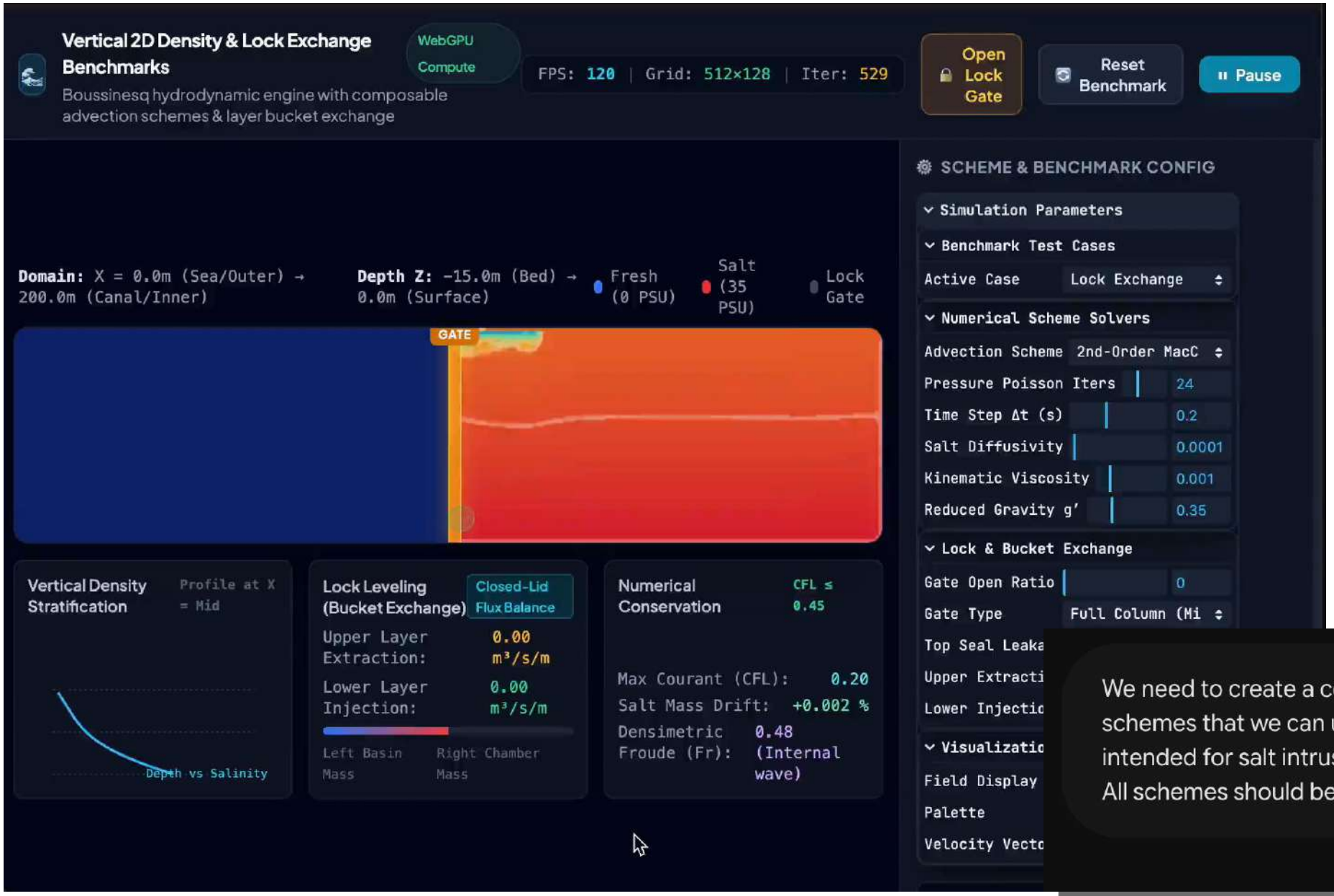
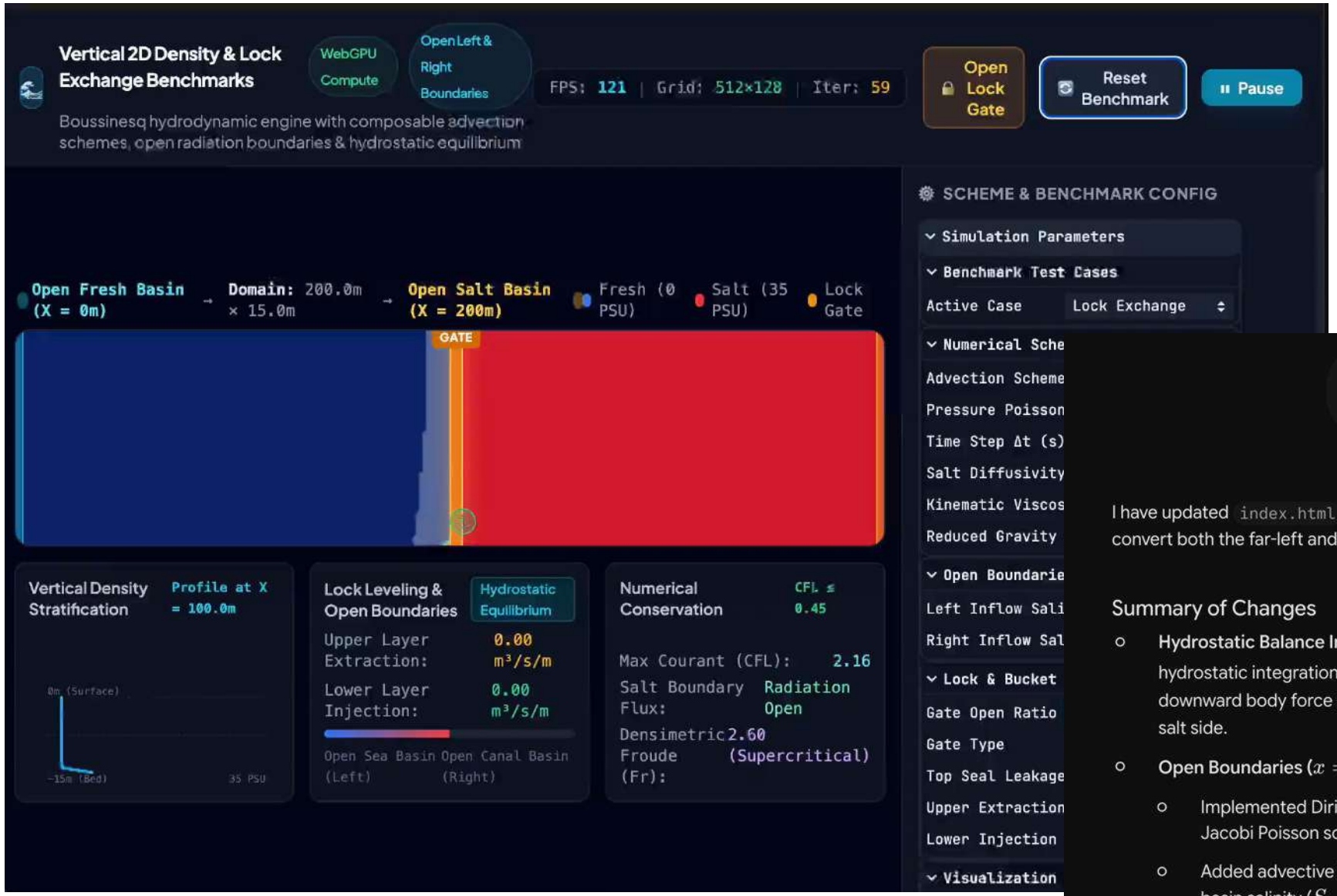
Experiment 2: Numerical challenges in 2Dv Lock

