Smoothed Particle Hydrodynamics for Procedural Fluid Simulation in Houdini

Caia Gelli

Advisor: Adam Mally

University of Pennsylvania

ABSTRACT

This project aims to develop a particle-based fluid simulation in Houdini, leveraging Smoothed Particle Hydrodynamics (SPH) principles for realistic fluid motion and particle interactions. Using Houdini's POP network, the simulation will model fluid liquid dynamics through force calculations, density estimation, and pressure-based interactions for up to 3000 particles.

Git Repo: https://github.com/CaiaG/SeniorDesignFluidSim.git

1. INTRODUCTION

Fluid simulations are widely used in computer graphics, games, and visual effects for various types of animations. However, simulating fluid behavior accurately in a digital environment is a complex challenge, requiring computationally efficient methods to balance realism and performance. Traditional Computational Fluid Dynamics involves solving the Navier-Stokes equations directly, which is computationally expensive and unnecessary for CGI fluid dynamics. Thus, the computer graphics community has been continuously developing smoothed particle hydrodynamic (SPH) principles for fluid simulation as a practical and flexible solution.

Houdini is a 3D procedural software tool developed by SideFx that offers a wide range of capabilities, including particle systems. The program's workflow is primarily node-based, with an additional platform to fine tune details using its shader language VEX. Houdini's primary tool for fluid particle systems is the FLIP solver, a hybrid system of particle based and volume-based fluid simulations and a faster alternative to SPH. It is an incredibly versatile and robust system. The ease of use leaves a gap between the user and understanding the mechanics behind fluid simulation. This project seeks to address this gap by providing a hands-on, code-driven implementation from a beginner's perspective to explore some fundamental concepts behind fluid simulation.

Understanding how to implement a custom fluid solver within Houdini is valuable both for improving artistic control over simulations and for learning procedural techniques that are applicable to other simulation problems. This project explores a particle-based fluid simulation using SPH principles within Houdini, aiming to create a physically believable small-scale simulation.

The expected contributions of this work include:

- A particle-based solver tailored towards generic fluid simulation
- A structured approach to implementing a 3D simulation with procedural river generation.

This project aims to contribute to an anthology of resources to advance the technical understanding of fluid simulations.

1.1 Design Goals

This project is intended for users with minimal to no experience in the technical aspects of fluid simulations. The goal is to provide an intuitive and digestible approach to understanding the topic and application of SPH. The simulation will be designed with clear, modular components, allowing users to understand the software pipeline without prior knowledge of fluid dynamics.

1.2 Projects Proposed Features and Functionality

Features and Functionality

- Custom SPH-Based solver: Implementing a robust solver tailored for procedural water simulation.
- Particle-Based Fluid Interaction: Developing realistic fluid behavior through particle-particle and particle-boundary interactions.
- Procedural Terrain Adaptation: Ensuring the river conforms to non-uniform landscapes.
- Comprehensive Documentation & Learning Resources: Providing a well-documented guide on fluid simulation principles in Houdini.

2. Related Work

Smoothed Particle Hydrodynamics (SPH) has been widely explored in computer graphics, physics-based animation, and real-time applications. This section categorizes prior work into three key areas.

2.1 SPH Foundations & Theoretical Background

Sources:

SPH Tutorial A Physics-Based Introduction - provides a step-by-step introduction to SPH equations, focusing on density estimation, pressure forces, and viscosity handling. [KBST19].

Smoothed Particle Hydrodynamics: Theory, Implementation, and Application to Toy Stars – SPH and how it can be applied to find the equilibrium state of a toy star model in MATLAB [Moc11].

Fluid Simulation Lecture - covers Navier-Stokes-based solvers and traditional Eulerian grid-based methods, contrasting with Lagrangian-based SPH techniques [BM07].

How My Work Builds on This:

- Implements SPH within Houdini, bridging the gap between theory and real-world applications for artists.

2.2 SPH in Computer Graphics & Real-Time Applications

Sources:

Screen Space Fluid Rendering for Games NVIDIA Direct3D Effects - explores real-time SPH fluid simulation using GPU acceleration [Gre10].

SPH Fluid Simulation on YouTube - demonstrates from scratch system in Unity [Lag23].

