SPH Fluid Simulation

INOVATION
PRACTICE

T.O.P

Motivation

[Dev Archives: The Engine Behind Crimson Desert | GDC 2025]

Related Work

- Bump Map
- FFT
- CFD
- Wave particles
- Data-Driven method

Eulerian approach: grid-based

Lagrangian approach : particle-based

Routine

- Simulating: SPH(Smoothed Particle Hydrodynamics) [Müller, 2003]
- Rendering: PBF(Position-based Fluid) [Green, 2013]
- Interacting

Implement

- Engine: Unity 2022.3.57f1c1 LTS
- Render Pipeline: URP 14
- Use Render Feature to dispatch custom pass
- Collaboration: Git + Github

SPH

- **Navier-Stokes Equation**

$$\nabla \cdot \vec{U} = 0$$

$$\frac{\partial \vec{U}}{\partial t} + \vec{U} \cdot \nabla \vec{U} + \frac{\nabla p}{\rho} - \mu \frac{\nabla^2 \vec{U}}{\rho} - \vec{g} = 0 \xrightarrow[\text{method}]{\text{Particle-based}} \frac{D\vec{U}}{Dt} = \frac{1}{\rho}(\rho \vec{g} - \nabla p + \mu \nabla^2 \vec{U})$$

- **SPH can interpolate each quantities**

$$a_i = \frac{D\vec{U}_i}{Dt} = \frac{f_i^{external} + f_i^{pressure} + f_i^{viscosity}}{\rho_i}$$

$$A_S(r) = \sum_j A_j \frac{m_j}{\rho_j} W(r - r_j, h)$$

SPH: Derivatives

- **Derivatives can be easily computed by simply apply it to the kernel**
  - **Gradient**

$$\nabla A_S(r) = \nabla \sum_j A_j \frac{m_j}{\rho_j} W(r - r_j, h) = \sum_j A_j \frac{m_j}{\rho_j} \nabla W(r - r_j, h)$$

  - **Laplacian**

$$\nabla^2 A_S(r) = \nabla^2 \sum_j A_j \frac{m_j}{\rho_j} W(r - r_j, h) = \sum_j A_j \frac{m_j}{\rho_j} \nabla^2 W(r - r_j, h)$$

SPH: Compute Forces

- **Gravity**

$$f_i^{external} = \rho_i g$$

- **Pressure**

$$f_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_i - r_j, h) \qquad \nabla W_{\text{spiky}} = -\frac{45}{\pi h^6} \begin{cases} (h-r)^2 e_r & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

- **Viscosity**

$$f_i^{viscosity} = \mu \sum_j m_j \frac{\vec{U}_i - \vec{U}_j}{\rho_j} \nabla^2 W(r_i - r_j, h) \qquad \nabla^2 W_{\text{viscosity}}(r, h) = \frac{45}{\pi h^6} \begin{cases} h - r & 0 \le r \le h \\ 0 & \text{otherwise} \end{cases}$$

