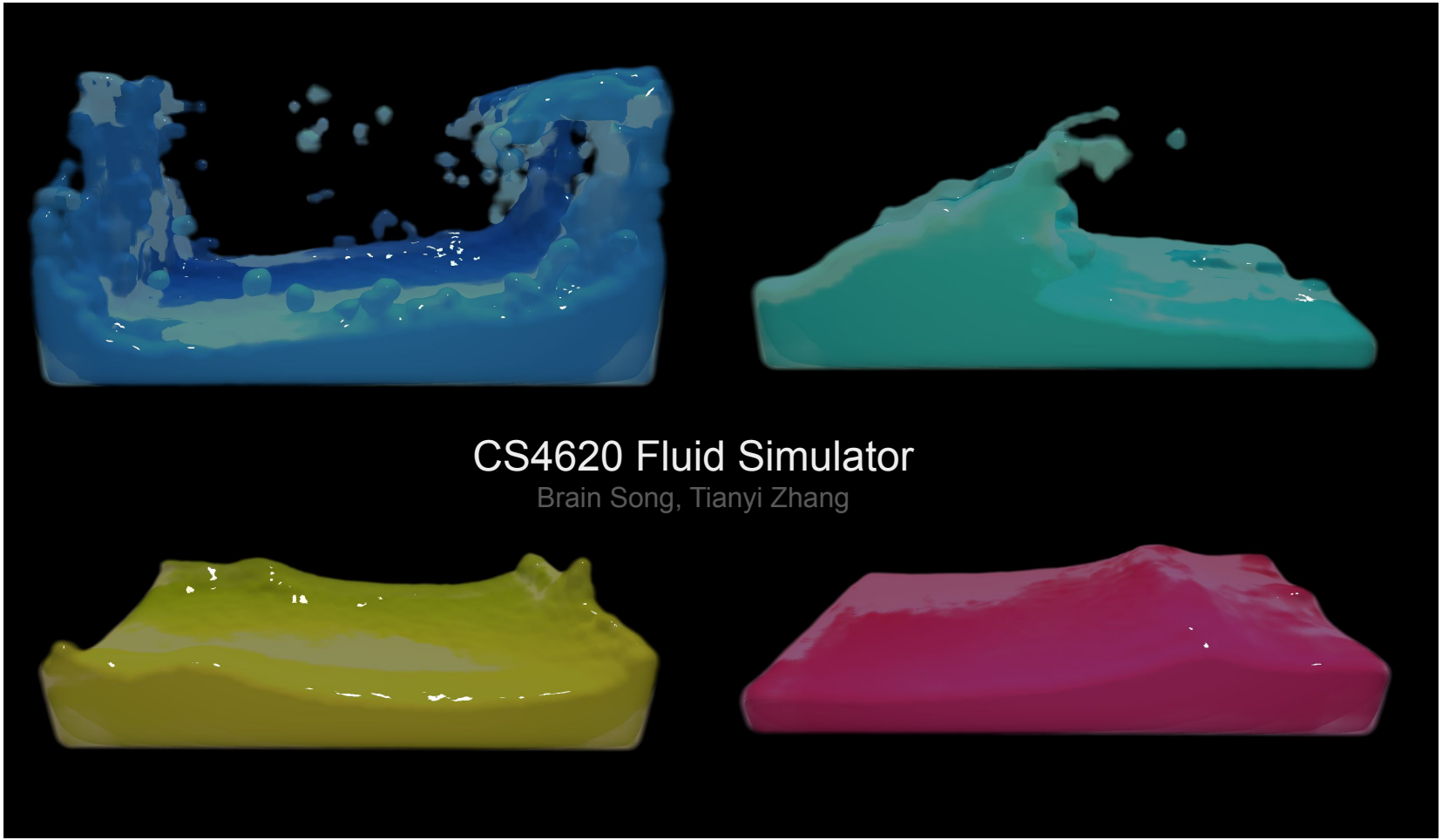

CS4620 Fluid Simulator
Brain Song, Tianyi Zhang

Feature List

Feature 1: Density field simulation
Feature 2: Pressure simulation
Feature 3: Viscosity
Feature 4: Prediction Step
Feature 5: SDF collision response
Feature 6: Particle Lifecycle Management
Feature 7: Packing Particle Data into the GPU
Feature 8: Manual Density Pass
Feature 9: Sub-stepping for Stable Simulation
Feature 10: 3D spatial hash
Feature 11: Bitonic sort
Feature 12: O(1) Index Lookup
Feature 13: 3D Density Field Generation
Feature 14: Core Rendering Pipeline
Feature 15: Volumetric Shading with Beer–Lambert Law
Feature 16: Adaptive Ray Marching & Surface Detection
Feature 17: Fresnel Rim Light & Depth-Based Color
Feature 18: Gradient-Based Normal Reconstruction
Feature 19: SDF-driven Shoreline
Feature 20: Fake Refraction & Environment Blending
Feature 21: Edge Hardening & Alpha Control
Feature 22: Water Interaction

Feature 1: Density field simulation

Use the **kernel function** to estimate density for each particle based on nearby particles.

For each particle, we accumulate contributions from neighbors. (Graph is 2D version)

We compute both

- **standard density (`density`)** – used for overall pressure and fluid behavior, and
- **near-field density (`nearDensity`)** – emphasizes very close neighbors, helping to model stronger local forces (e.g., surface tension or clumping effects).

```
private smoothingKernel(radius: number, dst: number): number {  Show usages  zs292
    if (dst >= radius || radius <= 0) return 0;
    const volume : number  = Math.PI * Math.pow(radius,  y: 4) / 6.0;
    return (radius - dst) * (radius - dst) / volume;
}

private smoothingKernelDerivative(radius: number, dst: number): number {  Show usages  zs292
    if (dst >= radius || radius <= 0) return 0;
    const scale : number  = 12 / (Math.PI * Math.pow(radius,  y: 4));
    return (dst - radius) *scale;
}

private nearDensityKernel(radius: number, dst: number): number {  Show usages  Tianyizhang *
    if (radius <= 0 || dst >= radius) return 0;

    const q : number  = 1 - dst / radius; // [0,1]
    return q * q * q;
}

private nearDensityDerivative(radius: number, dst: number): number {  Show usages  Tianyizhang
    if (radius <= 0 || dst >= radius) return 0;

    const q : number  = 1 - dst / radius;
    // nearDensity = q^3, q = 1 - r/h
    // d/dr (q^3) = 3 q^2 * d q/dr = 3 q^2 * (-1/h) = -3 q^2 / h
    return -3 * q * q / radius;
}
```

