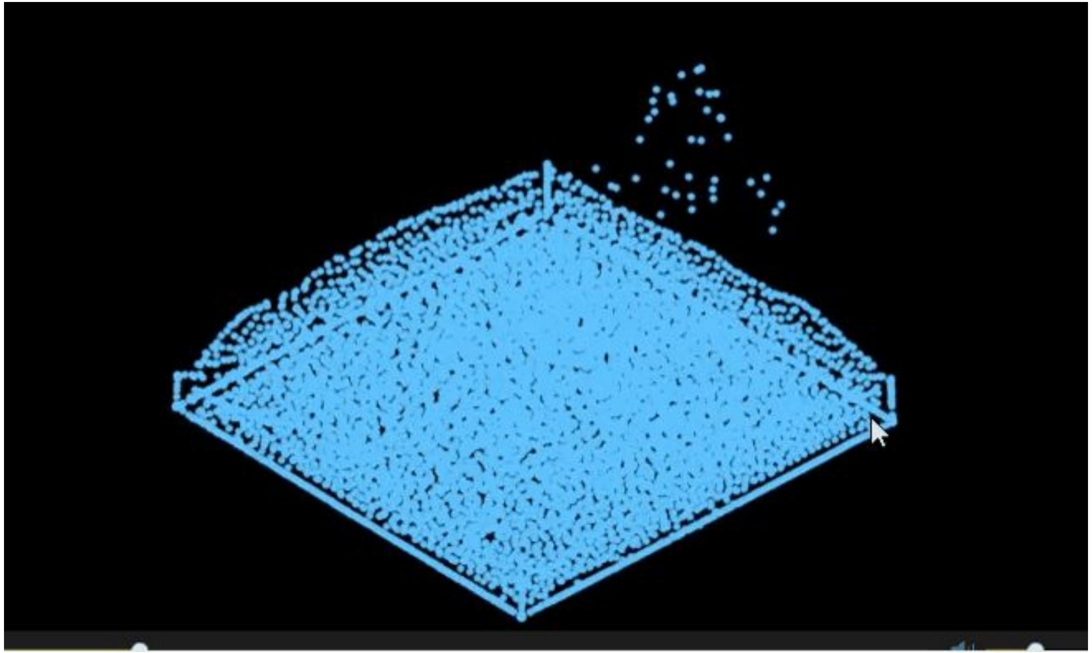
3D Particle Fluid Simulation

Beomseok Park, Shutong Peng

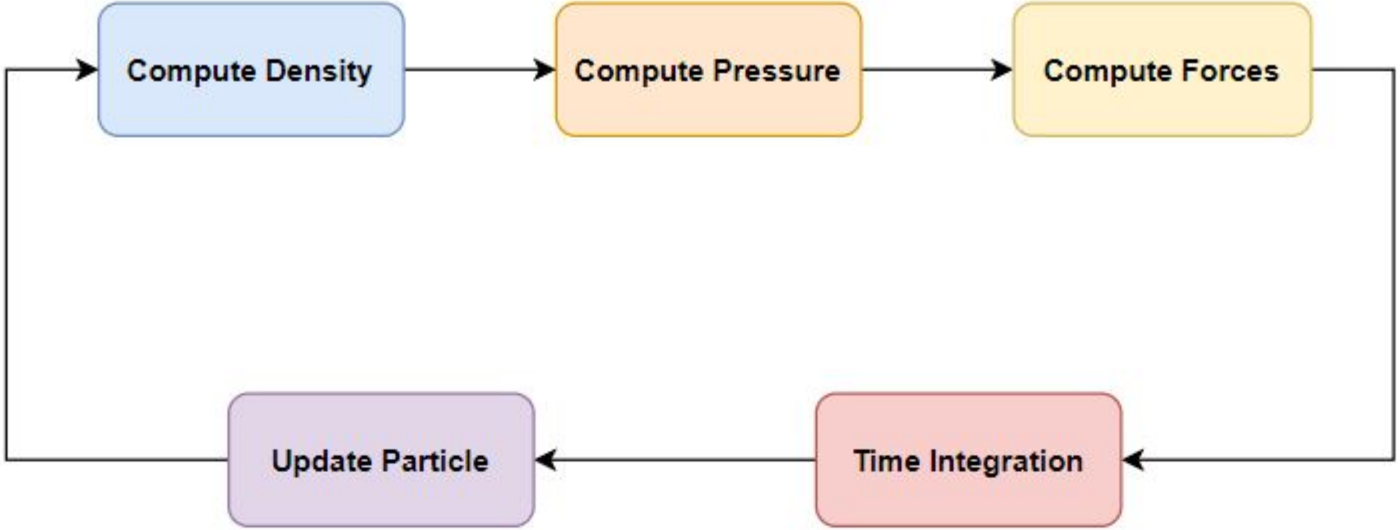
Smoothed Particle Hydrodynamics (SPH)

