Real-Time SPH Fluid Simulation and SDF Raymarching in Unity URP

Xuwen Dong
Bournemouth University / NCCA
UK
s583453@bournemouth.ac.uk

Abstract

This report presents a Unity URP implementation of Smoothed Particle Hydrodynamics (SPH). The project was used to study particle-based fluid simulation, GPU compute shaders, instanced rendering, and SDF raymarching for real-time fluid visualisation.

1 Introduction

Fluid simulation is widely used in visual effects, games, interactive installations, and scientific visualisation. This project focuses on learning the basic principle of SPH and understanding how a particle simulation can be implemented and visualised in Unity (version6000.4.6f1). Instead of representing liquid as a fixed mesh, the fluid is represented as a group of moving particles. Each particle stores physical data such as position, velocity, density, and pressure.

The project also explores how Unity GPU instancing works. The same particle data used by the simulation is read by a grid particle shader to draw many debug spheres, and by a raymarching shader to create a smoother liquid surface. This makes the project useful not only as a fluid simulation exercise, but also as a practical study of data sharing between C# scripts, compute shaders, and rendering shaders.

The project repository is available at: https://github.com/adalek/My-SPH. The implementation was developed as a Unity 6 URP migration and extension of AJTech2002's SPH base project [1].

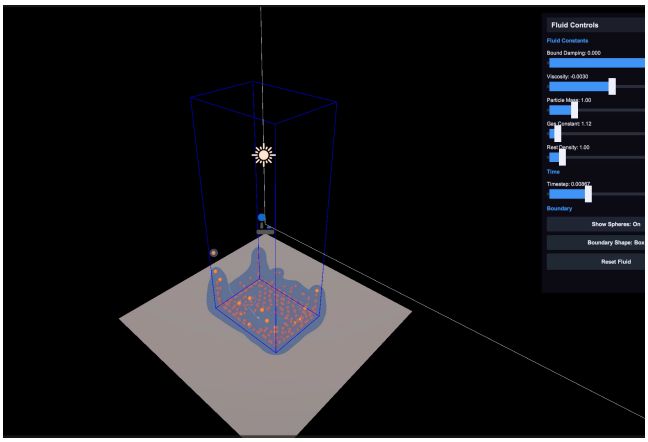

Figure 1: Final interactive SPH fluid simulation in Unity URP with the raymarched fluid visualisation enabled.

2 Related Background Knowledge

2.1 Why SPH is Used

Fluid can be simulated in two main ways: Eulerian and Lagrangian. An Eulerian method observes the fluid through fixed positions in space, usually using a grid. This is stable and widely used for smoke, pressure projection, and large volumetric effects. A Lagrangian method follows moving material points. SPH belongs to the Lagrangian family because the fluid is represented by particles that carry mass, position, velocity, density, and pressure.

SPH is useful for this project because liquids often have free surfaces, splashes, and changing shapes. Since particles move with the fluid, the free surface appears naturally from the particle distribution. This avoids the need to constantly update complex mesh topology. The trade-off is that particle methods can be noisy, compressible, and expensive if every particle checks every other particle.

Table 1: Eulerian and Lagrangian fluid descriptions.

Method	Description and typical use
Grid / Eulerian	Fixed space samples; stable for smoke and pressure solves.
Particle / Lagrangian	Moving material samples; suitable for free-surface liquids.
SPH	Kernel-smoothed particles; useful for interactive water, slime, and splashes.

2.2 Navier–Stokes and SPH

The Navier–Stokes equations describe the motion of fluids. In simplified form, the acceleration of fluid is affected by pressure, viscosity, and external forces:

$$\rho \frac{D\mathbf{u}}{Dt} = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f}_{ext}. \tag{1}$$

In this equation, $\mathbf{u}$ is velocity, p is pressure, μ is viscosity, and $\mathbf{f}_{ext}$ includes forces such as gravity. SPH can be understood as a particle discretisation of this equation. Instead of solving values on a grid, SPH estimates density, pressure, and force by looking at nearby particles within a smoothing radius.