Feature 2: Pressure simulation

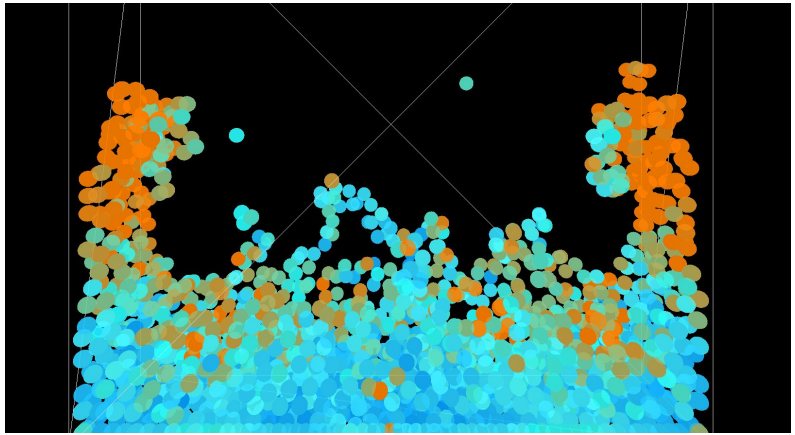
Compute **pressure** and **near-field pressure** for each particle using an **equation of state** (pressure increases with density).

The **near-field pressure term** is added to **discourage particles from clumping too tightly** and to keep the fluid more evenly distributed.

Use the **gradient of the Spiky kernel** to compute the **pressure gradient force**, which pushes particles from high-pressure regions toward low-pressure regions.

Spiky kernel is the standard kernel for SPH simulation, because it has **non-zero gradient** at r=0. This prevents the particles from overlapping each other.

Color different pressure



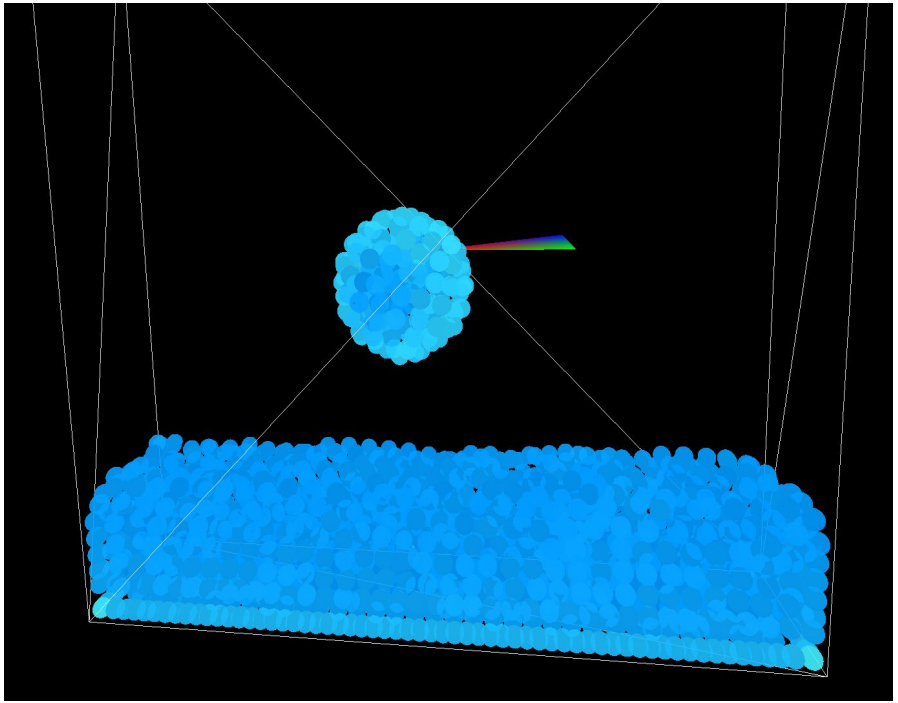
CPU version

Feature 3: Viscosity

Compute the **velocity differences** between a particle and its neighbors to estimate viscous effects.

Apply a **viscous damping force** (based on neighbor velocity differences) to smooth out velocity variations and simulate the natural stickiness of real fluids.

With viscosity included, the fluid becomes **less prone to breaking apart** — so when we apply external attraction forces, the water **sticks together rather than scattering**.

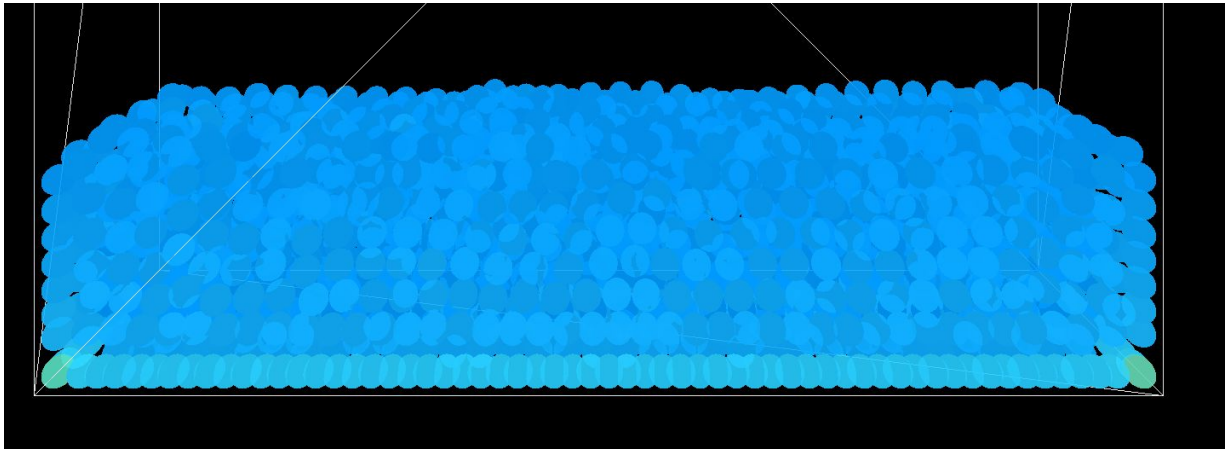


Feature 4: Prediction Step

First compute a **predicted position** `p_pred = pos + vel * dt` based on the current velocity.