Particle Based Fluid Simulation for Interactive Applications – extension and explanation of SPH techniques to highly deformable bodies. Also designs simple but optimal poly6 kernel for SPH calculations [MCG03]

How My Work Builds on This:

- Designed for offline simulation with full procedural terrain integration
- Implements in Houdini, allowing procedural artists to modify behavior dynamically, unlike previous black-box implementations in engines like Unity or NVIDIA's Direct3D.

2.3 Procedural Fluid-Terrain Interactions

Sources:

A Beginner's Guide to Procedural Terrain Modelling Techniques provides an overview of Perlin noise, Voronoi tessellation, and midpoint displacement, commonly used for terrain generation [EPG*19].

How My Work Builds on This:

- Combines SPH fluid simulation with procedural terrain deformation.

3. Project Proposal

For this project I will be using Houdini's POP network system and VEX to directly design a fluid simulation based on SPH. Particle behavior will be defined using Houdini POP wrangle which allows you to control the behavior of a particle at each time step.

3.1 Anticipated Approach

I will first set up a basic particle system to emit from a Grid SOP in 2D. The first particle behavior to define is the force of gravity and collision with the ground plane. I will then use an SPH smoothing kernel to estimate local density before calculating the pressure forces between particles. Next, I will introduce viscosity forces to smooth particle velocities and surface tension to prevent clumping.

Once the solver is stable in 2D I will modify all the former calculations to extend to 3D. The next step will be surface reconstruction using marching cubes or screen space fluid rendering. I will also have to store particle normals for later shading processes.

The final phase of development will surround the procedural river and setting up renders for the final presentation.

3.2 Target Platforms

I will be utilizing Houdini Apprentice 20.5 and a mid-range Nvidia graphics card (1660 Super).

3.3 Evaluation Criteria

To evaluate the efficacy of the system I will Houdini's built in performance monitoring tool. The tool offers a way of measuring computation time for all stages of the pipeline.

4. Project Timeline

Project Milestone Report (Alpha Version)

- Finished readings
- Completed particle initialization
- Basic fluid effects

Project Milestone Report (Beta Version)

- Additional Fluid Effects
- Begun procedural river generation
- Final plan for render

Project Final Deliverables

- Videos of particle behavior
- Documentation of process via research paper

Project Future Tasks

- Extend it to a system that allows for real time interaction with the particles
- Experiment with stylized shaders

5. Method

The solver follows a standard Smoothed-Particle Hydrodynamics (SPH) loop—neighbourhood search, density estimation, pressure/viscosity forces, external forces, and semi-implicit integration—implemented entirely inside a single Houdini POP network.

5.1 Particle Initialization

The simulation is initialized by emitting particles from a Grid SOP using the POP source node. Each particle has the following attributes: position, velocity, mass, local density, pressure and force. An attribute wrangle initializes these values before the first step of the simulation.

The first operation conducted on a particle is neighborhood search using Houdini's built in search function pcopen. With an upper bound for neighbor count, the relevant values (velocity, position, density) are temporarily stored in arrays for easy access.

5.2 Force Integration

To define fluid behavior, we need to apply both external and internal forces to each particle. The first external force introduced is gravity, which is scaled by the estimated density of the particle to maintain consistent mass-based influence.

The total force applied to a particle at each timestep is computed as the sum of:

- Gravity (external)
- Pressure forces (internal, from surrounding particles)
- Viscosity forces (internal, smoothing velocities)
- Surface tension (internal, maintaining fluid cohesion)

Each component contributes to the particle's acceleration via Newton's second law: a = force / mass.

These accelerations are integrated using a forward Euler scheme, updating velocities and positions as outlined below.

```
v@force = force;

vector acceleration = v@force / local_density;

velocity += acceleration * @TimeInc;
```

5.3 Density Estimation

To simulate realistic fluid behavior, each particle must estimate its local density based on the positions of nearby particles. This is the core idea behind SPH: particles influence one another through a smoothing kernel, which defines how much neighboring particles contribute to local quantities like density. The closer a neighbor is, the more of a contribution it will have to the estimated sum.

For each particle *i*, the density is computed as:

$$f = \sum_j m_j * W(r_i - r_j, h)$$

This formula is essentially a weighted sum of each neighbor within the search radius of particle i.