SPH is a mesh-free, particle-based method for simulating fluid dynamics.



Pipeline of SPH

Particle has five attributes: 1) position, 2) velocity, 3) force, 4) density, and 5) pressure



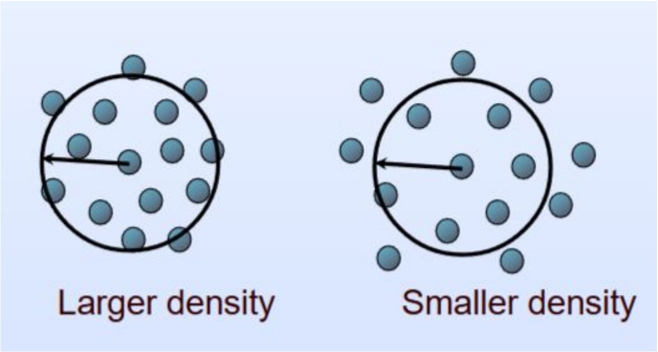
Density Computation

The density ρ_i at particle i is computed by summing contributions from neighboring particles j:

$$\rho_i = \sum_j m_j W_{poly6}(r_{ij}, h)$$

- m_j: mass of particle j
- r_{ij}: distance between particles i and j
- h: smoothing radius
- W_{poly6} kernel smoothing function

$$W_{\text{poly6}}(r, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - r^2)^3, & r \le h \\ 0, & r > h \end{cases}$$



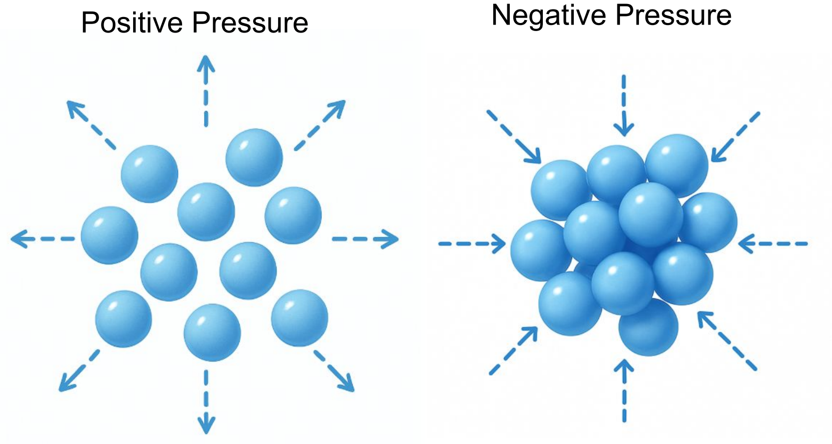
Pressure Computation

Larger density, larger pressure

The pressure p_i at particle i is determined from the density deviation using an equation of state:

$$p_i = k(\rho_i - \rho_0)$$

- k: Gas constant
- ρ_0: rest density



Force Computation

For particle i, the total force $\mathbf{F}_i$ includes pressure, viscosity, and gravity:

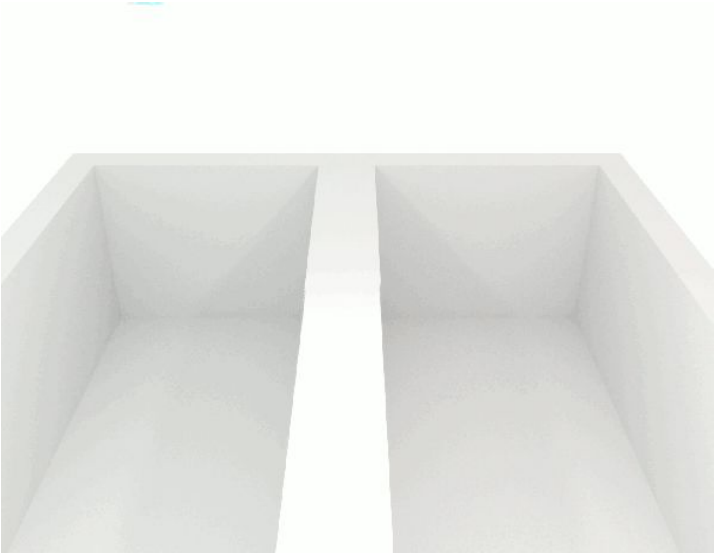
$$\boxed{\mathbf{F}_i = \mathbf{F}_i^{\text{pressure}} + \mathbf{F}_i^{\text{viscosity}} + \mathbf{F}_i^{\text{gravity}}}$$

- $\mathbf{F}_i^{\text{pressure}} = -\sum_{j \neq i} m \frac{p_i + p_j}{2\rho_j} \nabla W_{\text{spiky}}(\mathbf{r}_{ij}, h)$
  - m: mass of particle j
  - p: pressure of a particle
  - ∇W_{spiky}: Spiky Gradient

- $\mathbf{F}_i^{\text{viscosity}} = \sum_{j \neq i} \mu m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{\text{viscosity}}(\mathbf{r}_{ij}, h)$
  - μ: viscosity coefficient
  - v: velocity
  - Viscosity Laplacian $\nabla^2 W_{\text{viscosity}}$

- $\mathbf{F}_i^{\text{gravity}} = \rho_i \mathbf{g}$
  - g: gravitational acceleration vector
  - ρ: density

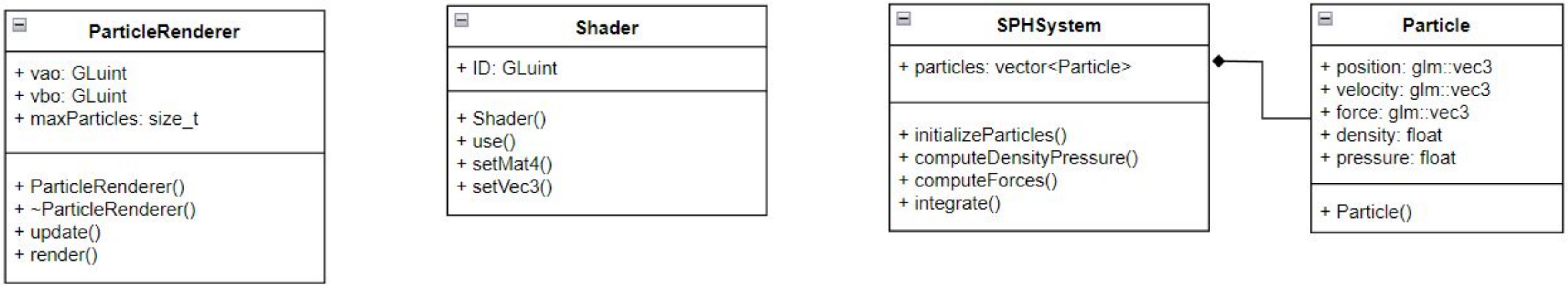
Low Viscosity vs High Viscosity



Time Integration and Update Particle

- Acceleration $a_i = \dfrac{F_i}{\rho_i}$
- Velocity Update: $\mathbf{v}_i^{new} = \mathbf{v}_i^{old} + a_i \Delta t$
- Position Update: $\mathbf{x}_i^{new} = \mathbf{x}_i^{old} + \mathbf{v}_i^{new} \Delta t$