SPH: Spatial Hashing

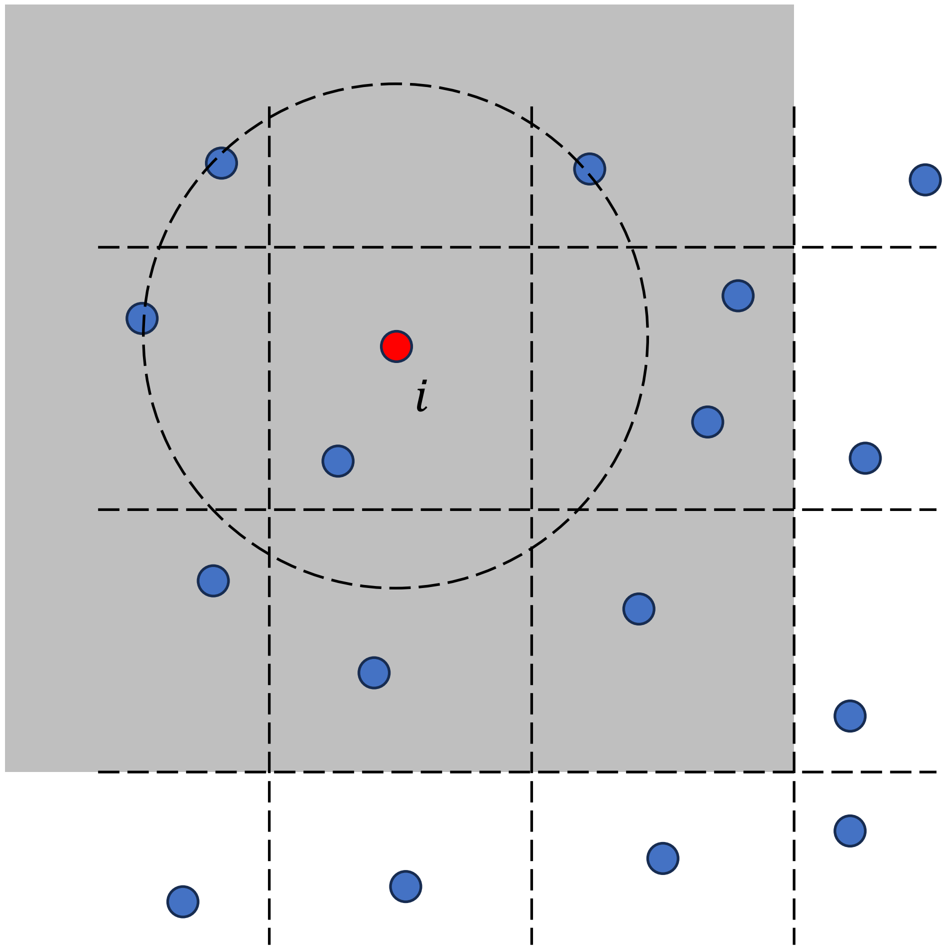
- **Separate space into cells**

$$hash(X_{grid}) = (iP_1)XOR(jP_2)XOR(kP_3) mod\, N$$

$$P_1 = 73856093,\ P_2 = 19349663,\ P_3 = 83492791$$

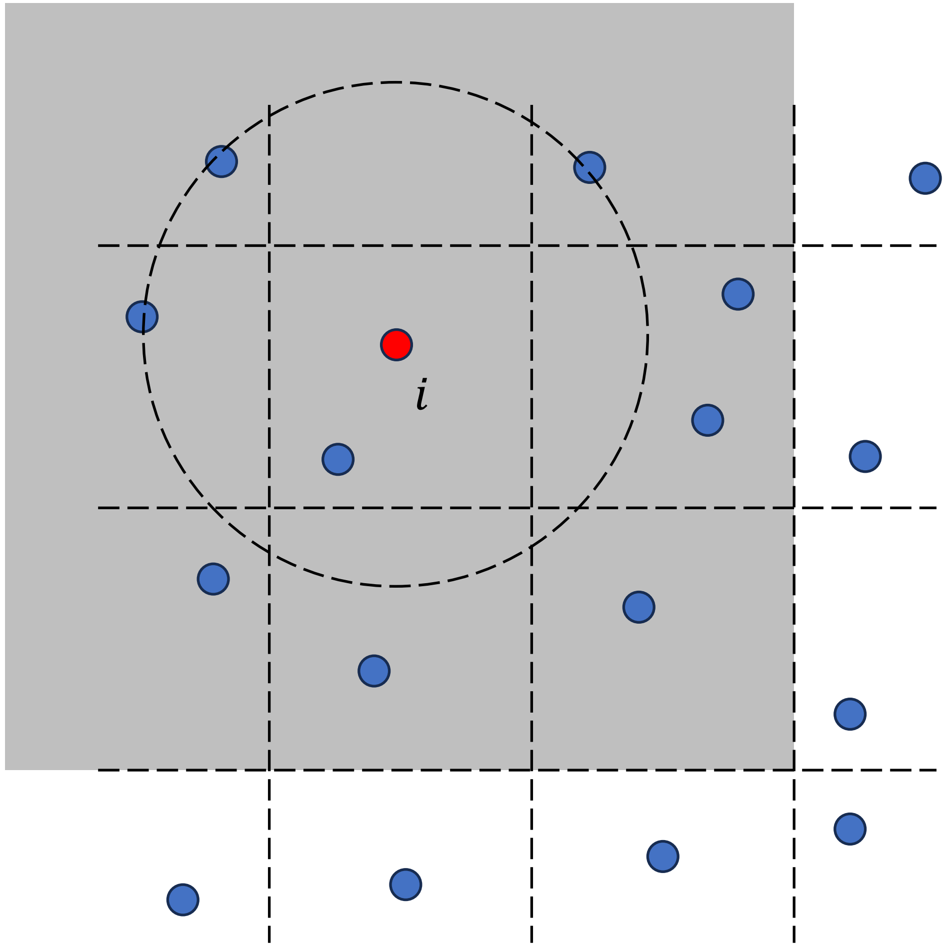
- **Map particle position to cell index**