2.3 Incompressibility

Real water is approximately incompressible, meaning its volume should remain almost constant. In continuous fluid mechanics this is often written as:

$$\nabla \cdot \mathbf{u} = 0. \tag{2}$$

This means the velocity field has no net local expansion or compression. In SPH, incompressibility is usually encouraged indirectly

through density conservation. If the local density becomes larger than the rest density, pressure increases and pushes particles apart. This project uses a weakly compressible SPH approach, so density can vary slightly. This is less accurate than a full incompressible pressure solver, but it is simpler and suitable for real-time interaction.

2.4 Kernel Functions

SPH uses smoothing kernels to approximate continuous fluid fields from discrete particles. A kernel function $W(\mathbf{x}_i - \mathbf{x}_j, h)$ gives a weight based on distance, where h is the influence radius. Nearby particles contribute strongly, while particles outside the radius contribute nothing.

This weighted interpolation is what turns separate particles into an approximation of a continuous fluid. The kernel is normalised over its support region, so density can be estimated directly as a weighted mass sum. For other field quantities and force terms, SPH often still uses m_j/ρ_j as the neighbour particle's volume weight.

3 Literature Review and Broader Context

Fluid simulation methods are commonly divided into grid-based and particle-based approaches. Grid methods store fluid values in fixed cells, which is useful for stable pressure solving and smoke simulation. Particle methods move with the fluid, so they are more natural for free-surface liquid effects such as splashes and separated droplets.

SPH is a particle-based method. It can be understood as a numerical discretisation of the Navier–Stokes equations, which describe conservation of momentum in a fluid. In a grid solver the equations are discretised over cells; in SPH they are discretised over moving particles. This project follows that idea by estimating density, pressure, and force from neighbouring particles, then updating velocity and position each frame.

The main limitation of this simple SPH implementation is cost. Each particle compares itself with every other particle, so the neighbour search is $O(n^2)$. This is acceptable for a small learning project, but larger systems usually need spatial partitioning. For rendering, the raw particles are useful for debugging, while the raymarching pass blends them into a smoother SDF fluid surface [2].

3.1 GPU Parallelism

SPH is well suited to GPU compute shaders because every particle performs a similar calculation. In this project, one GPU thread is used to calculate one particle in each simulation pass.

Listing 1: Thread group size used in the compute shader

```
[numthreads(100, 1, 1)]
void ComputeForces(uint3 id : SV_DISPATCHTHREADID)
{
    // id.x is the particle index for this thread
}
```

The value `100` is not the number of GPU cores. It means one compute shader thread group contains 100 threads. The C# script then dispatches enough groups to cover all particles:

Listing 2: Dispatching enough groups for all particles

```
int threadGroups = Mathf.CeilToInt(particleCount / 100f);
shader.Dispatch(kernel, threadGroups, 1, 1);
```

For example, with 1000 particles, Unity dispatches 10 groups. Each group contains 100 threads, so 1000 GPU threads are launched in total. The GPU decides internally how many groups can run at the same time; the program does not manually assign work to a fixed number of cores.

The group size is a performance choice rather than a physics setting. Changing it should not change the simulation result if the dispatch count and bounds checks are updated correctly. Common values are 64, 128, or 256, but the best value depends on the target GPU and should be tested with the Unity Profiler.

This parallelism helps, but the current solver is still expensive. Each thread calculates one particle, but inside that thread it still loops over all particles to find neighbours. Therefore 1000 particles can still produce about one million neighbour checks per kernel.

4 Technical Overview

The project is divided into three connected systems built around Unity's Universal Render Pipeline [4]:

- **Simulation:** `SPH_Compute.cs` creates GPU buffers and dispatches `SPHComputeShader.compute`.
- **Debug rendering:** `GridParticle.shader` reads the particle buffer and draws one sphere instance per particle.
- **Fluid surface rendering:** `Raymarching.compute` reads the same particle buffer and blends particles into a continuous SDF surface.

4.1 File Structure and Data Flow

The important project files are organised around one shared GPU particle buffer. The C# scripts create and bind the data, while the shaders update or render it.

