There are values taken from the `FluidRayMarching` script which contains the Source ,Destination and Depth textures and the camera world coordinates.

The Particle structure defines all the properties of the particles here (This is similar to the `SPHCompute` shader). Then there is a particles buffer and all other values.

```
#pragma kernel CSMain

Texture2D<float4> Source;
RWTexture2D<float4> Destination;
Texture2D<float4> _DepthTexture;

float4x4 _CameraToWorld;
float4x4 _CameraInverseProjection;

static const float maxDst = 80;
static const float epsilon = 0.001f;
static const float shadowBias = epsilon * 50;

struct Particle
{
    float pressure;
    float density;
    float3 currentForce;
    float3 velocity;
    float3 position;
};

StructuredBuffer<Particle> particles;
int numParticles;
float particleRadius;
float blendStrength;
float3 waterColor;
float3 _Light;
float3 _AmbientLight;
float3 _CameraPos;
```

Listing 17: Particle struct shader

6.5.2 Ray Initialization

The `Ray` structure is defining a Ray with two attributes origin and direction, the `CreateRay` function then creates a Ray object which it adds the origin and direction attributes to it.

The `CreateCameraRay` takes a `uv` coordinate and transforms the camera world space coordinates into xyz coordinates, the same it does for directions as well. It then adds that to the variable and calls the `CreateRay` function which then returns the ray from the camera.

The `SphereDistance` just calculates from the ray point to the particle center and subtracts the radius of sphere so when it is positive it is outside of the sphere, zero then on the surface and inside the sphere when negative.

```
struct Ray {
    float3 origin;
    float3 direction;
};

float SphereDistance(float3 eye, float3 centre, float radius) {
    return distance(eye, centre) - radius;
}

Ray CreateRay(float3 origin, float3 direction) {
    Ray ray;
    ray.origin = origin;
    ray.direction = direction;
    return ray;
}

Ray CreateCameraRay(float2 uv) {
    float3 origin = mul(_CameraToWorld, float4(0,0,0,1)).xyz;
    float3 direction = mul(_CameraInverseProjection, float4(uv,0,1)).xyz;
    direction = mul(_CameraToWorld, float4(direction,0)).xyz;
    direction = normalize(direction);
    return CreateRay(origin,direction);
}
```

Listing 18: Ray and SphereDistance

6.5.3 Smooth Minimum

The smooth minimum (*smin*) function is a mathematical smoothing operation that is used in raymarching and fluid/particle blending. The formula of the smin function goes like this

$$smin(a, b, k) = lerp(b, a, h) - k \cdot h(1 - h), h = clamp(0.5 + 0.5 * (b - a)/k, 0, 1) \quad (31)$$

With this formula we can basically blend two objects smoothly which is what the `Blend` function does. The `Combine` function takes two colors and two distances and passes them in the `Blend` function which uses *smin* to blend the colors together.

The `GetSphereDistance` function returns the distance from the particle to the eye which is the camera.

```
// polynomial smooth min (k = 0.1);
// from https://www.iquilezles.org/www/articles/smin/smin.htm
float4 Blend( float a, float b, float3 colA, float3 colB, float k )
{
    float h = clamp( 0.5+0.5*(b-a)/k, 0.0, 1.0 );
    float blendDst = lerp( b, a, h ) - k*h*(1.0-h);
    float3 blendCol = lerp(colB,colA,h);
    return float4(blendCol, blendDst);
}

float4 Combine(float dstA, float dstB, float3 colourA, float3 colourB) {
    float dst = dstA;
    float3 colour = colourA;
    float4 blend = Blend(dstA,dstB,colourA,colourB, blendStrength);
    dst = blend.w;
    colour = blend.xyz;
    return float4(colour,dst);
}

float GetShapeDistance(Particle particle, float3 eye) {

    return SphereDistance(eye, particle.position, particleRadius);
    return maxDst;
}
```

Listing 19: smin function

6.5.4 Calculating Scene Info and Normals

The `SceneInfo` function evalutes the whole particle-based fluid scene at a give 3D point, and return the close surface's signed distance + color. Which is uses the Combine function to get a smin of the particles which is then done to every particle in the scene.