This SPH implementation uses the Poly6 kernel [MCG03] for density estimation.

$$W(r,h) = \frac{315}{128\pi h^9}\begin{cases}(h^2 - r^2)^3 & 0 \le r \le h \\ 0 & otherwise\end{cases}$$

The implementation of the code is as follows:

```
float calc_density(float rest_density, radius; vector position;
int num; float mass; int neighbors[]; vector
neighbor_positions[]; int handle) {

  float density = 0.;
  foreach(int i; int pt; neighbors) {
    vector Pj = neighbor_positions[i];

    float distance = distance(position, Pj);

    density += mass * poly6(distance, radius);  // Accumulate
density
  }

  density = max(density, rest_density / 2);
  density = min(density, rest_density * 1.25);
  return density;
}
```

5.4 Pressure Force Calculation

Once local density is estimated, pressure forces are computed to simulate how fluid resists compression.

We first use Tait's equation of state, which relates the deviation from rest density to a pressure value.

$$P = B\left[\left(\frac{\rho}{\rho_j}\right)^{\gamma}\right] - 1$$

Where ρ_j is the reference density of the fluid, γ is the adiabatic index, and B is a constant that varies depending on the given fluid properties and temperature.

To compute the pressure force on each particle, the SPH formulation considers both the particle's own pressure and its neighbors', using a Spiky kernel gradient for smooth interaction:

$$W(r,h) = \frac{15}{\pi h^6}\begin{cases}(h - r)^3 & 0 \le r \le h \\ 0 & otherwise\end{cases}$$

Where r is the distance between a particular particle p and its neighbor j, and h is the search distance radius.

The equation to calculate the pressure force exerted on a given particle is as follows:

$$f = \sum_j m_j \frac{p_i + p_j}{2\rho_j} W(r_i - r_j, h)$$

This term drives fluid expansion and prevents particles from collapsing too closely under high compression. Here is the implementation below:

```
vector calc_pressure(float pressure, radius, mass; vector
position;
            int neighbors[]; vector neighbor_positions[];
            float neighbor_densities[], neighbor_pressures[]) {

  vector p_force = {0, 0, 0};

  foreach(int i; int pt; neighbors) {
    vector Pj = neighbor_positions[i];
    float density_j = neighbor_densities[i];
    float pressure_j = neighbor_pressures[i];

    vector r_ij = position - Pj;
    float r = length(r_ij);
    vector r_dir = normalize(r_ij);

    if (r > radius || r == 0) continue;

    float W_grad = (-45 / (M_PI * pow(radius, 6))) * pow(radius -
    r, 2);
    vector grad_W = r_dir * W_grad;

    float mean_pressure = (pressure + pressure_j) * 0.5;
    vector pressure_term = mean_pressure * (mass / density_j)
* grad_W;
    p_force += -pressure_term;
  }
  return p_force;
}

float local_density = calc_density(rest_density, radius,
position, @ptnum, mass, neighbors, neighbor_positions,
handle);
f@density = local_density;

float ratio = local_density / rest_density;
float p = B * (pow(ratio, gamma) - 1.0);
float pressure = clamp(p, 0.001, 50000);

f@pressure = pressure;
vector pressure_force = calc_pressure(pressure, radius, mass,
position, neighbors, neighbor_positions, neighbor_densities,
neighbor_pressures);
v@pforce = pressure_force;
force = pressure_force;
```

5.5 Viscosity

Viscosity simulates the internal friction of the fluid, smoothing out particle velocities and preventing erratic motion. In SPH, viscosity is modeled using a force proportional to the relative velocity between neighboring particles.

The standard SPH formulation for viscosity force is:

$$f = \mu \sum_j m_j \frac{v_j - v_i}{2\rho_j} W(r_i - r_j, h)$$

This implementation uses the viscosity kernel proposed by Müller et al. [MCG03], chosen for its stable second derivative.

$$W(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & otherwise \end{cases}$$