Use this **predicted position** when computing **neighbor densities and forces**, so interactions are based on where particles *will* be.

This predict–correct strategy **improves numerical stability** and reduces artifacts such as jittering and particles tunneling into each other.



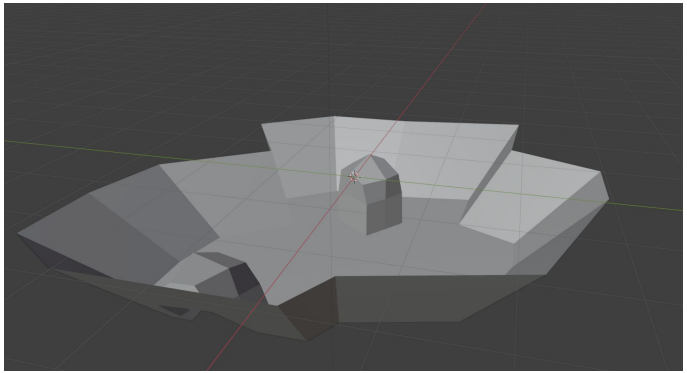
Feature 5: SDF collision response

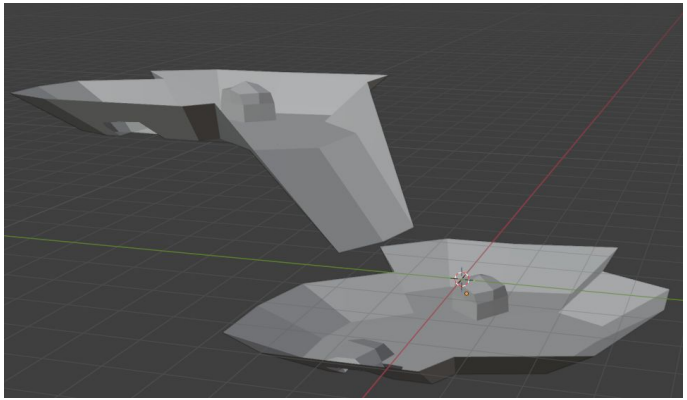
Preprocessing

- Take an arbitrary **triangle mesh (glb model)** and rasterize it into a **3D SDF voxel grid**.

Collision Response (Runtime)

- During each physics step, **sample the SDF and its gradient** at the particle position with a trilinear interpolation.
- Use the **SDF gradient as a surface normal** to compute
    - a **reflection** response (`Reflect`) and
    - a **push-out** correction (`Push`) to move particles back outside the surface.
- **Performance is independent of mesh complexity.** For each particle, the complexity is O(1). This allows us to support **arbitrary, highly complex container shapes** with smooth collision handling.

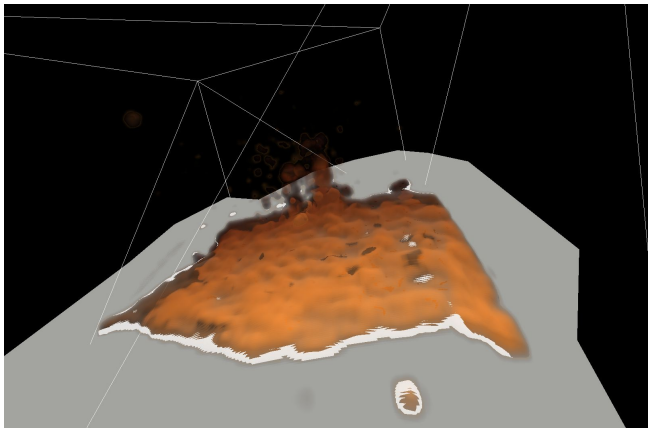


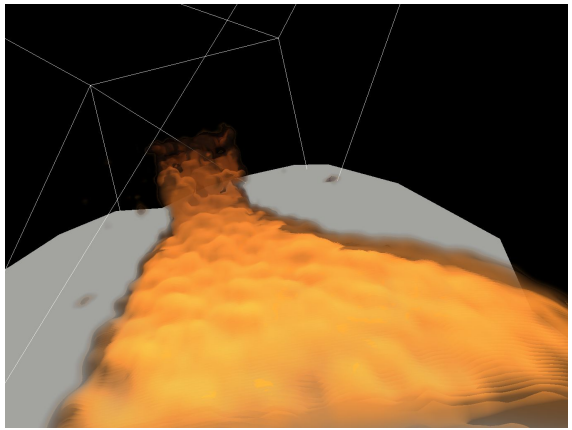


Feature 6: Particle Lifecycle Management

Out-of-Bounds Recycling

- Automatically identify particles that 1. escaped the simulation domain 2. got stuck in the model (sdf<0)
- Recycle or respawn them to maintain a stable particle count.





Feature 7: Packing Particle Data into the GPU

WebGL does not support compute shader (GPU use for general purpose computation). Only the rendering pipeline runs on GPU.

A way to get around this is to store data as **textures**, and computation logic as **fragment shaders**.

Luckily, three.js's `GPUComputationRenderer` helps with this trick.

Represent the physical state of a particle as a **texel**: Instead of [r,g,b,a], we store:

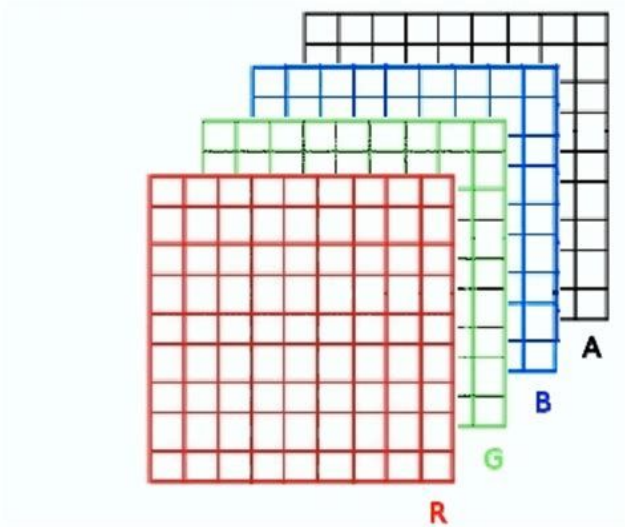
Position texture - [x, y, z, 0]

Velocity texture - [vx, vy, vz, 0]

Density texture - [density, nearDensity, 0, 1]

…

All simulation updates and storage can be done completely in GPU.