The `EstimateNormal` function estimates the normal vector to the given surface at a given point. It does this using central differences: the signed distance field is sampled at small offsets(eplision) along the x, y and z axes, and the gradient of these differences is normalized to obtain the surface normal. This is essential for shading, as normals are used in lighting calculations.

```
float4 SceneInfo(float3 eye) {
    float globalDst = maxDst;
    float3 globalColour = waterColor;

    for (int i = 0; i < numParticles; i ++) {
        Particle particle = particles[i];

        float localDst = GetShapeDistance(particle,eye);
        float3 localColour = waterColor;


        float4 globalCombined = Combine(globalDst, localDst, globalColour,
            localColour);
        globalColour = globalCombined.xyz;
        globalDst = globalCombined.w;
    }

    return float4(globalColour, globalDst);
}

float3 EstimateNormal(float3 p) {
    float x = SceneInfo(float3(p.x+epsilon,p.y,p.z)).w - SceneInfo(float3(p.x-
        epsilon,p.y,p.z)).w;
    float y = SceneInfo(float3(p.x,p.y+epsilon,p.z)).w - SceneInfo(float3(p.x,
        p.y-epsilon,p.z)).w;
    float z = SceneInfo(float3(p.x,p.y,p.z+epsilon)).w - SceneInfo(float3(p.x,
        p.y,p.z-epsilon)).w;
    return normalize(float3(x,y,z));
}
```

Listing 20: SceneInfo and EstimateNormal

6.5.5 Calculating Shadow and Depth

The `CalculateShadow` function calculates the shadow based on epsilon, if the distance is less than the epsilon which means the ray has hit the surface early then shadow intensity is set to low. If the distance of the ray is travelled more than the epsilon we then use this formula `shadowIntensity + (1-shadowIntensity) * brightness` to get the density.

The `LinearEyeDepth` function basically calculates all the depth for various graphics API (OPENGL, Vulkan, DirectX).

```
float CalculateShadow(Ray ray, float dstToShadePoint) {
    float rayDst = 0;
    int marchSteps = 0;
    float shadowIntensity = .2;
    float brightness = 1;

    while (rayDst < dstToShadePoint) {
        marchSteps ++;
        float4 sceneInfo = SceneInfo(ray.origin);
        float dst = sceneInfo.w;

        if (dst <= epsilon) {
            return shadowIntensity;
        }

        brightness = min(brightness,dst*200);

        ray.origin += ray.direction * dst;
        rayDst += dst;
    }
    return shadowIntensity + (1-shadowIntensity) * brightness;
}

float LinearEyeDepth( float rawdepth )
{
    float _NearClip = 0.3;
    float FarClip = 1000;
    float x, y, z, w;
    #if SHADER_API_GLES3 // insted of UNITY_REVERSED_Z
        x = -1.0 + _NearClip/ FarClip;
        y = 1;
        z = x / _NearClip;
        w = 1 / _NearClip;
    #else
        x = 1.0 - _NearClip/ FarClip;
        y = _NearClip / FarClip;
        z = x / _NearClip;
        w = y / _NearClip;
    #endif

    return 1.0 / (z * rawdepth + w);
}
```