$$pos_{grid} = \left(\frac{x}{L_x}, \frac{y}{L_y}, \frac{z}{L_z}\right)$$



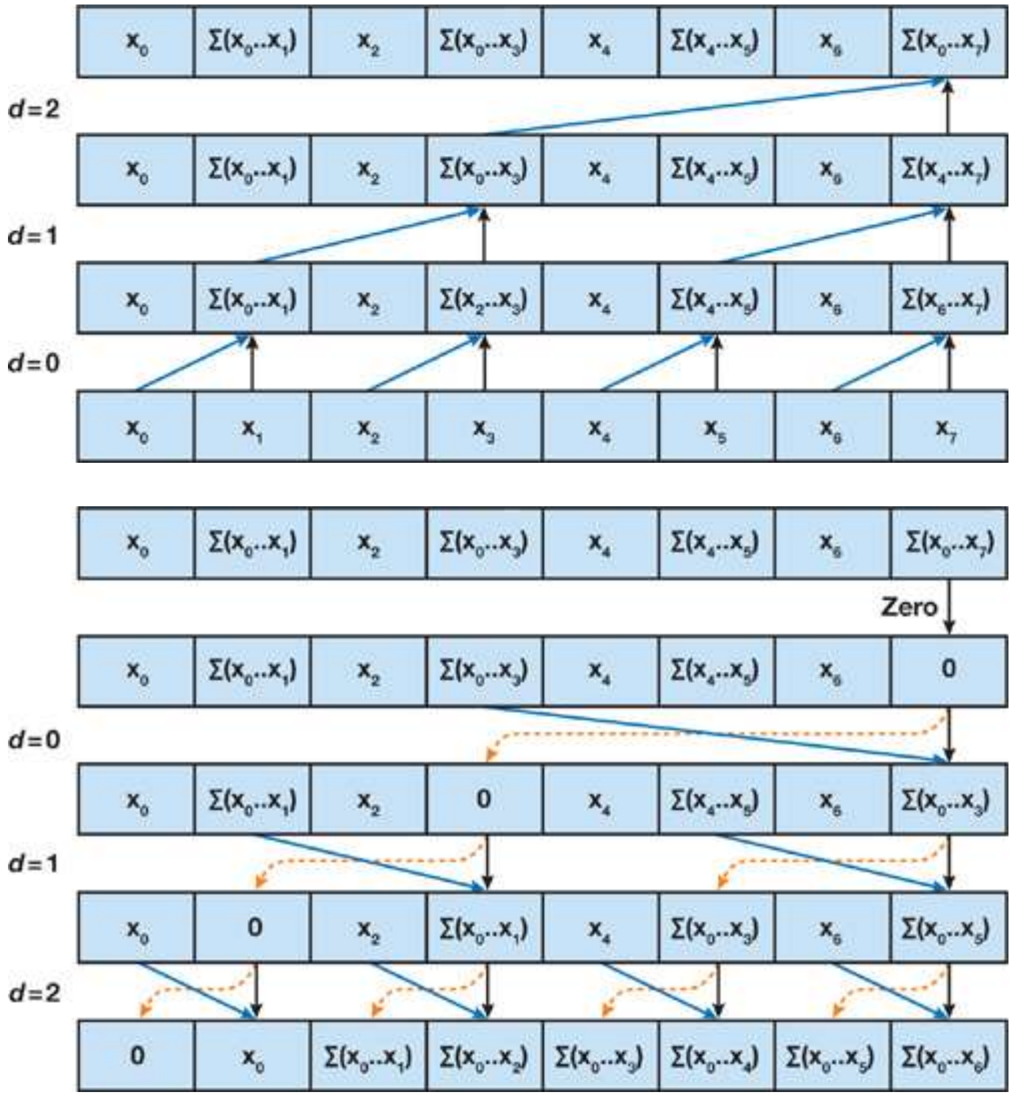
SPH: Spatial Hashing Cont.

- Separate space into cells
- Map the cell index to a hash table
- For a single particle compute its and neighbor cell idx
- Query particles in hash table



SPH: Radix Sort

- As shown in the figure
- Bottom-up & Up-Bottom
- Parallax Execution



INOVATION
PRACTICE

T.O.P

SPH: Time Integration

- **Explicit Euler: update velocity and position synchronously**

$$v_i(t+\Delta t) = v_i(t) + a_i(t)\Delta t$$
$$x_i(t+\Delta t) = x_i(t) + v_i(t)\Delta t$$