Pictorial representation of a RGBA texture. | Download Scientific Diagram

Feature 8: Manual Density Pass

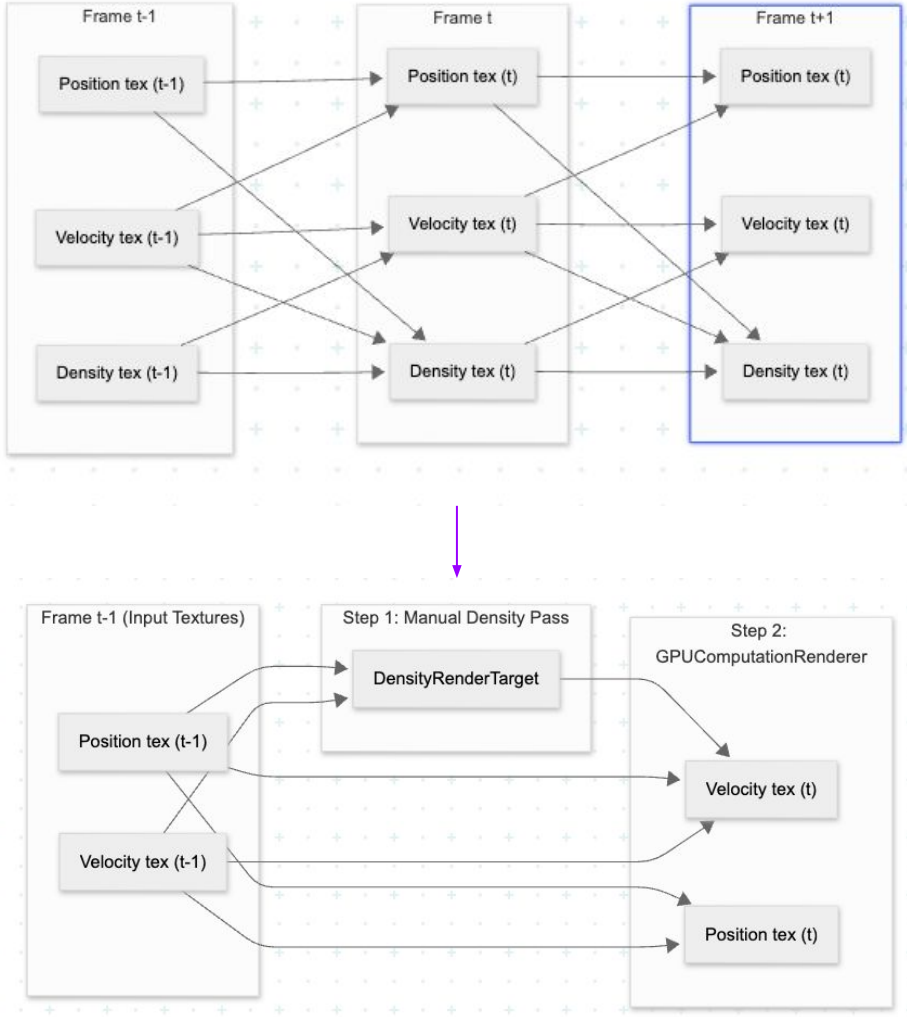
The simulation completely based on GPUComputationRenderer is very **unstable**.

Position and Velocity at frame t+1 is computed based on the textures from frame t.

Whereas Density at frame t comes from frame t-1.

This **1-frame lag** is the source of instability.

Solution: **Manually** shade (compute) DensityRenderTarget to compute the density right away.



Feature 9: Sub-stepping for Stable Simulation

Split each frame's physics update into **3 smaller sub-steps** instead of one large step.

Limit each sub-step's timestep `dt` to **no more than 1/360 s**, preventing large jumps in particle motion.

This significantly **improves stability**, especially for **high-pressure or fast-moving fluids**, reducing artifacts such as explosions, jitter, or particles tunneling through boundaries.

This is computationally achievable thanks to the GPU computation pipeline.

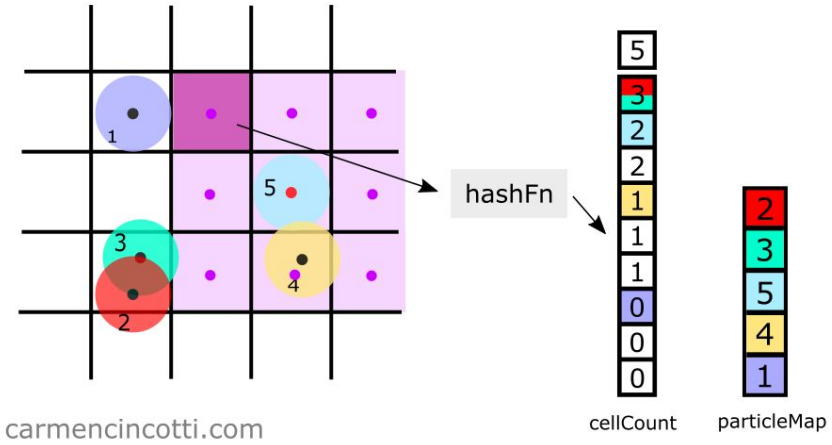
Feature 10: 3D spatial hash

Build a **3D uniform grid** whose cell size is based on the `smoothingRadius`.

Convert each particle position to an **integer cell coordinate**, then map it to a **unique CellID** via a hash function.

Store particles in a **spatial hash table**, so neighbor queries only need to look up a few hash buckets instead of scanning all particles.

Because we use a **hash table instead of a fixed 3D array**, the grid can **naturally extend to large or unbounded spaces** – we only allocate cells where particles actually exist.



Feature 11: Bitonic sort

Implement a **fully parallel bitonic sorting algorithm** on the GPU.