Viscosity helps reduce jittering, smooth velocity discontinuities, and improve numerical stability. Pressure forces will **not** stabilize without viscosity.

```
vector calc_viscosity(int neighbors[]; vector
neighbor_positions[], neighbor_velocities[]; float
neighbor_densities[]; float radius; vector position, velocity;
float mass; float mu; int handle) {
  vector viscosity_force = {0, 0, 0};
  foreach(int i; int pt; neighbors) {
    vector Pj = neighbor_positions[i];
    float density_j = neighbor_densities[i];
    vector Vj = neighbor_velocities[i];

    vector r_ij = position - Pj;
    float r = length(r_ij);
    float W_laplacian = viscosity_kernel_laplacian(r, radius);

    if (r > 0 && r < radius) {
      vector force_ij = mu * (mass / density_j) * (Vj - velocity) *
W_laplacian;
      viscosity_force += force_ij;;
    }
  }
  return viscosity_force;
}

vector viscosity_force = calc_viscosity(neighbors,
neighbor_positions, neighbor_velocities, neighbor_densities,
radius, position, velocity, mass, mu, handle);
force += viscosity_force;
```

5.6 Surface Tension

Surface tension simulates the cohesive forces that cause fluids to form smooth surfaces and resist breaking apart. It plays a crucial role in stabilizing the fluid boundary and maintaining droplet-like behavior, especially in low resolution or small scale simulations.

In SPH, surface tension can be modeled in several ways. This implementation uses a simple cohesion-based model where particles near the surface experience an inward directed force, calculated from the imbalance in neighboring particle distribution. A common approach uses a color field gradient, where the color field quantity indicates whether a point is inside or outside the fluid. The gradient of the color field becomes the surface normal.

$$n = \nabla c(r) = \sum_j m_j \frac{1}{\rho_j} W(r_i - r_j, h)$$

We then compute the curvature k as the divergence of the surface normal to track how the surface bends. Thus the full equation for surface tension is:

$$t^{surface} = \sigma k n = \sigma \nabla c^2 \frac{n}{|n|}$$

Where n is the surface normal, σ is the tension coefficient, and k is the curvature at the particle's location.

```
vector compute_surface_norm(int neighbors[]; vector
neighbor_positions[]; float neighbor_densities[]; float mass,
radius; vector position; int handle) {
  vector norm = {0, 0, 0};

  foreach(int i; int pt; neighbors) {
      float density_j = neighbor_densities[i];
      vector Pj = neighbor_positions[i];

      vector r_j = Pj - position;
      float r = length(r_j);
      vector r_dir = normalize(r_j);

      vector W = grad_poly6(r, radius) * r_dir;

      norm += W * mass / density_j;
    }
  return norm;
}

float compute_laplacian_cS(int neighbors[]; vector
neighbor_positions[]; float neighbor_densities[]; float mass,
density, radius; vector position; int handle) {
  float laplacian_cS = 0.0;

  foreach(int i; int pt; neighbors) {
    vector Pj = neighbor_positions[i];
    float density_j = neighbor_densities[i];

    float r = length(Pj - position);
    float W_lap = poly6_laplacian(r, radius);
    laplacian_cS += (mass / density_j) * W_lap;
  }

  return laplacian_cS;
}
```

```
float compute_curvature(int neighbors[]; vector
neighbor_positions[]; float neighbor_densities[]; float mass,
density, radius; vector position, n; int handle) {
  float laplacian_cS = compute_laplacian_cS(neighbors,
neighbor_positions, neighbor_densities, mass, density, radius,
position, handle);
  return -laplacian_cS / max(length(n), 1e-6);
}

vector compute_surface_tension(vector normal; float k) {
  float delta_s = length(normal);
  float sigma = 0.1;

  float threshold = 0.01;
  if (delta_s < threshold) {
    return {0, 0, 0};
  }

  return sigma * k * delta_s * normalize(normal);
}

vector s_normal = compute_surface_norm(neighbors,
neighbor_positions, neighbor_densities, mass, radius, position,
handle);
float k = compute_curvature(neighbors, neighbor_positions,
neighbor_densities, mass, local_density, radius, position,
s_normal, handle);

vector surface_tension = compute_surface_tension(s_normal, k);
force += surface_tension;
```

This simplified model is not physically rigorous but captures the visual essence of fluid cohesion.

5.7 Surface Reconstruction

To visualize the simulated fluid, the particle representation must be converted into a smooth surface mesh. Two methods were tested in Houdini for surface reconstruction: the VDB based approach and the Fluid Mesh DOP network.