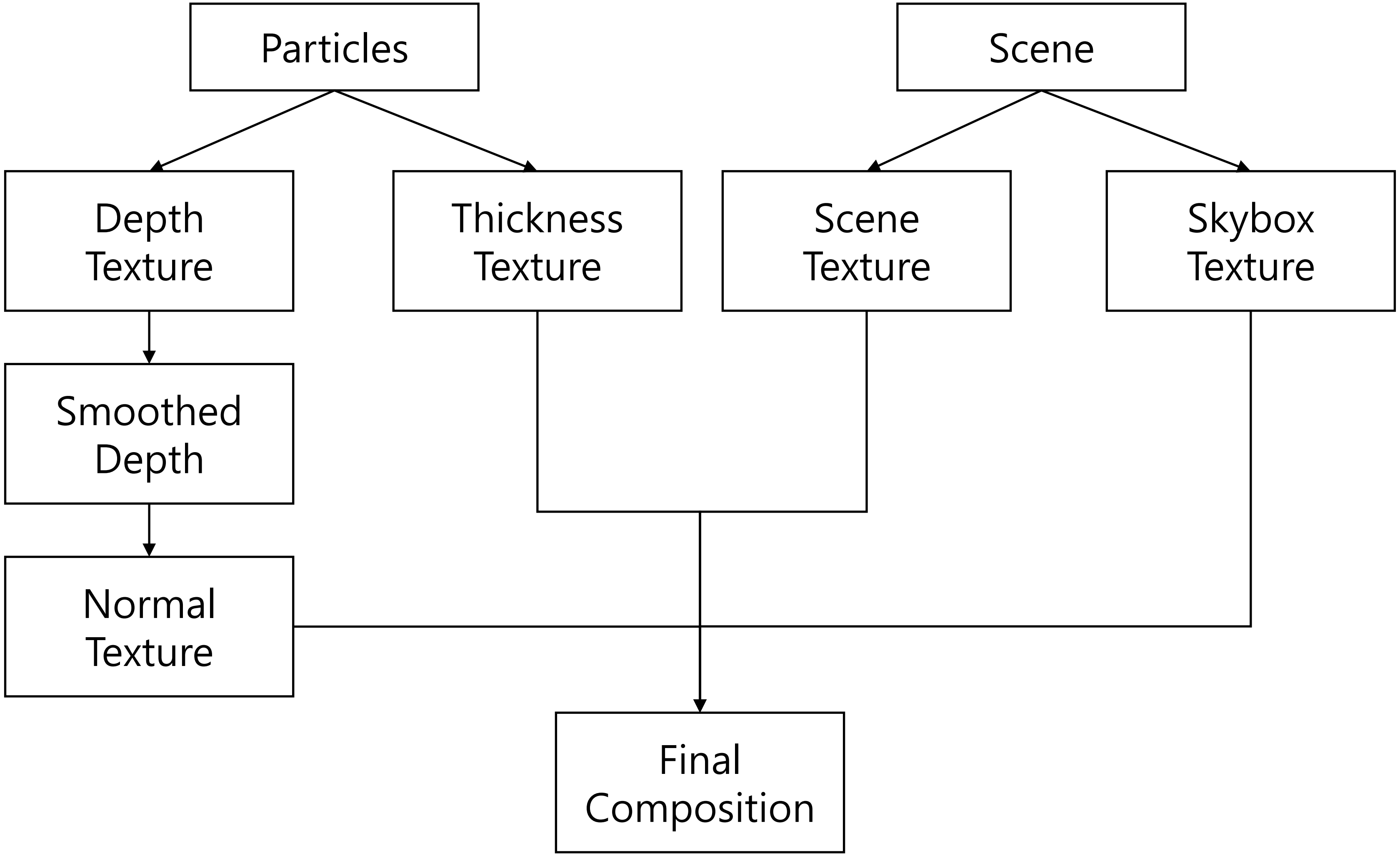
- **Semi-Implicit Euler: use updated velocity to update position ← We pick this!**

$$v_i(t+\Delta t) = v_i(t) + a_i(t)\Delta t$$
$$x_i(t+\Delta t) = x_i(t) + v_i(t+\Delta t)\Delta t$$

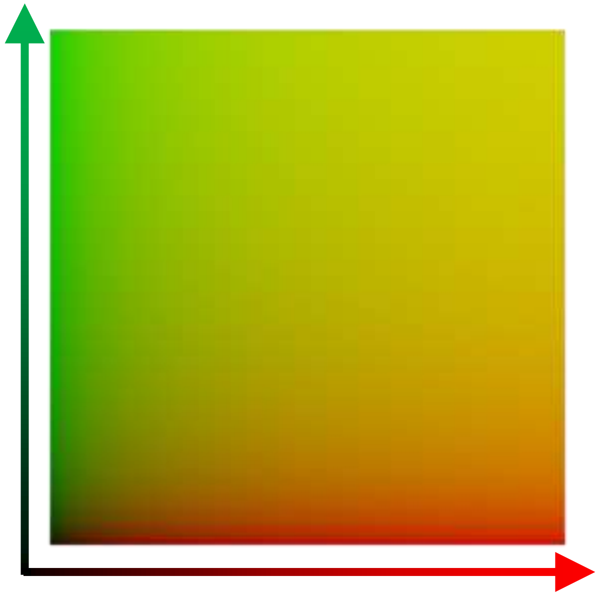
- **Leap-Frog: update velocity and position alternately**