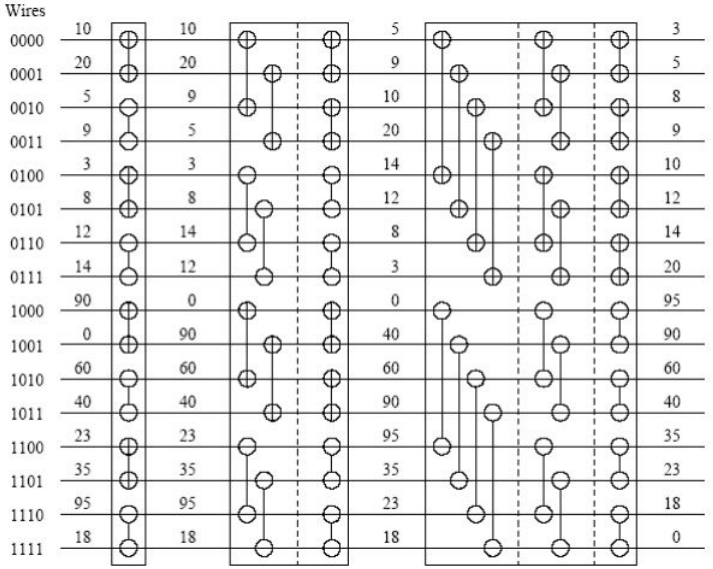
Sort all particles by their `CellID` directly inside the floating-point textures.

After sorting, particles belonging to the same or neighboring cells become **contiguous in memory**, which greatly improves **cache locality and coherent memory access** for neighbor queries.

Why Bitonic Sort?

- **Architecture Constraints:** WebGL fragment shaders lack the Random Scatter (random write) capabilities in modern Compute Shaders (e.g., Unity/CUDA). Bitonic sort gets around this by relying solely on fixed-pattern reads, at the cost of using more passes to store intermediate states.

- **Deterministic Sorting Network:** Many more efficient sortings require **atomic operations** (unsupported by WebGL). Bitonic Sort has a fixed comparison pattern that fits perfectly into the WebGL Ping-Pong rendering pipeline.



https://people.cs.rutgers.edu/~venugopa/parallel_summer2012/bitonic_overview.html

Feature 12: O(1) Index Lookup

Use a **CellRange Pass** to precompute, for every grid cell, the `[Start, End)` index range of particles in the sorted particle array.

This pass uses a GPU-based binary search to locate the first and last particle belonging to each `CellID`.

Once the ranges are computed, neighbor queries become **constant-time lookups**:

- Given a `CellID`, directly access its particle span in **O(1)**.

This eliminates per-frame scanning or searching, enabling fast and predictable neighbor retrieval.

```
void main() {
    ivec2 fragCoord = ivec2(gl_FragCoord.xy);
    int cellId = fragCoord.x + fragCoord.y * int(cellResolution.x);

    if (cellId >= int(cellCount)) {
        gl_FragColor = vec4(-1.0, -1.0, 0.0, 0.0);
        return;
    }

    int start = lowerBoundCell(cellId);
    if (start < 0) {
        gl_FragColor = vec4(-1.0, -1.0, 0.0, 0.0);
        return;
    }
    int endExclusive = upperBoundCell(cellId);
    gl_FragColor = vec4(float(start), float(endExclusive), 0.0, 0.0);
}
`;
```

Feature 13: 3D Density Field Generation

Reuse the **same grid / neighbor search logic** from the physics simulation to build a 3D density field.

For each particle, **“splat” its density contribution** into a separate **3D voxel grid**. This is different from the pass of density centered at each particle.

This produces a **continuous volumetric density texture** (3D texture turned into 2D atlas) that can be used for **ray-marching, volume rendering, and surface extraction**.

In the **fragment shader**, we perform **ray marching** through the density field to render the water volume.

Feature 14: Core Rendering Pipeline

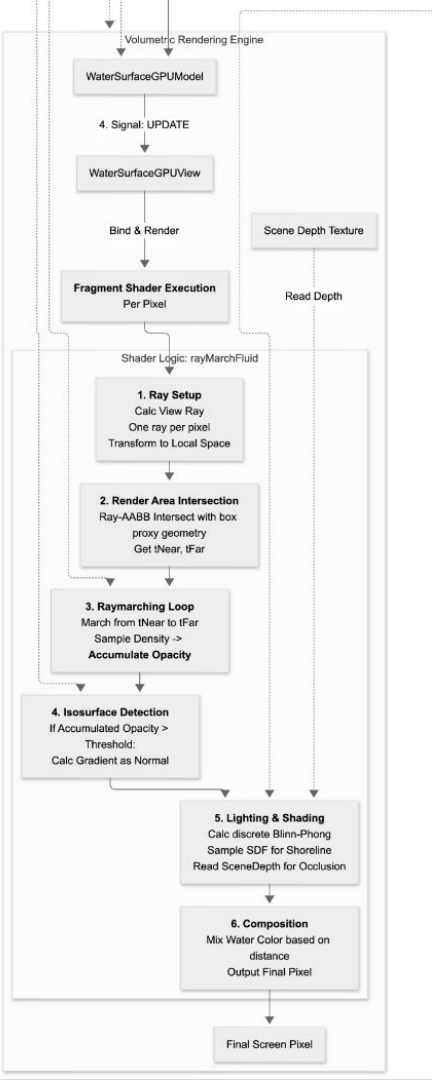
We draw a single **fullscreen quad**.
All water rendering is done **entirely in the fragment shader**, using raymarching over our 3D density field.

Depth Awareness & Occlusion

- Read the **scene depth texture** (`sceneDepthTexture`) and **reconstruct the world-space position** of each screen pixel.
- During raymarching, compare the **water depth** with the **scene depth** to get correct occlusion: Water appears in front of objects when closer to the camera, and objects correctly occlude water when they are in front of the fluid.

Ray–Box Intersection Optimization

- Before doing expensive raymarching, compute the intersection of the view ray with the water **axis-aligned bounding box (AABB)** to get `entry` and `exit` points.
- If the ray does **not** hit the water volume, we immediately `discard` the fragment.
- This drastically reduces the number of rays we march and **improves performance**.