Implementation of SPH Simulation with OpenGL



Parameters for SPH Simulation

```
const int numX = 10;
const int numY = 10;
const int numZ = 10;
const float spacing = 0.01f;
const float REST_DENSITY = 1000.0f;
const float GAS_CONSTANT = 200.0f;
const float VISCOSITY = 3.5f;
const float TIME_STEP = 0.0005f;
const float MASS = 0.001f;
const float SMOOTHING_RADIUS = 0.02f;
const float GRAVITY = -9.81f;
const float DAMPING = -0.3f;
```

Optimize SPH Physics using CUDA

Sequential Processing (CPU)

```
def compute_density_pressure(particles):
    for p_i in particles:
        density = 0.0
        for p_j in particles:
            distance = p_i - p_j
            #...
```

Parallel Processing (CUDA)

```
def compute_density_pressure_cuda(particles):
    thread_idx = blockIdx.x * blockDim.x + threadIdx.x
    p_i = particles[thread_idx]
    density = 0.0
    for p_j in particles:
        distance = p_i - p_j
        #...
```

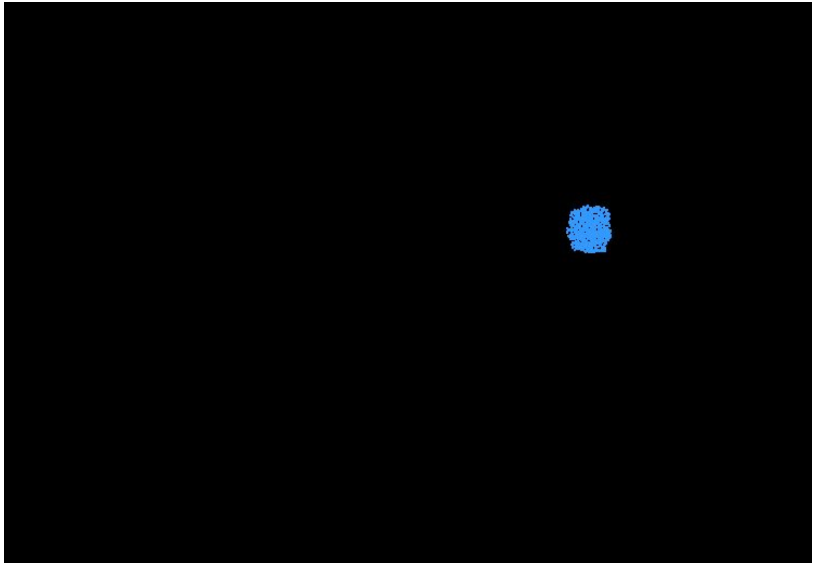
Qualitative Results

Approach	# Particles	Average Time (sec)	Total Time (sec)
CPU	125	0.004	4.161
CUDA	**125**	**0.0014**	**1.471**
CPU	1,000	0.169	169.067
CUDA	**1,000**	**0.0017**	**1.742**

- Average time is an average of rendering time across 1,000 frames
- Total time is measured a sum of rendering time up to 1,000 frames