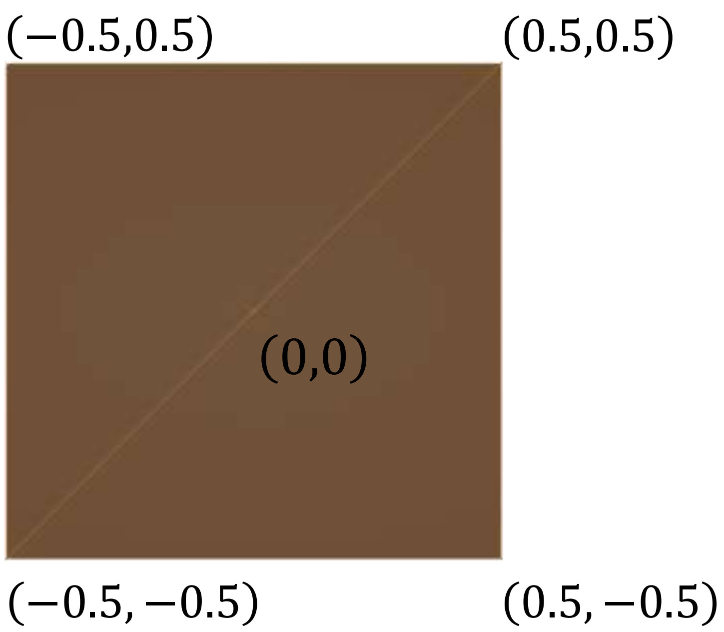
$$v_i\left(t+\frac{\Delta t}{2}\right) = v_i\left(t-\frac{\Delta t}{2}\right) + a_i(t)\Delta t$$
$$x_i(t+\Delta t) = x_i(t) + v_i(t+\frac{\Delta t}{2})\Delta t$$

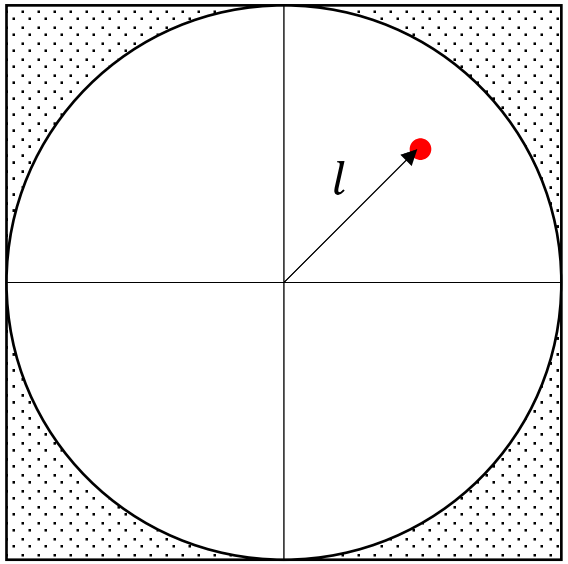
PBF


Particles
Scene
Depth Texture
Thickness Texture
Scene Texture
Skybox Texture
Smoothed Depth
Normal Texture
Final Composition