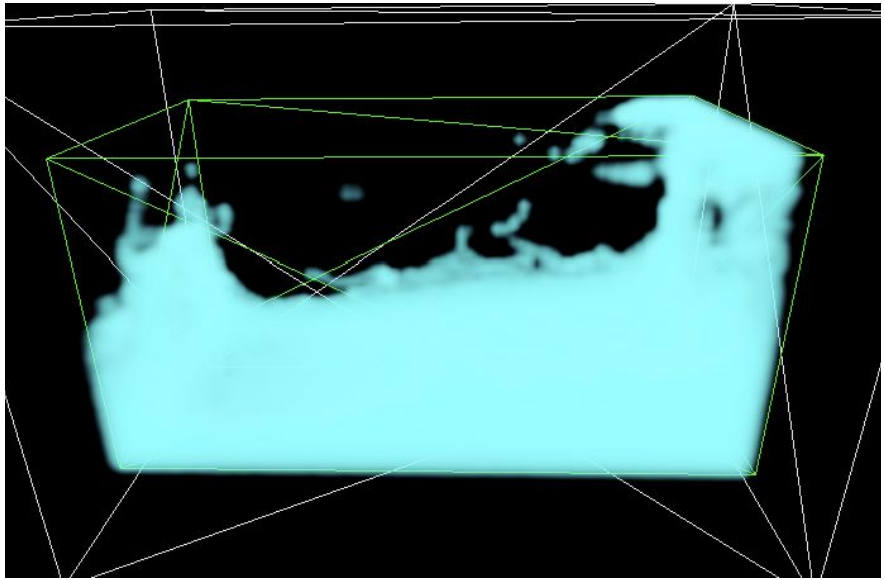


Feature 15: Volumetric Shading with Beer–Lambert Law

Beer–Lambert Absorption Model

- March along the view ray in small steps and **sample the 3D density texture (atlas)** generated by the physics layer (Feature 13).
- Use the **Beer–Lambert Law** to model exponential light attenuation inside the medium:
  ```
  Transmittance = exp(-density * stepSize * absorptionCoefficients)
  ```

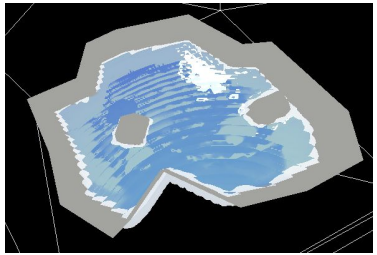
- Higher density leads to **stronger RGB absorption**, producing **deeper, richer water colors** in thick regions of the fluid.



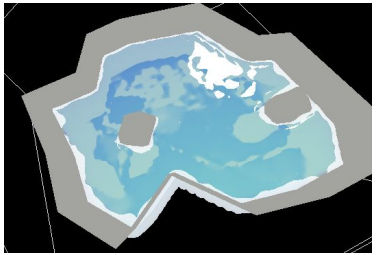
Feature 16: Adaptive Ray Marching & Surface Detection

Adaptive Stepping

- `maxViewSteps` (default **8192**) caps the **maximum number of ray-march steps** per pixel.
- The **initial** step size `stepSize` is **automatically scaled** based on the water volume's bounding box size, so at different scene scales we don't explode the total step count or tank performance.
- Step size after the initial setting still adjustable with a **multiplier**



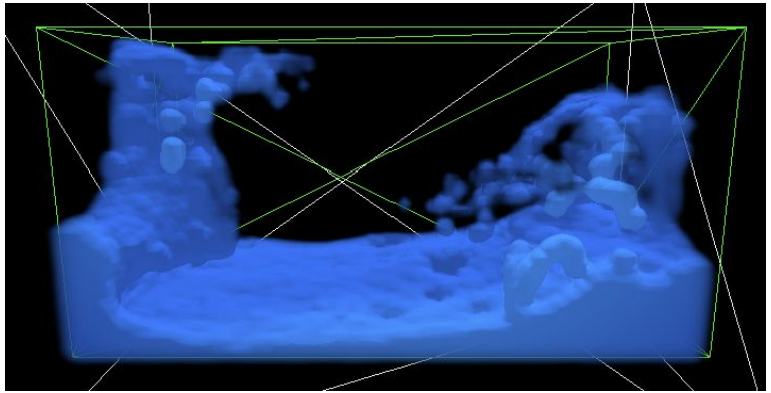
Large step size



Small step size

Implicit Surface Detection

- We do **not** rely on an explicit mesh surface.
- As we march along the view ray, we **accumulate opacity**; once it exceeds a threshold (`surfaceOpacityThreshold`), we treat this point as **hitting the water surface**.
- The corresponding `SurfacePos` is stored and used for **surface shading**, such as reflections, refractions, and highlights.



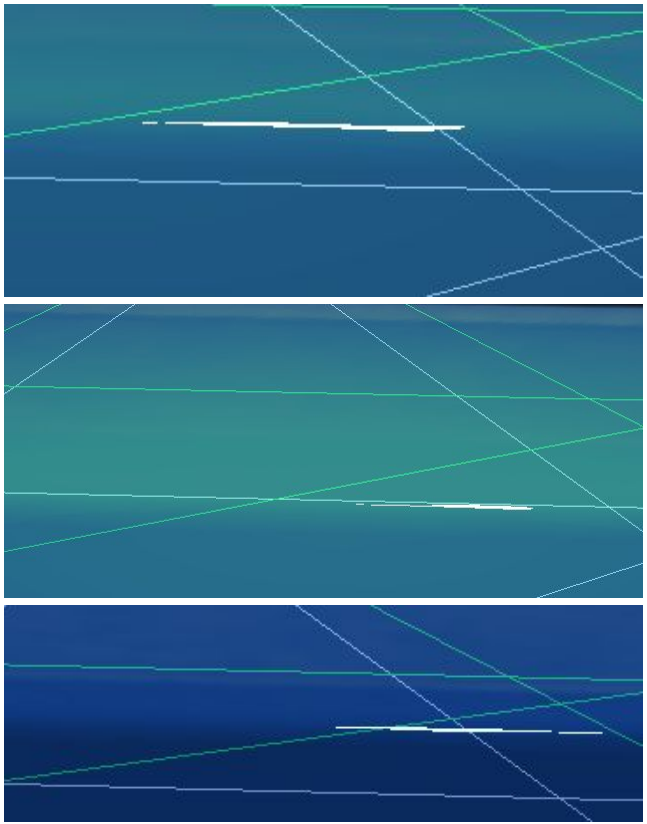
Feature 17: Fresnel Rim Light & Depth-Based Color

Fresnel Rim Light

- Think of looking at a glass or water surface:
  - **Straight-on** you mostly see through it → **less reflection** → not very bright.
  - **Grazing angle** (looking along the surface) reflection becomes strong → a **bright rim** shows up.
- We model this with: `pow(1.0 - dot(N, V), exponent)`
  - Higher `exponent` → **thinner, sharper** rim
    Lower `exponent` → **wider, softer** rim

Depth-Based Color Gradient

- The distance traveled along the camera ray from entering the water body's bounding box to the first intersection with the water surface. The more distance, the darker the color. (Move away camera, water gets darker)
- We use **`TravelDistance`** as a thickness estimate and blend colors:
  - shallow → `ShallowColor` (light/cyan)
  - deep → `DeepColor` (dark/blue)
  - `mix(ShallowColor, DeepColor, depthT)`



Feature 18: Gradient-Based Normal Reconstruction

Reconstructing Normals from the Density Field

- Since we have **no explicit mesh normals**, we estimate them directly from the **3D density field**.
- Use **central differences** along the three axes (`sampleDensity(pos ± epsilon)`) to compute the local **density gradient**, then normalize it to get a surface normal.