Listing 3: Main SPH project file structure

```
Assets/SPH/
  Scripts/
    SPH_Compute.cs
      Creates particles, owns ComputeBuffer,
      dispatches SPH kernels, draws debug spheres.

    FluidRayMarching.cs
      Provides camera, light, colour, and particle
          buffer
      references for the raymarching pass.

    FluidRayMarchingFeature.cs
      URP renderer feature. Runs Raymarching.
          compute
      as a screen-space render pass.

    FluidRuntimeControlUI.cs
      Runtime UI for timestep, gas constant,
          viscosity,
      damping, boundary shape, reset, and sphere
          preview.

  Shaders/
```

```
    SPHComputeShader.compute
      Updates density, pressure, force, velocity,
          position.

    GridParticle.shader
      Reads particle buffer and draws one sphere
          per particle.

    Raymarching.compute
      Reads particle buffer and blends particles
          into SDF fluid.
```

Listing 4: Shared data workflow between scripts and shaders

```
SPH_Compute.cs
  creates Particle[] on CPU
  uploads it to GPU ComputeBuffer:
      _particlesBuffer
        |
        v
SPHComputeShader.compute
  ComputeDensityPressure  -> writes density and
      pressure
  ComputeForces           -> writes currentForce
  Integrate               -> writes position and
      velocity
        |
        v
same _particlesBuffer, now updated on GPU
        |
        +--> GridParticle.shader
        |      instanceID selects particle index
        |      draws sphere at particle.position
        |
        +--> FluidRayMarchingFeature.cs
               dispatches Raymarching.compute
               Raymarching.compute samples all
                   particles
               builds smooth SDF liquid surface on
                    screen
```

The key point is that particle data is not copied back to the CPU every frame. The simulation and rendering stages share the same GPU buffer, so the compute shader can update particles and the rendering shaders can immediately read the latest positions.

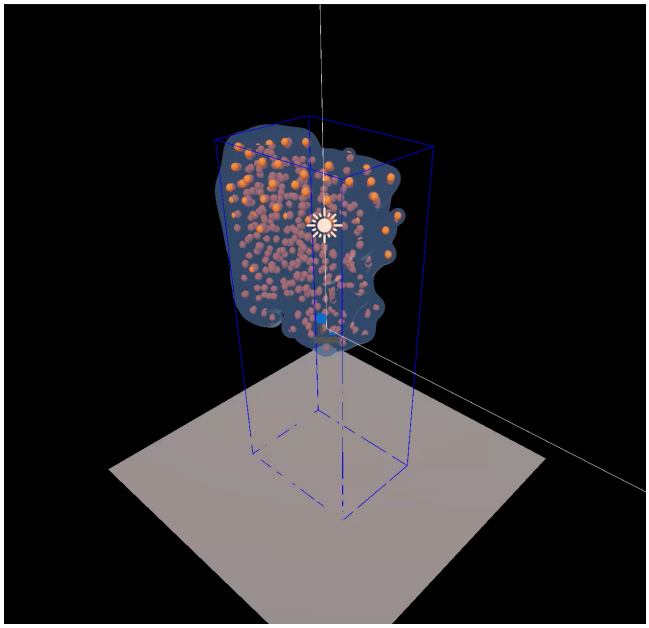

Figure 2: Initial particle layout at the beginning of the simulation. The particles are created by C# and then uploaded to GPU buffers for compute shader simulation.

5 Use of Mathematics and Physical Laws

Each particle stores position, velocity, force, density, and pressure. The solver uses a simplified SPH model based on local particle neighbourhoods.

5.1 Density Estimation

The density at particle i is estimated by summing nearby particle masses weighted by a smoothing kernel:

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h), \tag{3}$$

where ρ_i is density, m_j is particle mass, W is the smoothing kernel, and h is the smoothing radius. This is a kernel-weighted mass summation. It approximates the continuous density field using nearby discrete particles.

5.2 Equation of State

Pressure is computed from the difference between current density and rest density:

$$p_i = k(\rho_i - \rho_0), \tag{4}$$

where k is the gas constant and ρ_0 is the rest density. This is a weakly compressible SPH model: density deviation generates pressure. A larger gas constant makes the fluid resist compression more strongly, but may introduce instability.