Listing 21: CalulateShadow and LinearEyeDepth

6.5.6 Compute Shader Main Loop

- The `CSMain` kernel is the main rendering function of the compute shader, responsible for simulating the fluid surface.
- For each pixel, a ray is generated from the camera through that pixel and marched into the scene using sphere tracing.
- The `SceneInfo` function is called at each step to compute the signed distance to the closest surface.
- If the ray intersects the surface, the intersection point is calculated and the `EstimateNormal` function is used to approximate the surface normal via finite differences.
- The surface normal is then used in a Phong-inspired lighting model that combines ambient, diffuse, and specular lighting to shade the fluid.
- A refraction effect is applied by bending the viewing direction through the surface and sampling the background image, which is blended with the fluid's color.
- The final shaded color is written to the output texture, producing the rendered image of the fluid with lighting and transparency effects.

```
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint width,height;
    Destination.GetDimensions(width, height);

    Destination[id.xy] = Source[id.xy];

    float2 uv = id.xy / float2(width,height) * 2 - 1;
    float rayDst = 0;

    Ray ray = CreateCameraRay(uv);
    int marchSteps = 0;

    float depth = LinearEyeDepth(_DepthTexture[id.xy]);

    while (rayDst < maxDst) {
        marchSteps ++;
        float4 sceneInfo = SceneInfo(ray.origin);
        float dst = sceneInfo.w;

        if (rayDst >= depth) {
            Destination[id.xy] = Source[id.xy];
            break;
        }

        if (dst <= epsilon) {
            float3 pointOnSurface = ray.origin + ray.direction * dst;
```

```
            float3 normal = EstimateNormal(pointOnSurface - ray.direction *
                epsilon);
            float3 lightDir = -_Light;
            float lighting = saturate(saturate(dot(normal,lightDir))) ;

            float3 reflectDir = reflect(-lightDir, normal);
            float spec = pow(max(dot(ray.direction, reflectDir), 0.0), 32);
            float3 specular = 0.7 * spec * float3(1,1,1);

            float3 col = sceneInfo.xyz;

            float3 t1 = cross(normal, float3(0,0,1));
            float3 t2 = cross(normal, float3(0,1,0));
            float3 tangent = float3(0,0,0);
            if (length(t1) > length(t2)) {
                tangent = normalize(t1);
            }
            else {
                tangent = normalize(t2);
            }

            float3x3 tangentMatrix = float3x3(tangent,cross(tangent, normal),
                normal);

            float3 viewDir = normalize(pointOnSurface-_CameraPos);

            float3 refracted = mul(tangentMatrix, refract(viewDir, normal,1));


            Destination[id.xy] = float4(lerp(col, Source[id.xy+(refracted.xy)
                ], 0.8) * (specular + _AmbientLight + lighting * 0.01),1);


            break;
        }

        ray.origin += ray.direction * dst;
        rayDst += dst;
    }
}
```

Listing 22: CSMain Kernel

7. Results

The implementation of the raymarching-based particle fluid simulation produced a highly convincing and realistic fluid-like behavior. The particles were successfully blended together into a smooth surface using signed distance functions, and the marching cubes approximation created continuous fluid geometry. When rendered, the surface exhibited refraction, specular reflections, and lighting effects that closely resembled the optical properties of real water. The shading model, which incorporated both diffuse and specular components, added depth and realism to the visualization, while the refraction effect allowed the background scene to be distorted through the liquid in a physically plausible way. Furthermore, the simulation demonstrated the ability to interact naturally with objects in the surrounding scene, such as colliding against solid geometry or responding to environmental changes. Overall, the results validate the approach by combining physically inspired particle simulation with advanced rendering techniques, producing a visually appealing and interactive fluid representation.