- VDB Method: This approach uses the Particle Fluid Surface SOP to convert particles into a volumetric representation, followed by a marching cubes style meshing algorithm. It offers high performance and responsive feedback in the viewport, making it ideal for iterative development and rendering lightweight simulations.
- Fluid Mesh DOP: This node-based approach is more physically grounded, reconstructing the surface using internal pressure fields and neighborhood analysis. It produces greater geometric accuracy, particularly at small scales, but comes at the cost of significantly higher computation time and memory usage.

Ultimately, the VDB method was easier to use for the final render due to its speed and tunability. Parameters like particle separation, influence radius, and isosurface

threshold were adjusted to achieve a clean, cohesive fluid surface. The Fluid Mesh DOP proved a lot more restrictive and prone to making the program crash.

6. Results

The following results evaluate the solver from four complementary angles.

First, a systematic parameter sweep exposes how individual constants (particle mass, kernel radius, viscosity, surface tension, etc.) shape the fluid's behavior. Next, there is a brief discussion of some final renders. We then quantify stability and performance, comparing our SPH approach with Houdini's FLIP solver.

6.1 Parameter Testing

To isolate how each simulation knob affects stability and look, I compare every experiment with a baseline configuration I dubbed the "default" liquid due to its stable and desirable behavior. The parameters for said liquid are defined below:

Parameter	Symbol	Value
Particle count	P	1042
Initial density	ρ	300
Maximum neighbor count	n	80
Particle mass	m	2
Kernel/Search radius	r	0.4
Viscosity coefficient	μ	0.01
Surface tension coefficient	σ	0.1

A small housekeeping rule is applied every timestep: particles with fewer than five neighbors are culled. This helps us keep track of how many particles remain in the active liquid body.

6.1.1 Neighborhood Count

Neighborhood Count	Observations
n = 8	Particles only exhibit vertical movement. Majority of particles exhibit explosive movement and are lost from the scene. Remaining particles are all calculated to have the lower bound value of density. 1061 particles at 15 frames. 1061 particles remaining at 96 frames.
n = 20	The first 24 frames have lots of erratic movement. Still mostly vertical movement although particles also spread out a little. Particles spread into a small circle, not uniformly distributed. Motion stabilizes within 96 frames. All remaining particles have calculated density $\rho < 180$ 1054 particles at 15 frames. 1046 particles at 96 frames.
n = 50	Similar behavior to n = 20. Particles spread into an oval, not a circle. Ripples form in the oval. 1039 particles at 15 frames. 1039 particles at 96 frames.
n = 80 (default)	1042 particles at 15 frames. 1041 particles at 96 frames.
n = 120	Particles spread into a circle. Ripples form in the particle body. Jittery movement through 96 frames. 686 particles at 15 frames. 683 particles at 96 frames.

Takeaways:

Extremely small kernels (n = 8) starve each particle of support, so pressure forces spike, the particles explode upward, and density collapses. Increasing the count to 20–50 quickly dampens those instabilities: the particles still jitter for the first few dozen frames, but the main body settles into a coherent, ripple-bearing disc and retains > 99 % of its mass. The default 80 neighbor window shows similarly good behavior with negligible loss and the smoothest surface. Pushing past 100 neighbors switches the problem the other way: each particle averages so many contributions that density is over-smoothed, forces weaken, and nearly a third of the fluid evaporates from the scene. In short, a mid-range kernel of 50–80 neighbors delivers the best balance of stability and mass conservation for this particle setup, while counts below ~20 or above ~100 are detrimental.

6.1.2 Particle Mass

Mass	Observations

m = 0.1	Very explosive reactions within the first 24 frames. Particles end up with more space in between each other, leading to more dissipation over time. System never reaches equilibrium.
m = 20	Particle system reaches a point of stability within 96 frames. Particles form a uniform shape (circle). No explosive behavior after the first 12 frames.
m = 200	Particles stay very densely packed. Uneven distribution. Particles on the edge of the shape get ejected. System never reaches equilibrium.