- Goals
  - Simulate free-surface 2Dv lock exchange (as a demo in the browser)
  - Browser based
  - Interactive model
  - Numerical challenges:
    - Pressure gradient
    - Gate opening
    - Open boundaries
    - Open lid





We need to create a collection of test cases for numerical schemes that we can use in a 2d vertical density model. It is intended for salt intrusion in a lock simulation.
All schemes should be implemented in webgpu. They sho



Water is oscilating at t=0 at the salt side. Also both the far left and far right should be open boundaries

I have updated `index.html` to resolve the initial oscillation at $t = 0$ on the salt side and convert both the far-left and far-right domain limits into open boundaries.

Summary of Changes

- **Hydrostatic Balance Initialization**: Initialized the 2D pressure field with column-wise hydrostatic integration $P_{hydro}(x, z) = \int_z^H g' \frac{S(x,z')}{35} dz'$. This balances the Boussinesq downward body force at $t = 0$, eliminating initial vertical sloshing and wall shocks on the salt side.
- **Open Boundaries ($x = 0$ and $x = width - 1$)**:
  - Implemented Dirichlet hydrostatic pressure radiation at the left and right ends in the Jacobi Poisson solver.
  - Added advective inflow/outflow conditions: incoming fluids receive the ambient basin salinity ($S_{ambient}$), while outgoing fluids radiate outward with zero spatial gradient ($\partial S / \partial x = 0$).

WebGPU solver

Javascript: main loop

```
stepSimulation() {
  this.updateUniforms();
  const commandEncoder = this.device.createCommandEncoder();
  const workgroupsX = Math.ceil(this.width / 16);
  const workgroupsY = Math.ceil(this.height / 16);

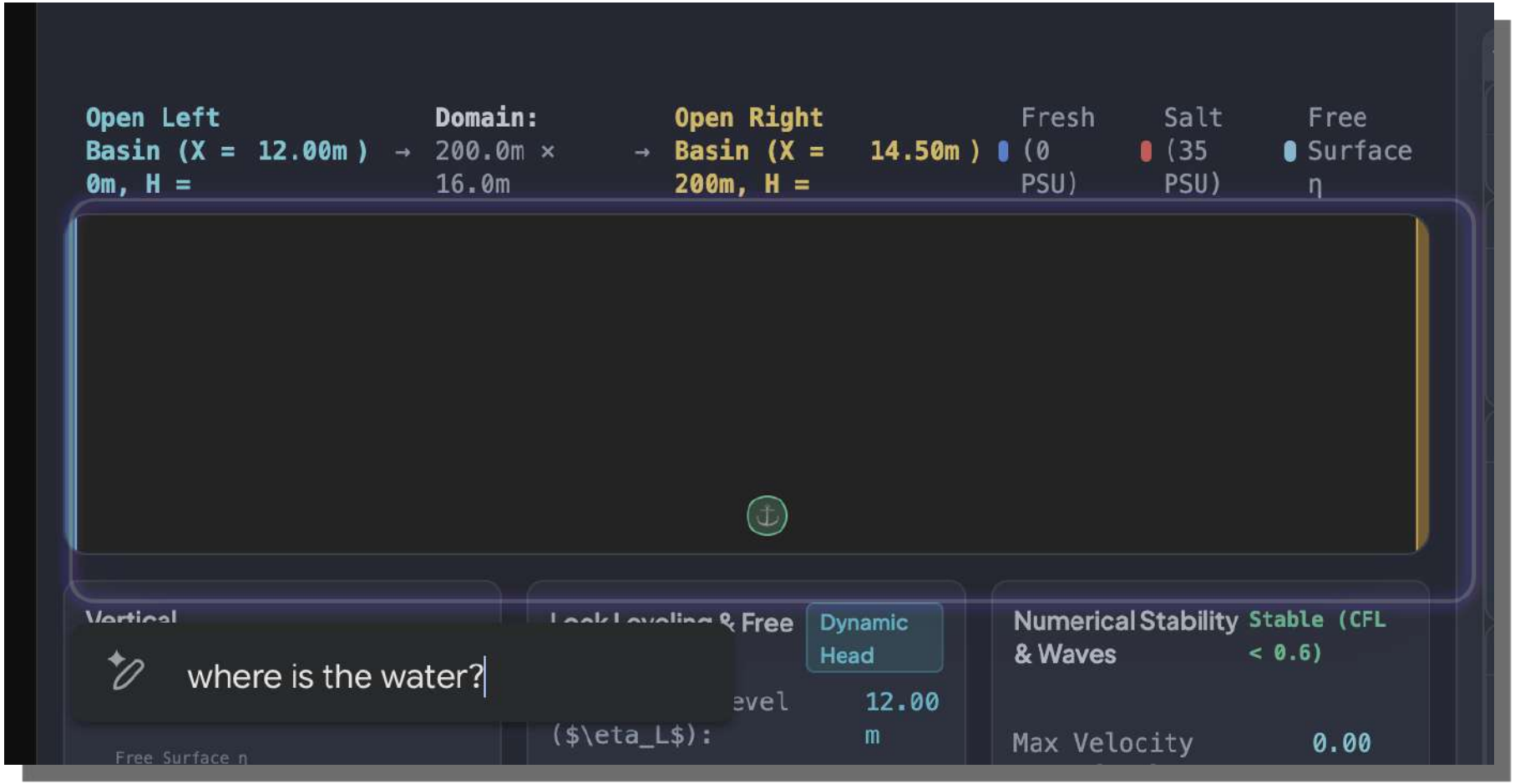
  // Advection Compute Pass
  {
    const bindGroup = this.device.createBindGroup({
      layout: this.advectPipeline.getBindGroupLayout(0),
      entries: [
        { binding: 0, resource: { buffer: this.uniformBuffer } },
        { binding: 1, resource: this.velTextureA.createView() },
        { binding: 2, resource: this.scalarTextureA.createView() },
        { binding: 3, resource: this.velTextureB.createView() },
        { binding: 4, resource: this.scalarTextureB.createView() }
      ]
    });

    const pass = commandEncoder.beginComputePass();
    pass.setPipeline(this.advectPipeline);
    pass.setBindGroup(0, bindGroup);
    pass.dispatchWorkgroups(workgroupsX, workgroupsY);
    pass.end();
  }
```