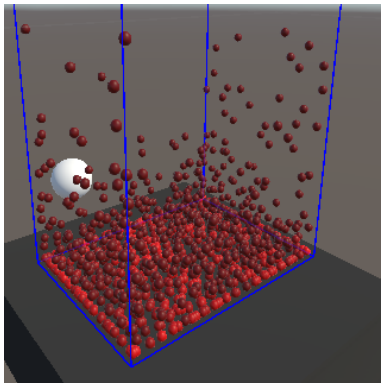

Figure 6: Particles Fluids

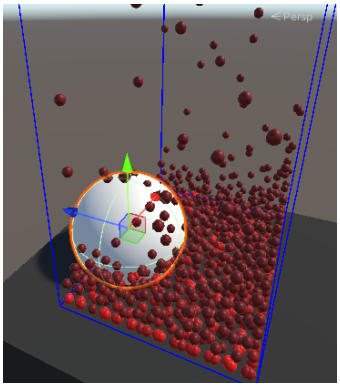

Figure 7: Fluid particles reacting with a sphere object

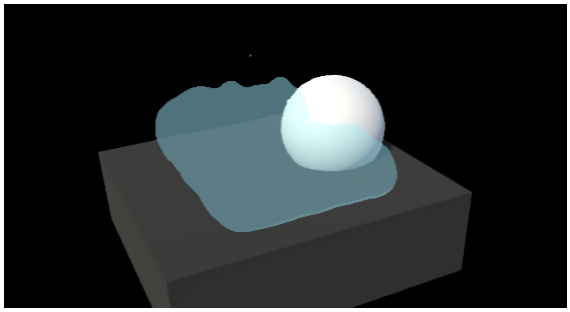

Figure 8: Fluid Rendering

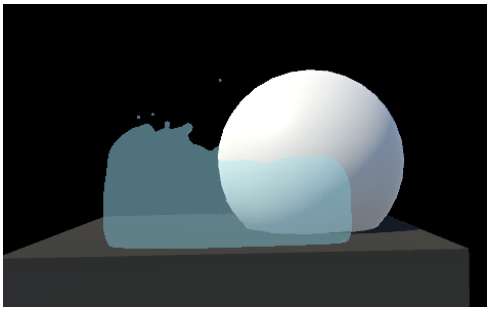

Figure 9: Fluid Rendering reacting to a sphere object

8. Discussion

The results obtained from the implementation of the fluid simulation demonstrate that the system is capable of producing visually realistic fluid-like behavior using a combination of Smoothed Particle Hydrodynamics (SPH) for particle dynamics and raymarching for surface rendering. The simulation not only generates a continuous and cohesive water-like surface but also allows for meaningful interaction with surrounding objects, which enhances the realism and applicability of the method in interactive environments such as games or virtual reality. Compared to traditional grid-based fluid solvers, this approach provides a more flexible and visually appealing representation of fluids, especially for dynamic particle systems, although it sacrifices some degree of physical accuracy in exchange for performance and rendering quality. One of the key strengths of this approach lies in its ability to balance computational efficiency with visual realism, leveraging GPU compute shaders to handle both the particle-based simulation and the raymarched surface efficiently. However, certain limitations remain evident. At higher particle counts, the computational cost increases significantly, leading to reduced frame rates, and the simplification of fluid properties such as surface tension and viscosity means that the simulation does not perfectly replicate real-world fluid dynamics. Despite these limitations, the work highlights the potential of combining SPH with raymarching as a practical approach to real-time fluid rendering. Future improvements could include the integration of more advanced physical models, optimizations for large-scale particle systems, and enhancements to lighting and refraction effects to increase realism. Overall, the discussion suggests that the presented method represents a promising direction for achieving realistic yet computationally efficient fluid simulations in real-time applications.

9. References

[1] Matthias Müller, David Charypar, and Markus Gross. *Particle-Based Fluid Simulation for Interactive Applications.*

Available at: https://matthias-research.github.io/pages/publications/sca03.pdf

[2] Inigo Quilez *Smooth Minimum.*

Available at: https://iquilezles.org/articles/smin/

[3] AJTech *Coding a Realtime Fluid Simulation in Unity*

Available at: https://www.youtube.com/watch?v=zbBwKMRyavE

A. Appendix

The code is unoptimized I tried to implement a basic randering ray marching and a basic SPH fluid simulation, there are ways in which this can be improved.

1. **Efficient Neighbour Search:**

Instaed of checking all the particles against each other, using a grid based spatial partitioning system to determine the neighbour particles instead of going over all the particles in the simulation and using Bitonic mergesort to sort them and use the memory effciently. Because the current algorithm is $O(n^2)$

2. **Ray Marching Optimization:**

The ray marching currently evalutes all particles on the screen, calculating rays for each and every particle is quite expensive. Ideally, the function should be optimized to consider only a subset of particles that significantly affect the rendered image. The current algorihm is $O(n)$, and further optimizations techniques remain to be explored.