PBF: Quad

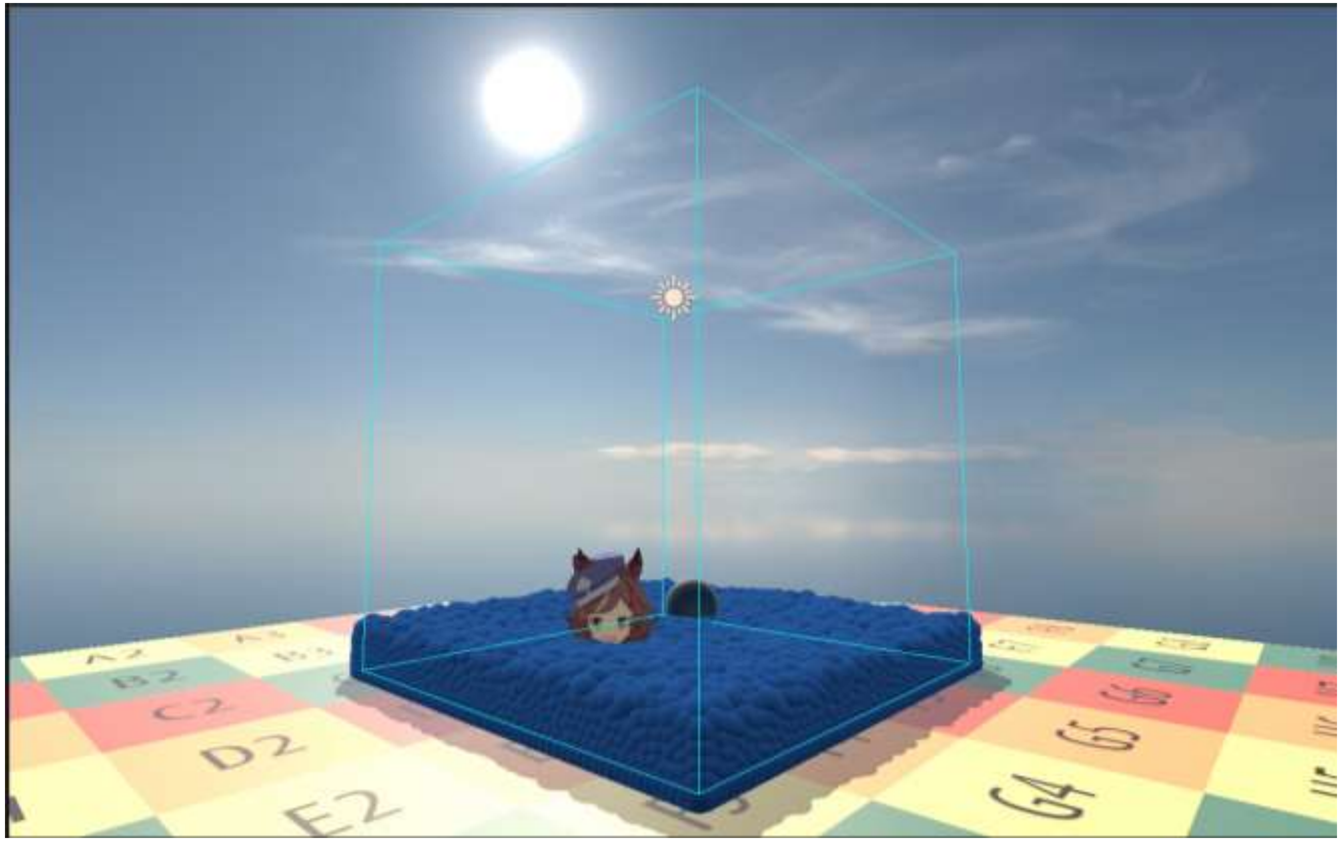






$$posObjectSpace.xy = UV - 0.5$$

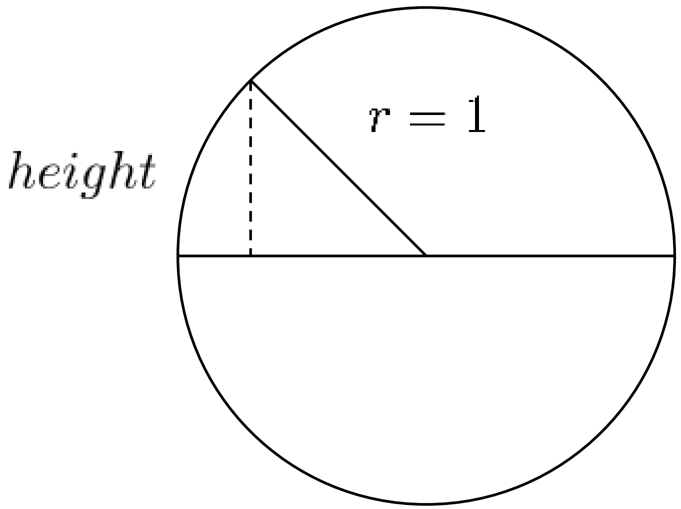
PBF: Fluid Particle



PBF: Particle Depth