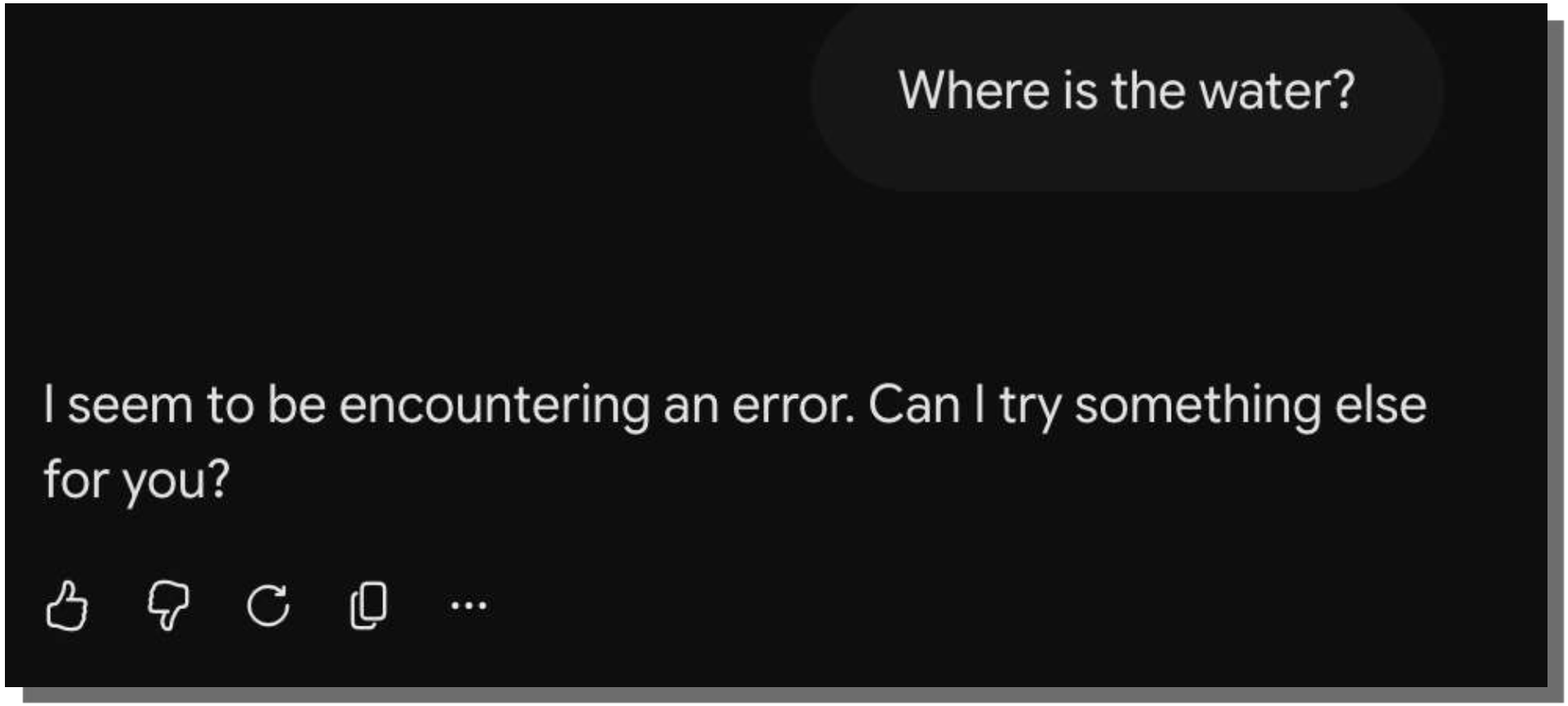
WGSL: compute shader for boundary

```
// Open Left Boundary (X = 0)
if (coords.x == 0) {
  let u_edge = textureLoad(velTextureIn, coords, 0).x;
  var s_edge: f32 = 0.0;
  if (u_edge > 0.001) {
    s_edge = params.ambient_s_left;
  } else {
    s_edge = textureLoad(scalarTextureIn, coords + vec2<i32>(1, 0), 0).r;
  }
  let v_inner = textureLoad(velTextureIn, coords + vec2<i32>(1, 0), 0).xy;
  textureStore(scalarTextureOut, coords, vec4<f32>(s_edge, 0.0, 0.0, 1.0));
  textureStore(velTextureOut, coords, vec4<f32>(v_inner.x, 0.0, 0.0, 1.0));
  return;
}
```