Qualitative results

CPU-based SPH with 1,000 particles

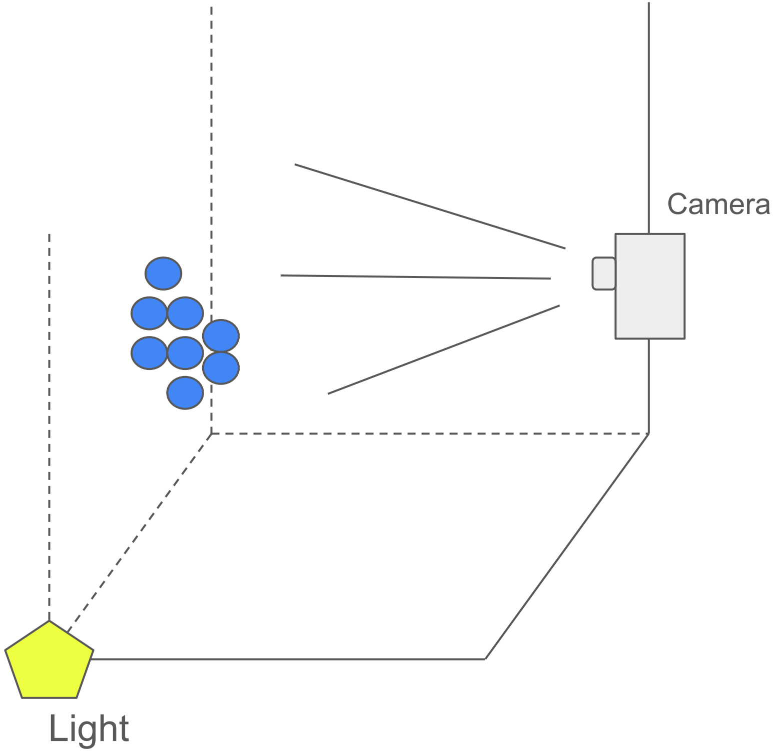


CUDA-based SPH with 1,000 particles

Camera and Light Setting

```
// Camera setup
glm::vec3 cameraPos    = glm::vec3(2.0f, 1.0f, 0.5f);   // Camera position in world space
glm::vec3 cameraTarget= glm::vec3(0.5f, 0.5f, 0.5f);   // Look at the center of the scene
glm::vec3 up           = glm::vec3(0.0f, 1.0f, 0.0f);   // Up vector
glm::mat4 view = glm::lookAt(cameraPos, cameraTarget, up);
glm::mat4 projection = glm::perspective(glm::radians(45.0f), 800.0f/600.0f, 0.01f, 100.0f);
glm::mat4 model = glm::mat4(1.0f); // Identity model matrix (no additional transform)
```

```
// Pass camera and lighting uniforms to shader
shader.setVec3("lightPos", 0.0f, 0.0f, 0.0f);    // Light source position (e.g., a corner of the scene)
shader.setVec3("viewPos", cameraPos.x, cameraPos.y, cameraPos.z); // Camera position (for viewDir)
shader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);  // Light color (white)
shader.setVec3("waterColor", 0.2f, 0.6f, 1.0f);  // Base water color (slightly blue)
```



Point Sprite

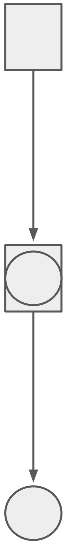
```
// Reconstruct normalized device coordinates of this fragment within the point sprite
// gl_PointCoord ranges from 0 to 1 across the sprite
vec2 uv = gl_PointCoord * 2.0 - 1.0;      // map to [-1, 1]
float r2 = uv.x*uv.x + uv.y*uv.y;
if (r2 > 1.0) {
    // Discard fragments outside the circle (to make a round particle)
    discard;
}

// Compute the normal vector of the sphere at this fragment (in view space)
float zComponent = sqrt(1.0 - r2);
vec3 normalView = vec3(uv.x, uv.y, zComponent);

// Transform normal to world space (ignore translation, use invView rotation)
vec3 normalWorld = normalize((invView * vec4(normalView, 0.0)).xyz);

// Compute fragment position in world space (approximate as particle center + normal * radius)
float radius = 0.01; // must match the radius used in vertex shader
vec3 fragPosWorld = WorldPos + normalWorld * radius;

// Compute view direction and light direction (in world space)
vec3 viewDir = normalize(viewPos - fragPosWorld);   // from fragment toward camera
vec3 lightDir = normalize(lightPos - fragPosWorld); // from fragment toward light source
```

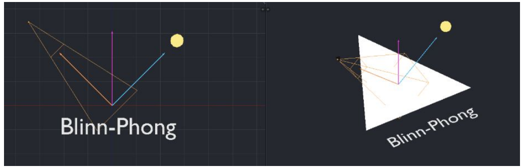


Lighting

```
// --- Lighting calculations ---
// Ambient (small base light)
vec3 ambient = 0.5 * lightColor * waterColor;

// Diffuse (Lambertian)
float diff = max(dot(normalWorld, lightDir), 0.0);
vec3 diffuse = 0.6 * diff * lightColor * waterColor;

// Specular (Blinn-Phong)
vec3 reflectDir = reflect(-lightDir, normalWorld);
float specStrength = 0.2;
float shininess = 32.0;
float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
vec3 specular = specStrength * spec * lightColor;
```