Takeaways:

Low mass (m = 0.1) causes the particles to accelerate wildly and spread apart, dissolving the scene overall. Raising mass to a moderate value (m ≈ 20) restores a healthy pressure-density balance: the splash calms in under half a second, settles into a clean circular pool, and remains stable. Pushing mass two orders higher (m = 200) over-amplifies pressure; the core clumps rigidly while edge particles are flung off, preventing equilibrium. Thus, a middle-ground mass (~20 units for this scale) delivers the only configuration that converges, avoiding both runaway expansion and over-compressed breakup.

6.1.3 Neighborhood Search Radius

Kernel radius	Observations
r = 0.1	System does not stabilize. Particles spread uniformly. Position adjustments occur in ripples.
r = 0.2	Similar behavior to r = 0.1. Larger ripples than r = 0.1
r = 0.6	Less spread than previous options. A thick band of particles around the perimeter of the circle. There is also a small gap between the perimeter particles and the rest of the particles. System stabilizes but does not end up uniformly.
r = 0.8	Particles do not spread at all.

Takeaways:

When the support radius is tiny (r = 0.1–0.2 m), each particle samples only a sliver of neighbors; pressure waves travel as visible ripples and the pack never fully settles. Expanding the radius to r ≈ 0.6 m supplies enough overlap for a stable solve, but the surplus interaction over-binds the perimeter, leaving a dense outer ring and a void just inside it—evidence of over-smoothed density. Pushing still further (r = 0.8 m) locks the particles together so tightly that the blob hardly spreads at all, behaving more like jelly than water. Optimal behavior sits between these extremes—large enough to damp ripple-driven chaos, yet small enough to avoid perimeter clumping—suggesting an operating window of roughly 0.3 m ≤ r ≤ 0.5 m for this scene scale.

6.1.4 Viscosity Coefficient

Coefficient value	Observations
μ = 0.001	Lots of spraying particle motion for the first 48 frames. Lots of ripples and pops. Non uniform spacing through 96 frames.
μ = 0.1	Similar spraying motion but less ripples and pops. Particles stay drifting gradually through 96 frames.
μ = 0.5	Like previous two μ values, but rippling motions get gentler over time. Spacing of particles is less uniform.
μ = 1.0	The first 24 frames exhibit very explosive behavior. Particles spread out less uniformly. No ripples.

Takeaways:

Very low viscosity (μ ≈ 0.001) offers almost no momentum diffusion—splash crowns spray high, ripples snap back, and the system appears extremely disordered. Raising μ to 0.1–0.5 damps the high frequency pops; the spray still occurs but gradually calms, though particle spacing grows patchy as shear is over-smoothed. Pushing viscosity to 1.0 flips the response: the initial pressure spike ejects a wave of particles, after which the remaining fluid trudges forward with no ripples. A mid-range viscosity (≈ 0.3) splits the difference, taming chaotic spray without stifling natural wave motion.

6.1.5 Surface Tension Coefficient

Coefficient Value	Observations
σ = 0.01	Subtle wiggling motion throughout.

σ = 0.2	Particles experience popping motion. Particles pull together and ripple.
σ = 0.6	Shape is a lot rounder than the previous two trials. Clumps resembling droplets start to form over time. Particles ripple and pop.
σ = 1	Like the σ = 0.6 trial, but with more extreme clumping that breaks the surface integrity.

Takeaways:

With almost no cohesion (σ ≈ 0.01) the fluid "wiggles" because forces are too weak to smooth the surface. Increasing to σ ≈ 0.2 gives the particles enough pull to snap together after pops, creating visible ripples but still allowing occasional detachments. A stronger setting (σ ≈ 0.6) rounds the mass into a tidy drop that begins to spawn secondary droplets—evidence that curvature forces dominate without yet breaching the surface. Pushing to σ ≈ 1.0 over-tightens that pull: clumps coalesce so aggressively that the surface fractures into beads and loses overall surface integrity. Thus, mid-range tension (≈ 0.4–0.6) best balances ripple damping with cohesive realism, while extremes either under or over-bind the fluid.

6.2 Visuals

The final renders exhibit particle motion in a couple environments, a box and on procedural terrain. The box shows the liquid's ability to hold shape and the terrain shows how the particles flow across an uneven surface while keeping surface integrity. All renders are done with the parameter settings of the "default" liquid.