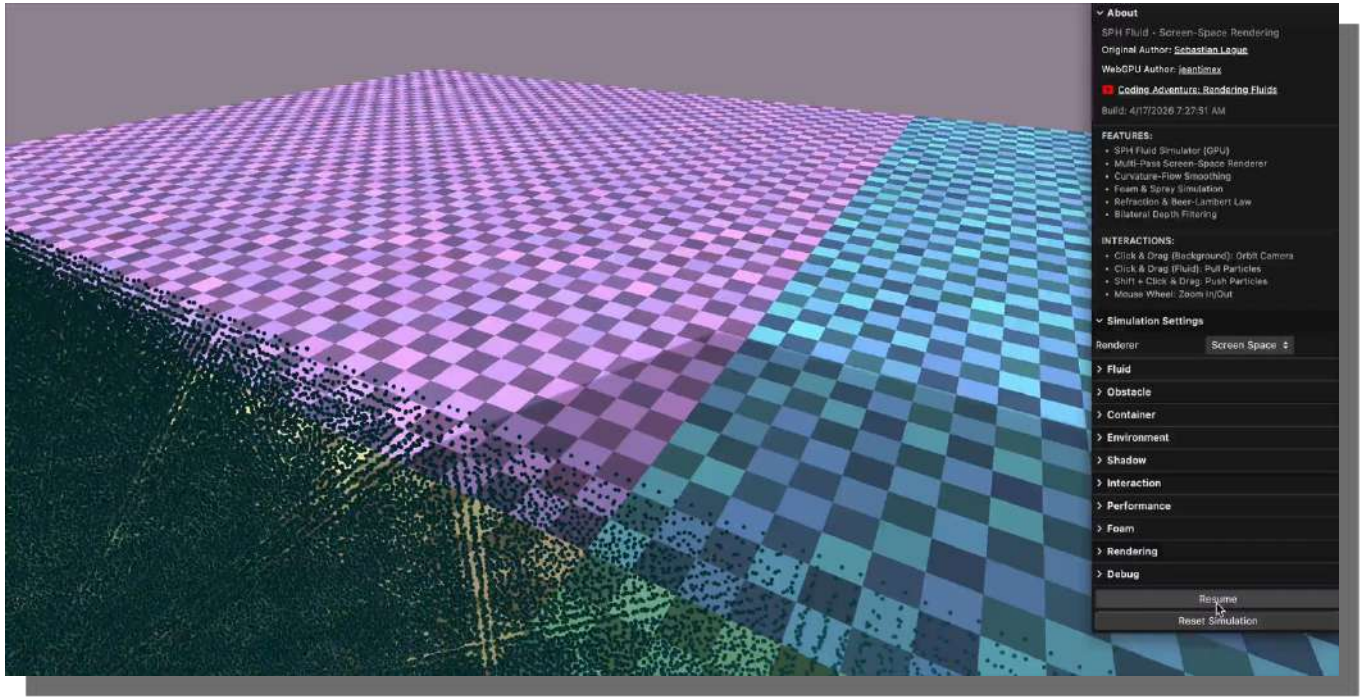
Next iteration: no more water

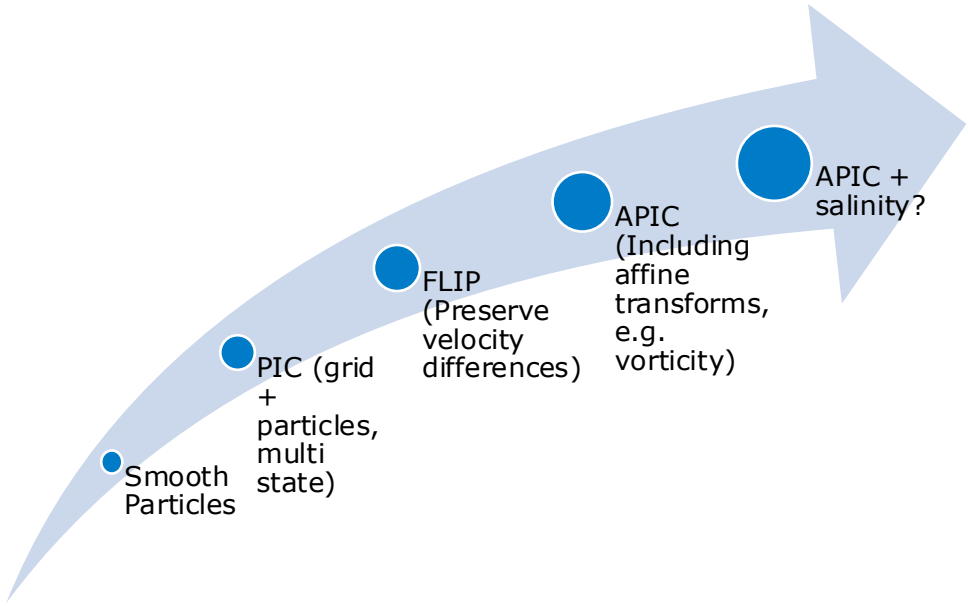


Next iteration: “doom loop”