Ambient

$$\mathbf{L}_{\text{ambient}} = k_a\, \mathbf{L}_{\text{light}} \otimes \mathbf{C}_{\text{water}}$$

Diffuse

$$\mathbf{L}_{\text{diffuse}} = k_d\, \big(\max(\mathbf{N}\cdot\mathbf{L},\, 0)\big)\, \mathbf{L}_{\text{light}} \otimes \mathbf{C}_{\text{water}}$$

Blinn-Phong

$$\mathbf{R} = \text{reflect}(-\mathbf{L},\, \mathbf{N}) = -\mathbf{L} - 2\,(\mathbf{N}\cdot(-\mathbf{L}))\,\mathbf{N} = \mathbf{L} - 2(\mathbf{N}\cdot\mathbf{L})\,\mathbf{N}$$

$$\text{spec} = \big(\max(\mathbf{V}\cdot\mathbf{R},\, 0)\big)^{\alpha}$$

$$\mathbf{L}_{\text{specular}} = k_s \times \text{spec} \times \mathbf{L}_{\text{light}}$$

Reflection & Refraction

```
// --- Environment reflection & refraction ---
// Compute reflection and refraction vectors for environment mapping
vec3 I = normalize(fragPosWorld - viewPos);  // incident view ray (from camera to frag)
vec3 reflectVec = reflect(I, normalWorld);
// Refractive index of water ~1.33, compute refraction (air->water)
vec3 refractVec = refract(I, normalWorld, 1.0/1.33);
// Sample the environment cubemap for reflection and refraction colors
vec3 reflectColor = texture(skybox, reflectVec).rgb;
vec3 refractColor = texture(skybox, refractVec).rgb;
// vec3 reflectColor = vec3(0.8, 0.9, 1.0);
// vec3 refractColor = waterColor * 0.6;
```

Reflection $$\mathbf{R} = \mathbf{V} - 2(\mathbf{N} \cdot \mathbf{V})\mathbf{N}$$

Refraction $$\mathbf{T} = \eta \mathbf{V} - \left(\eta(\mathbf{N} \cdot \mathbf{V}) + \sqrt{1 - \eta^2\left(1 - (\mathbf{N} \cdot \mathbf{V})^2\right)}\right)\mathbf{N}$$

Simulate Abortion

$$\mathbf{C}_{\text{refract}} \leftarrow \mathbf{C}_{\text{refract}} \otimes \mathbf{C}_{\text{water}}$$

```
// Apply water tint to refracted color (simulate absorption)
refractColor *= waterColor;

// Fresnel factor for reflectance (using Schlick's approximation)
float cosTheta = max(dot(normalWorld, -I), 0.0);
float fresnelFactor = 0.04 + (1.0 - 0.04) * pow(1.0 - cosTheta, 5.0);

// Mix reflection and refraction based on Fresnel (angle-dependent)
vec3 envColor = mix(refractColor, reflectColor, pow(fresnelFactor, 0.3));
```