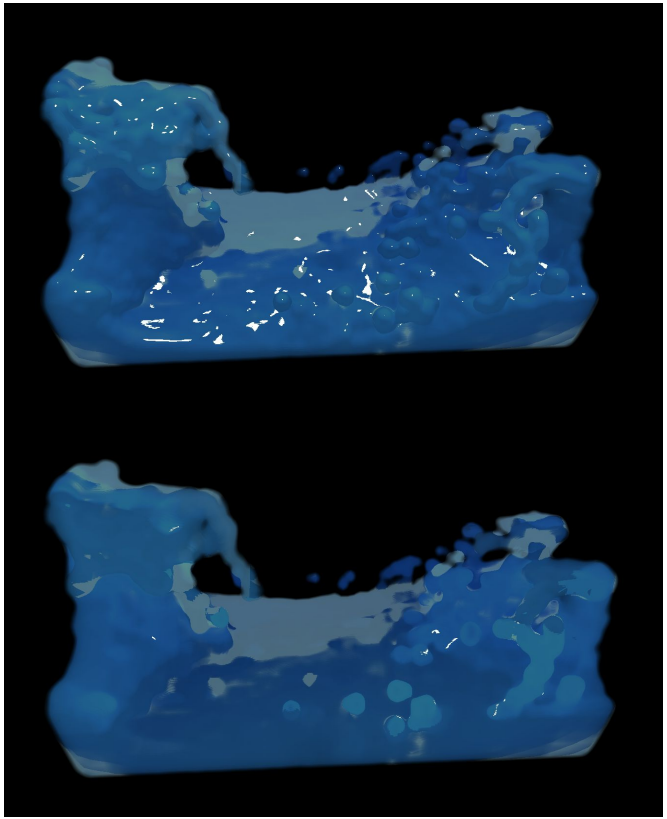
Dual-Normal Setup for Better Shading

- We compute **two sets of normals** at each surface point, differing only in the **epsilon scale factor** used for sampling:

  `Normal` (controlled by `normalEpsilonFactor`): Used for reflection with **Fresnel** effects and **refraction**.

  `SpecularNormal` (controlled by `specularNormalEpsilonFactor`):Used for **specular highlights**.

- This separation allows an exaggeration for fresnel to create a watercolor-like coloring (by actually making the normal inaccurate), while preserving the correctness of specular highlights.



Feature 19: SDF-driven Shoreline

Uses the simulation's collision SDF texture

Sample the SDF at the estimated water surface position to get the distance to the nearest solid

Smaller distance ⇒ closer to the shore, used to place foam along boundaries

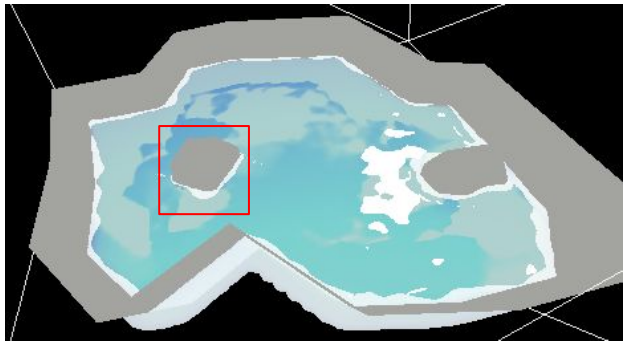
Procedural noise for natural variation

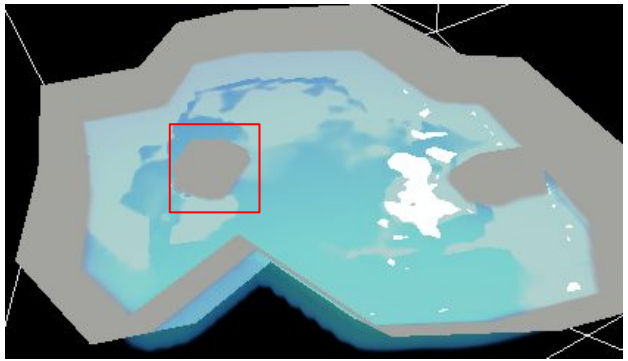
Spatial warping (low-frequency noise): perturb the sampling position before SDF lookup to avoid rigid, perfectly smooth shorelines

Dynamic erosion (time-varying high-frequency noise, uTime): modulate the distance field over time to mimic waves breaking and foam flicker

Stylized hard edge

Apply a step-based threshold to binarize the foam region, producing a crisp white foam outline





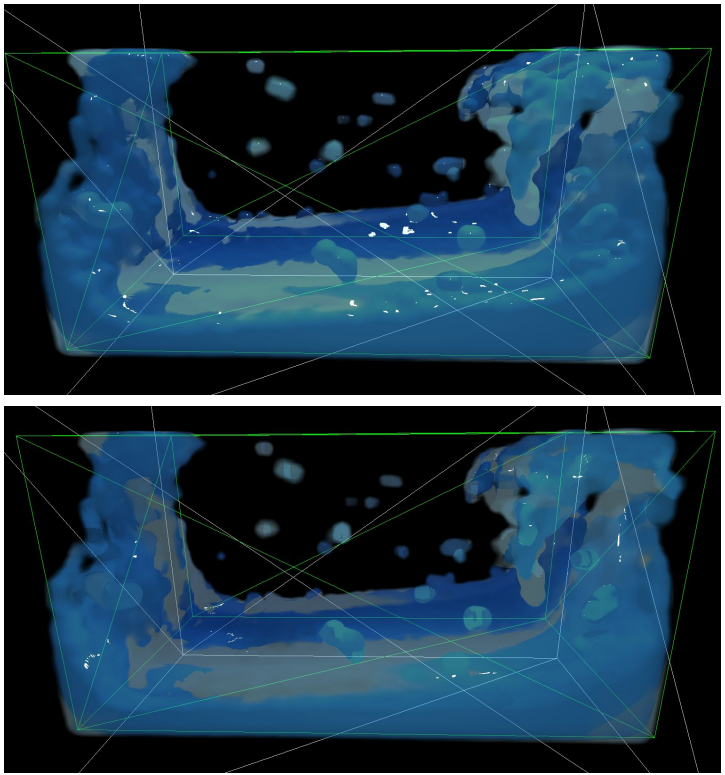
Feature 20: Fake Refraction & Environment Blending