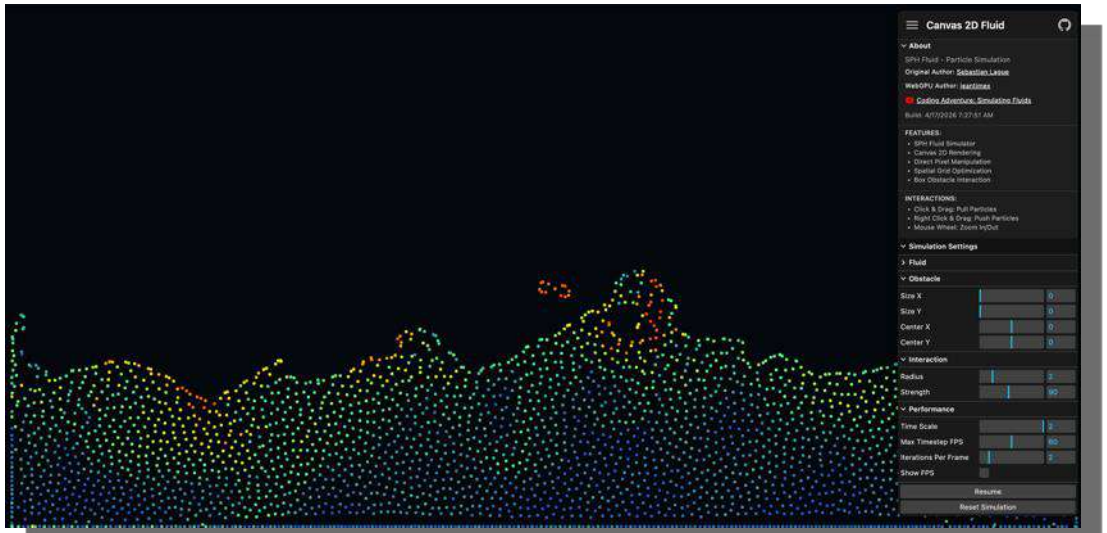


Different solver approach: APIC solver, 3D



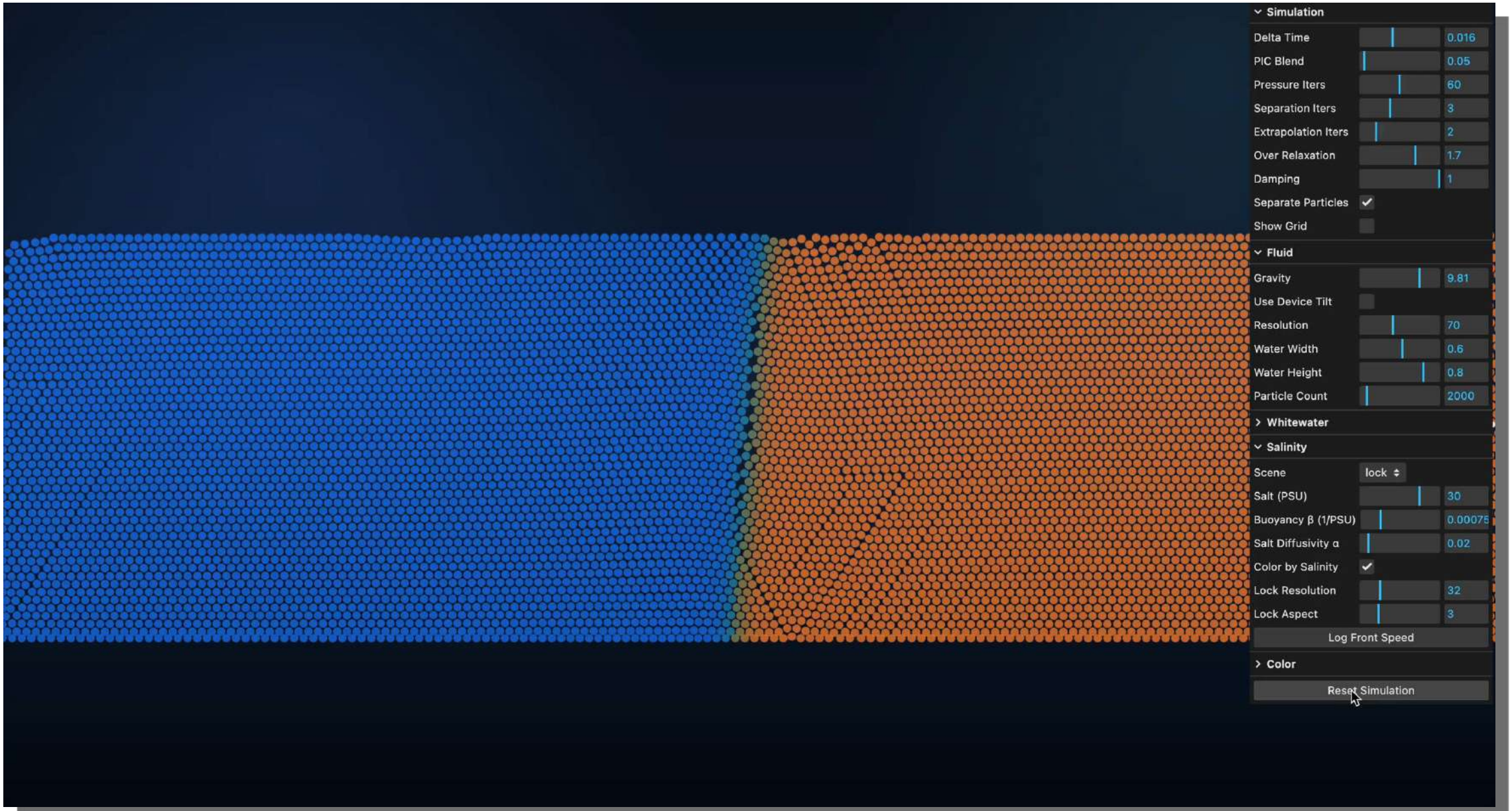


Salinity: boussinesq bouyance per particle



Extend solver by: jeantimex

```
+  // Boussinesq buoyancy. The pressure projection carries a single scalar
+  // density coefficient (cp = density * h / dt in solveIncompressibility), so a
+  // variable-density projection is out of reach; instead salinity enters only
+  // as a body force on the particle velocities, applied *before* the P2G
+  // transfer so the projection converts it into circulation rather than a
+  // uniform acceleration.
+  //
+  // Linear equation of state:  ρ = ρ0 · (1 + beta · (S − Sref))
+  // so the anomaly accelerates heavy water along +g and light water against it.
+  //
+  // The force follows the actual gravity vector rather than a hardcoded −y,
+  // because gravity rotates with device tilt. Damping is deliberately not
+  // re-applied here: applyGravity already damps the velocity this step.
+  applyBuoyancy(dt: number, gravityX: number, gravityY: number, beta: number): void {
+    if (beta === 0.0) return;
+    const gMag = Math.hypot(gravityX, gravityY);
+    if (gMag === 0.0) return;
+    const ux = gravityX / gMag;
+    const uy = gravityY / gMag;
+    const sRef = this.salinityRef;
+    for (let i = 0; i < this.numParticles; i++) {
+      const a = dt * gMag * beta * (this.particleSalinity[i] - sRef);
+      this.particleVel[2 * i]     += a * ux;
+      this.particleVel[2 * i + 1] += a * uy;
+    }
+  }
```


Simulation
Delta Time 0.016
PIC Blend 0.05
Pressure Iters 60
Separation Iters 3
Extrapolation Iters 2
Over Relaxation 1.7
Damping 1
Separate Particles
Show Grid
Fluid
Gravity 9.81
Use Device Tilt
Resolution 70
Water Width 0.6
Water Height 0.8
Particle Count 2000
Whitewater
Salinity
Scene lock
Salt (PSU) 30
Buoyancy β (1/PSU) 0.00075
Salt Diffusivity α 0.02
Color by Salinity
Lock Resolution 32
Lock Aspect 3
Log Front Speed
Color
Reset Simulation