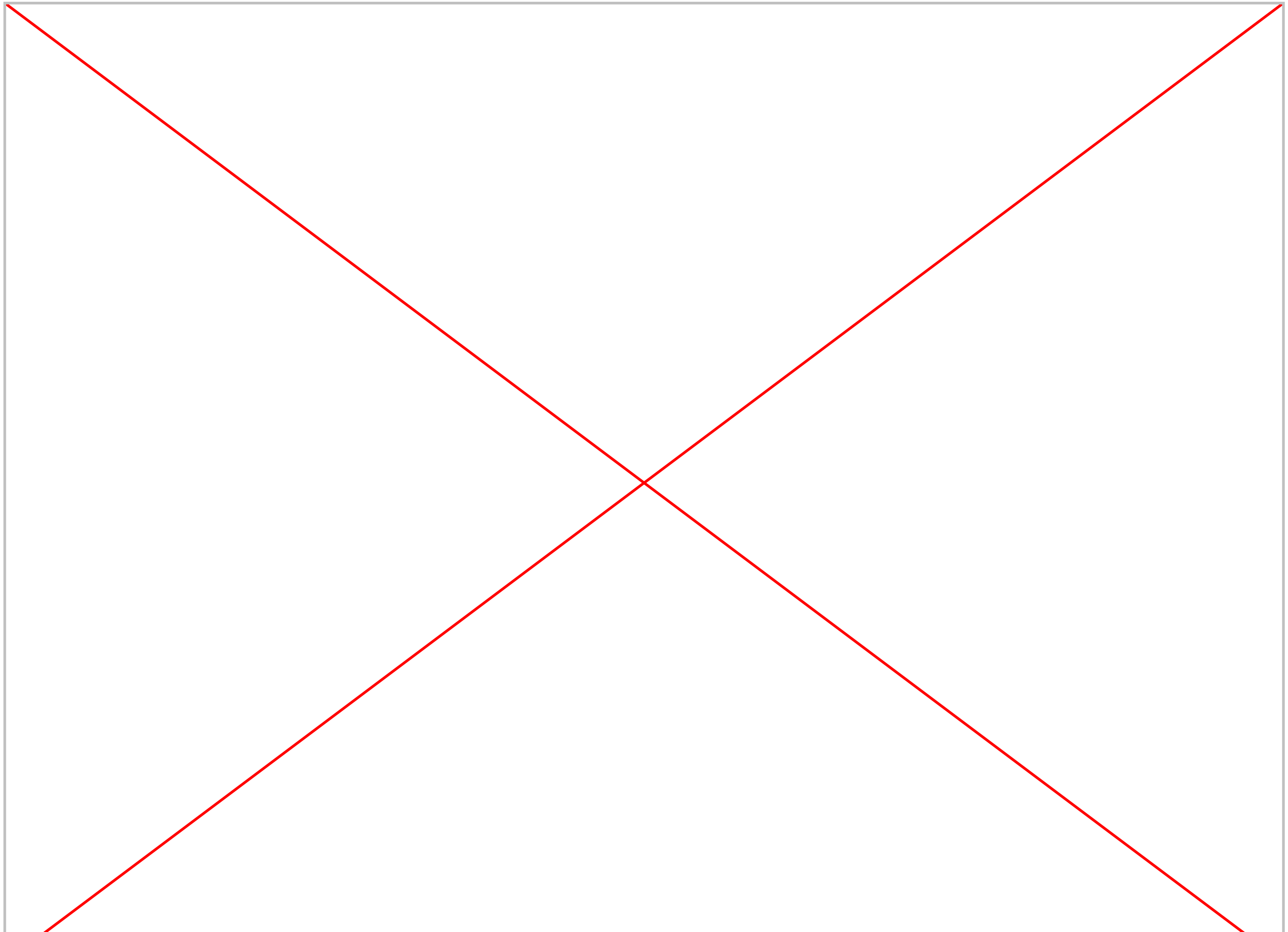
$$F(\theta) = F_0 + (1 - F_0)\,(1 - \cos\theta)^5$$

$$\mathbf{C}_{\text{env}} = (1 - t)\,\mathbf{C}_{\text{refract}} + t\,\mathbf{C}_{\text{reflect}}, \quad t = F(\theta)^{0.3}$$

Foam Effect

```
// --- Foam effect ---
// Mix in foam (white) based on foam factor
finalColor = mix(finalColor, vec3(1.0, 1.0, 1.0), pow(Foam, 1.5));

// Increase opacity if foam is present (foam makes water more opaque)
float baseAlpha = 0.2;
float alpha = mix(baseAlpha, 1.0, Foam);
```



Future Work

1. Thicken and shade the tank walls as a semi-transparent glass box—with Fresnel-based reflection/refraction and a raised bottom rim—to make container real water-tank realism.

2. Then render the particles into a depth texture, apply a bilateral or Gaussian blur to smooth out small ripples, and use that softened depth map for thickness and refraction so the fluid appears like water instead of jelly.

Thank you

For detail, refers to my blog: https://baampark.github.io