- **Assume the quad as a unit sphere, the height can given by:**

$$height = \sqrt{1 - u^2 - v^2} \qquad * : uv = originaluv * 2 - 1$$



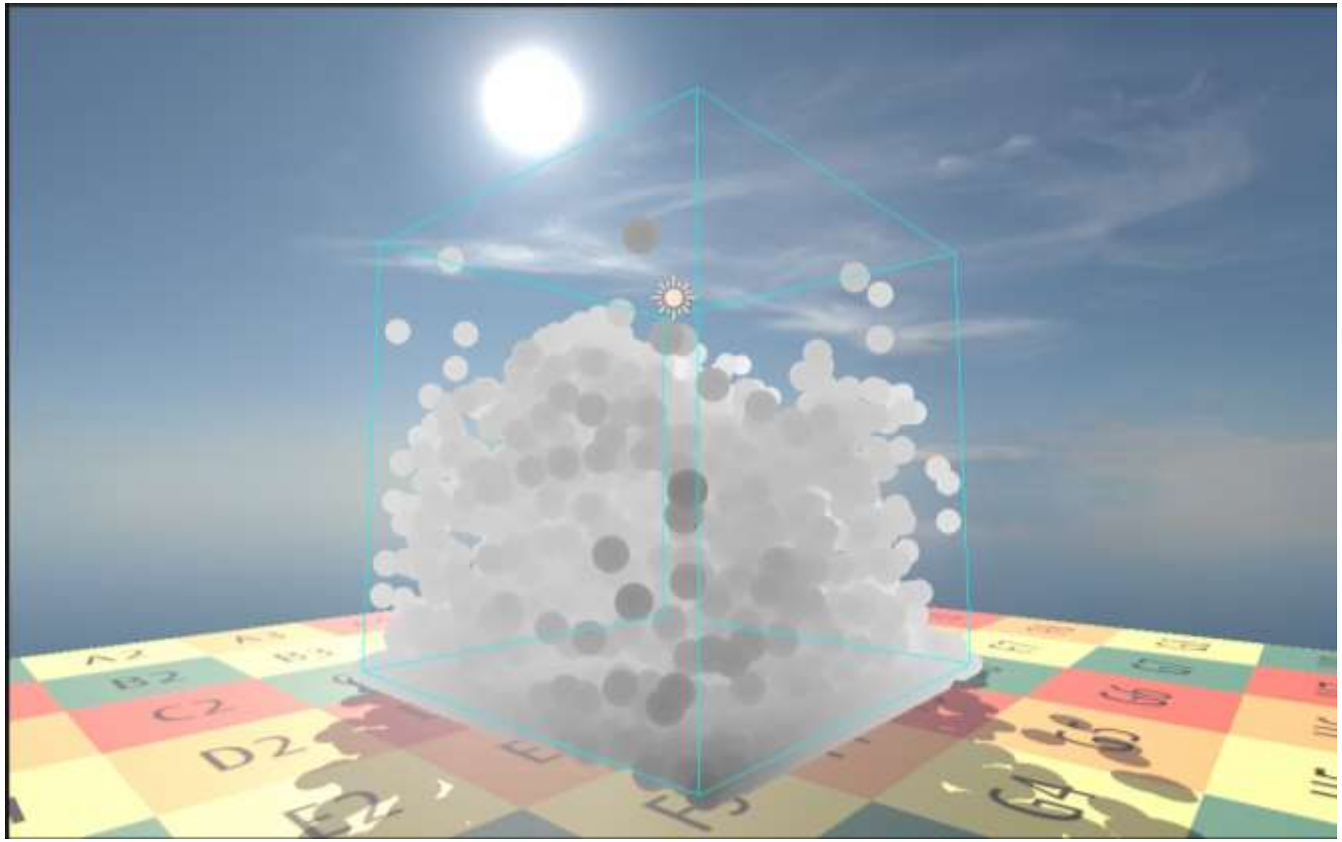
- **So fragPos of "sphere" can be represented as:**