5.3 Pressure Force

The pressure term from Navier–Stokes is discretised as a particle-to-particle force. It pushes particles away from high-density regions:

$$\mathbf{f}_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{5}$$

5.4 Viscosity Force

Viscosity smooths relative velocity differences:

$$\mathbf{f}_i^{viscosity} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h). \tag{6}$$

This term is visually important because it controls whether the fluid feels watery, slimy, or thick. A higher viscosity can create gooey or lava-lamp-like motion, while lower viscosity makes particles separate and flow more freely.

5.5 External Force

Gravity is added as an external force:

$$\mathbf{f}_i^{gravity} = \rho_i \mathbf{g}. \tag{7}$$

5.6 Integration

Particle velocity and position are advanced using an explicit time step:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \Delta t \frac{\mathbf{f}_i}{\rho_i}, \tag{8}$$

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t). \tag{9}$$

This method is simple and fast, but it requires careful tuning of timestep, gas constant, viscosity, particle mass, and boundary damping.

6 Implementation

6.1 GPU Particle Data

Particles are created in C# and uploaded to a structured GPU buffer. The same buffer is shared by simulation and rendering passes.

Listing 5: Simplified particle structure used by C# and HLSL

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

The compute shader updates particles in parallel. Each GPU thread uses its dispatch ID as the particle index:

Listing 6: GPU thread index maps directly to a particle index

```
uint id = dispatchThreadID.x;
Particle particle = particles[id];
```

6.2 Compute Shader Passes

The simulation is split into three kernels:

- `ComputeDensityPressure`: calculates density and pressure.
- `ComputeForces`: calculates pressure, viscosity, and gravity forces.
- `Integrate`: updates velocity and position, then resolves boundary collisions.

This structure keeps the data flow clear and avoids reading partially updated values during the same stage.

6.3 Force and Velocity Calculation

In the project, force is calculated in `SPHComputeShader.compute` by accumulating pressure, viscosity, gravity, and a simple obstacle collision force. The pressure term pushes particles away from dense areas, while viscosity reduces relative velocity differences between neighbouring particles.

Listing 7: Simplified force calculation from the project

```
pressure = float3(0, 0, 0);
visc = float3(0, 0, 0);

for each particle j:
    if j is the same particle:
        continue

    dist = distance(particle[i].position, particle
        [j].position);

    if dist < radius * 2:
        dir = normalize(particle[i].position -
            particle[j].position);

        pressure += mass^2 *
            (particle[i].pressure / particle[i].
                density^2 +
             particle[j].pressure / particle[j].
                 density^2) *
            SpikyKernelGradient(dist, dir);

        visc += viscosity * mass^2 *
            (particle[j].velocity - particle[i].
                velocity) /
            particle[j].density *
            SpikyKernelSecondDerivative(dist);

force = gravity - pressure + visc;
```

Velocity and position are then updated in the `Integrate` kernel:

Listing 8: Simplified velocity and position update

```
velocity = velocity + (force / particleMass) *
    timestep;
position = position + velocity * timestep;

if position is outside the selected boundary:
    position = clamped boundary position;
    velocity = reflect(velocity, boundaryNormal) *
         abs(boundDamping);
```

This means the timestep directly controls how far the simulation advances per update. A larger timestep makes the fluid move faster, but it can also make the simulation unstable.

6.4 Boundary Shapes

The fluid can collide with several container shapes: box, sphere, and cylinder. The shape is selected at runtime through the UI and sent to the compute shader as an integer enum. Boundary tests are performed in container local space, then collision normals are converted back into world space.

Listing 9: Simplified boundary shape selection

```
if (boundaryShape == 0)
    ResolveBoxBoundary(position, velocity);
else if (boundaryShape == 1)
    ResolveSphereBoundary(position, velocity);
else
    ResolveCylinderBoundary(position, velocity);
```

6.5 Debug Sphere Rendering

`GridParticle.shader` uses GPU instancing. Unity provides an instance ID for each sphere, and the shader uses this ID to read the matching particle from the GPU buffer:

Listing 10: Instance ID selects the matching particle

```
Particle p = particlesBuffer[instanceID];
float3 worldPosition = p.position;
```

This makes particle preview efficient because thousands of spheres can be drawn with one indirect draw call.