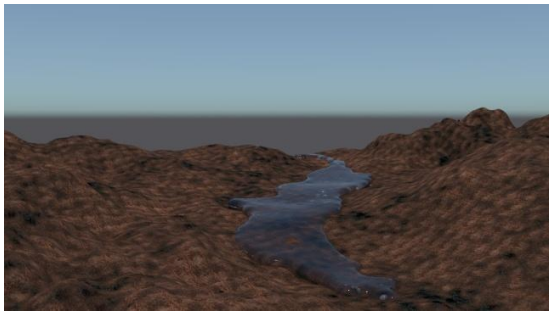

Figure 1: Small-scale SPH river test—rough-meshed terrain with a water like fluid flowing through the valley. This shot illustrates the solver's ability to conform to complex topography and preserve a thin, cohesive surface without volume loss.

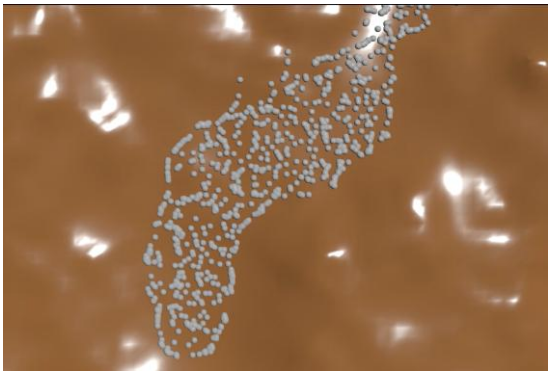

Figure 2: Overhead snapshot of particle positions meshed as VDB spheres while flowing across the procedurally generated terrain. The view captures an intermediate frame in the simulation, illustrating how the particle cloud elongates and conforms to slope changes along the riverbed.

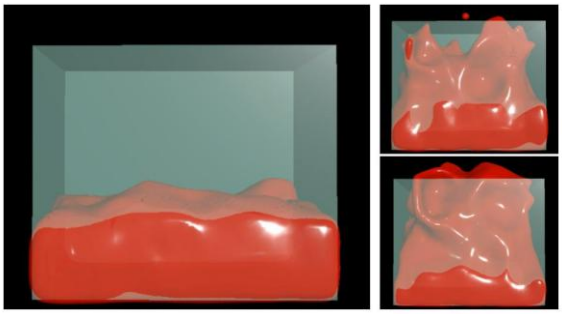

Figure 3: Frame sequence from the high-viscosity red liquid. Left—initial state after settling; top-right—mid-splash as the liquid starts to descend; bottom-right—early frame showing the initial dropping point of the liquid. The biggest issue seen here from the Fluid Surface DOP is that overall volume is somewhat inconsistent from the dropping to settling point.

6.3 Performance

To gauge how each stage of the pipeline affects runtime, I profiled the simulation twice—first in its default liquid form and then with a VDB-based surfacer enabled. For every node I chose to report three metrics: Cook (s) (CPU time spent cooking the node itself), Time (s) (total elapsed time, including child nodes and any waiting), and Solve (s) when applicable (the portion of Cook time that runs the actual solver, e.g., the GAS_GeometryVex pass in the POP Wrangle). All values were captured in the viewport by the Houdini Performance Monitor, with no previous simulation data cached.

6.3.1 Default Liquid (no VDB)

- Time step: *Δt = 0.02, 2 sub-steps*
- The highlighted value is the time it takes for GAS_GeometryVex to solve - the actual Vex powered SOP solver

Node / Stage	Cook (s)	Time (s)	Solve (s)
import_fluid	0.129	0.129	–
fluid_sim (POP Net)	2.302	14.258	–
POP Wrangle (SPH VEX)	0.026	9.555	9.529
POP solver	0.440	2.576	2.135
Viewport Render	–	5.145	–

Comparison: FLIP solver with same particle count

Node / Stage	Cook (s)	Time (s)
FLIP DOP	1.361	8.659
Viewport Render	–	5.979

6.3.2 Default Liquid (with VDB)

- VDB Smoothing ended up being the most computationally expensive node in the entire pipeline.