Geometric Refraction Direction

- Use the physical refraction formula
  `refractDir = refract(-rayDir, normal, refractionEta)`
- This gives a **physically-based refracted direction**, but we do **not** trace it through the real scene — we use it as a direction cue.
- Refracted light is **darkened and tinted** based on the traveled thickness, simulating **light absorption inside the fluid**.

Simplified Environment Sampling (Fake Refraction)

- Instead of sampling the real scene color, we use a **placeholder / simplified environment color** via `sampleEnvironmentSimple()`.
- This environment term is then blended with the water's own color using a **`refractionBlend` coefficient**.
- As a result, the effect behaves like **"fake refraction / environment refraction"**:
  - direction from a physical **refract()**,
  - but background comes from a **simplified environment model**, not the actual screen or scene texture.



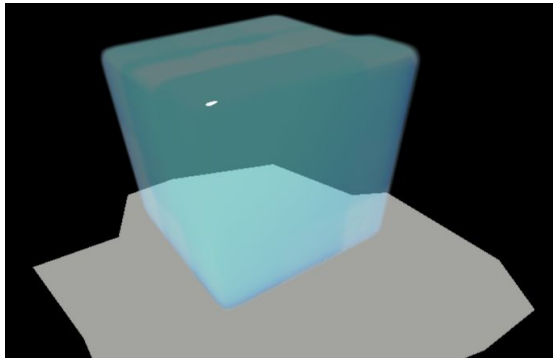
Feature 21: Edge Hardening & Alpha Control

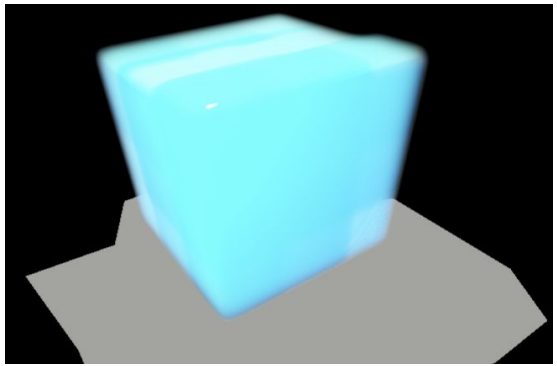
Edge Hardening

- Apply a `smoothstep` or linear remap like `(opacity - edgeCut) / (1.0 - edgeCut)` to the final accumulated alpha.
- This **cuts off very thin, foggy layers** and makes the **water silhouette sharper and cleaner** at the edges.

Max Opacity Cap

- Clamp the water's **maximum opacity** to a fixed upper bound.
- Even in the **deepest regions**, this keeps a bit of **see-through translucency**, so the water never looks like a solid wall.





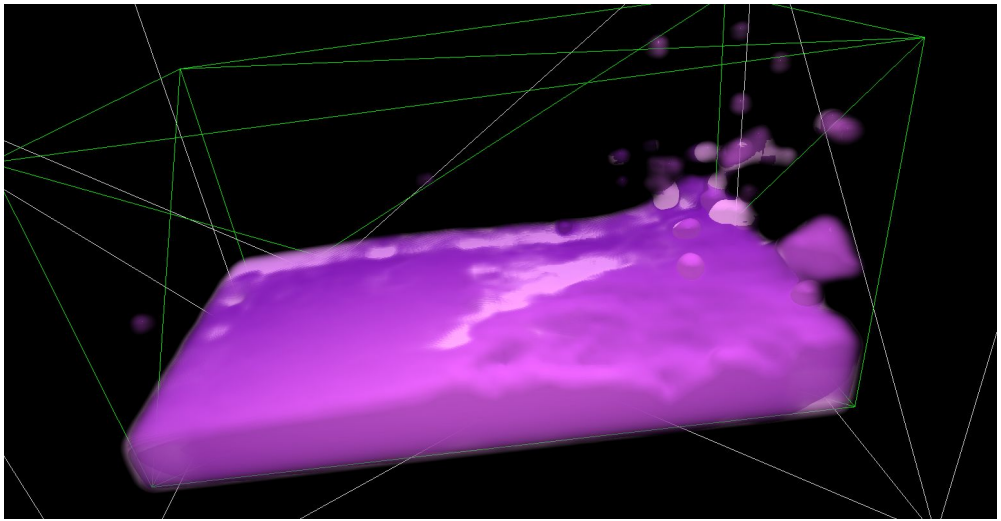
Feature 22: Water Interaction

Press **V** to trigger an interactive **raycast-based force** from the camera/cursor direction.

We compute the ray's **intersection with the container** (via collision geometry / SDF), and apply the force only within that local region.

As a result, **only water near the container surface** (near the hit point) responds, giving intuitive "push/pull" interaction without affecting the entire fluid.

Apply force on the right



References for Fluid Simulation

Bitonic sort: https://www.youtube.com/watch?v=uEfieI0MumY&t=464s

Fluid physics: https://www.youtube.com/watch?v=rSKMYc1CQHE

Shoreline style reference: https://www.youtube.com/watch?v=uOXagWeOAv4&t=3s

Shader: https://www.youtube.com/watch?v=3mfvZ-mdtZQ&t=39s

Threejs GPGPU: https://threejs.org/examples/?q=gpgpu#webgl_gpgpu_birds

Rendering Fluid: https://www.youtube.com/watch?v=kOkfC5fLfgE

RayMarching: https://www.youtube.com/watch?v=BNZtUB7yhX4

SDF: https://www.youtube.com/watch?v=LyQWZRfWotQ