6.6 SDF Raymarching

For the final liquid surface, each particle is treated as a signed distance sphere:

$$d_i(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_i|| - r. \tag{10}$$

Multiple particle SDFs are blended using a smooth minimum function:

$$d(\mathbf{x}) = smin(d_1, d_2, ..., d_n). \tag{11}$$

The raymarching pass samples along camera rays until the SDF surface is hit. Lighting and refraction are then applied in screen space.

7 Runtime Controls

A runtime UI was added to support testing and demonstration. It exposes:

- boundary damping;
- viscosity;
- particle mass;
- gas constant;
- rest density;
- timestep;
- boundary shape;
- sphere debug visibility;
- reset simulation.

This allows parameters to be adjusted during play mode.

8 Results

The implemented system demonstrates interactive particle-based fluid behaviour in Unity URP. The debug sphere view confirms that the SPH particles move, collide with the container, and respond to parameter changes. The raymarching view produces a smoother continuous liquid-like surface by blending particle SDFs.

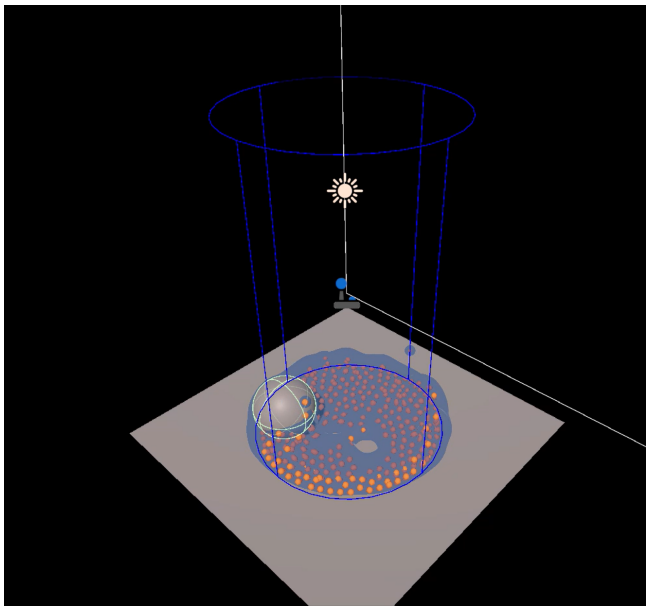

Figure 3: Collision test showing the simulated fluid interacting with a spherical obstacle.

The main achievements are:

- GPU-based SPH simulation using compute shaders;
- URP-compatible rendering and raymarching pass;
- runtime parameter control for testing;
- support for multiple container shapes;
- visual comparison between raw particles and smooth fluid surface.

9 Limitations and Future Work

The current implementation has several limitations, and these also suggest useful directions for future work:

- The SPH solver uses brute-force neighbour search, so performance scales poorly with particle count. A spatial hash or grid-based neighbour search would make the simulation more scalable.
- The raymarching pass loops through all particles, which is expensive at high resolution. Particle culling or adaptive raymarching could reduce this cost.
- Boundary shapes are static. A future script could add moving container physics with wall velocity and angular velocity.
- The surface is screen-space/SDF based rather than a physically reconstructed mesh. Improved surface reconstruction and water shading could make the result look more realistic.

Even with these limitations, the project is still useful as a learning tool for SPH, Unity compute shaders, GPU instancing, and stylised real-time fluid effects.

10 AI Usage

ChatGPT by OpenAI was used as an assistive tool for planning, code explanation, report structure, and LaTeX editing [3]. The implementation decisions, testing, parameter choices, and final evaluation remain the responsibility of the author.

References

[1] AJTech2002. Smoothed particle hydrodynamics: youtube-base branch. https://github.com/AJTech2002/Smoothed-Particle-Hydrodynamics/tree/youtube-base, 2024. Base Unity SPH project used as the starting point for this migration and extension.

[2] Simon Green. Screen space fluid rendering for games. In *Game Developers Conference*, 2010.

[3] OpenAI. Chatgpt. https://chat.openai.com/, 2026. AI assistance used for planning, explanation, and report editing.

[4] Unity Technologies. Universal render pipeline documentation. https://docs.unity3d.com/Manual/universal-render-pipeline.html, 2026. Accessed 2026-05-27.