Node / Stage	Cook (s)	Time (s)	Solve (s)
import_fluid	59.120	59.120	–
VDB Smooth	48.869	48.869	–
fluid_sim (POP Net)	2.894	14.622	11.728
POP Wrangle (SPH VEX)	0.026	9.555	9.529
POP solver	0.440	2.576	2.135
Viewport Render	–	21.578	–

Takeaways: The SPH solver's core loop is only a around a second slower than Houdini's FLIP solver in-viewport, so raw math isn't the bottleneck. The spikes come from scene I/O and post-processing: live VDB smoothing adds nearly a minute for 96 frames, whereas running the same sim without VDB delivers comparable motion for a fraction of the time. Stability also degrades beyond ~5000. A likely culprit is the pcopen() neighbour search: as particle counts approach 5000, each POP Wrangle rebuilds its own point-cloud tree, multiplying memory use and cache misses. That overhead can dwarf the actual SPH math, stalling both the solve and any downstream VDB meshing. Larger scenes will need an offline meshing pass or a lighter surfacer. Meanwhile, FLIP is able to spawn upwards of 6000 particles with much less difficulty. In short, this SPH simulation offers fine art-direction but FLIP remains the faster option when top-end fidelity can be traded for speed.

Constraints

I kept the neighborhood search to one pcopen() call per frame by packing density, pressure, and viscosity work into a single POP Wrangle. Even so, the memory usage and computation time were still too high to use the Fluid Mesh DOP without a lot of difficulty. POP networks don't expose low-level memory access, and rolling a grid inside a single wrangle would have meant re-implementing neighbor queries from scratch. Given time and hardware limits, I kept the built-in point-cloud search and accepted its memory overhead as a constraint.

7. Conclusions and Future Work

This project presents a beginner-oriented SPH solver implemented entirely in Houdini that produces convincing water splashes with tunable viscosity, surface tension, and density control for scenes up to ≈ 3000 particles. While the solver yields interesting results, several features remain open:

- Foam mask: Completing the white-water attribute and accompanying shader will let the sim communicate aeration, spray, and shoreline churn that pure SPH particles can't convey.
- Vorticity: Initial attempts to curl-boost small eddies were unstable; revisiting this term (or adopting position-based vorticity methods) should restore swirling detail without explosions.
- Volumes/smoke: Expanding the usage of the simulation to gaseous fluids since I was focused on liquids.
- Real-time engine integration: Packaging particles or meshed surfaces into Unity would make the solver useful for interactive projects while shifting the heavy computation off Houdini.

Taking these next steps would evolve the solver into a versatile fluid-FX playground—capable of driving everything from foamy rivers to swirling smoke plumes in real-time.

References

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-Based Fluid Simulation for Interactive Applications. *Proc. Eurographics/SIGGRAPH Symposium on Computer Animation* (2003).

[Gre10] GREEN S. 2010. Screen Space Fluid Rendering for Games. *Game Developers Conference* (March 2010).

[KBST19] KOSCHIER D., BENDER J., SOLENTHALER B., TESCHNER M.: Smoothed Particle Hydrodynamics: Techniques for the Physics-Based Simulation of Fluids and Solids. EUROGRAPHICS

2019, Tutorial, W. Jakob and E. Puppo (Eds.), (2019).

[BM07] BRIDSON R., MÜLLER M.: Fluid Simulation. *Proc. SIGGRAPH 2007* (2007).

[Moc11] MOCZ P.: Smoothed Particle Hydrodynamics: Theory, Implementation, and Application to Toy Stars. Mon. Not. R. Astron. Soc. 000, 1–9 (2011).

[Lag23] LAGUE S.: Coding Adventure: Simulating Fluids with Smoothed Particle Hydrodynamics. YouTube video, (2023). Available at: https://www.youtube.com/watch?v=rSKMYc1CQHE.

[EPG*19] EMMANUEL K. S., MATHURAM C., PRIYADARSHI A. R., GEORGE R. A., ANITHA J.: A Beginner's Guide to Procedural Terrain Modelling Techniques. Proc. 2019 2nd International Conference on Signal Processing and Communication (ICSPC), IEEE, (2019). DOI: 10.1109/ICSPC46172.2019.8976682.

	1/27	2/3	2/10	2/17	2/24	3/3	3/10	3/17	3/24	3/31	4/7	4/14	4/21	4/28	4/30
Particle Initialization															
Basic Fluid Effects															
Additional Fluid Effects															
Procedural river generation															
Final presentation + render															
Writeup															

Figure 4: *Gantt chart.*