$$fragPos = viewSpacePos + SphereNormalVS * 0.5$$

- **Transform to clipSpace and derive depth:**

$$depth = clipSpacePos.z/clipSpacePos.w$$

PBF: Fluid Depth



PBF: Particle Thickness

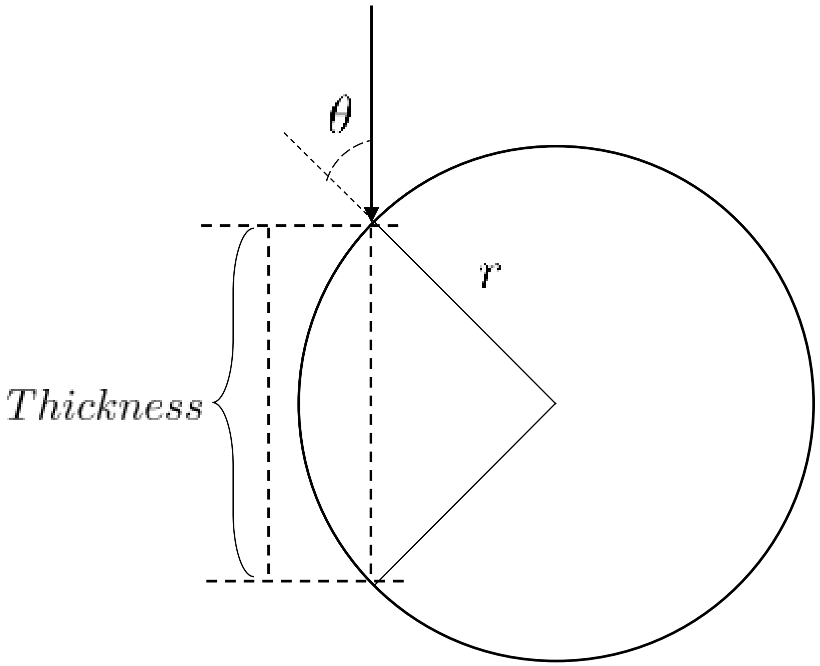
- **Thickness is a param given by:**

$$Thickness = 2r\cos\theta$$

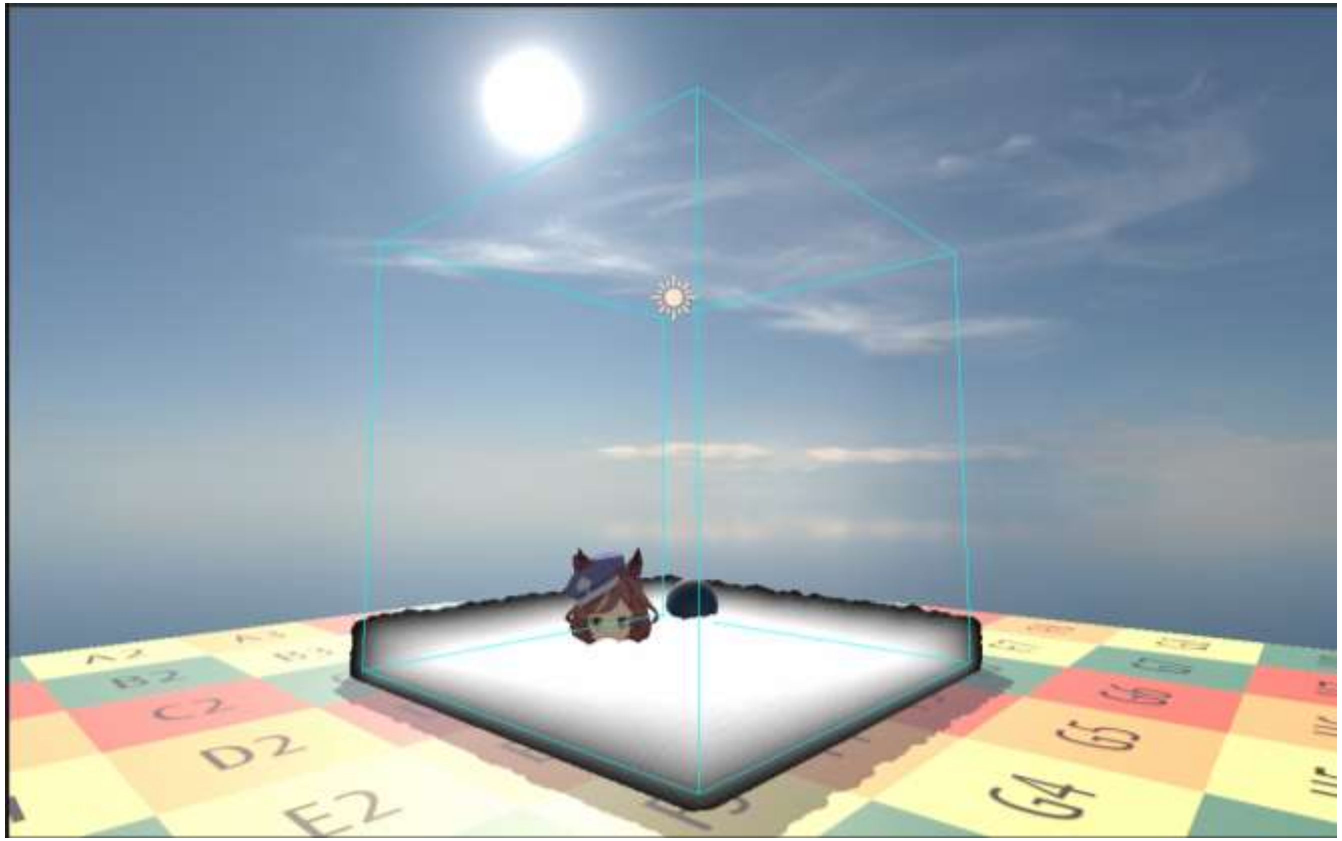
$$\cos\theta = dot(viewSpaceNormal, viewSpaceviewDir)$$

- **Luckily in view space, viewDir is fixed (0,0,1), hence:**

$$Thickness = 2r|viewSpaceNormal.z|$$



PBF: Fluid Thickness



PBF: Reconstruct Normal

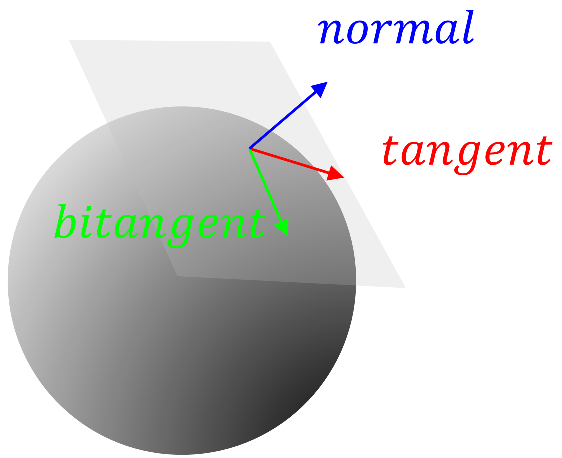
- **Normal is a vector that always perpendicular to surface**
  - **Choose 2 vector in tangent plane to make a cross product**

$$\mathbf{n} = \mathbf{t} \times \mathbf{b}$$

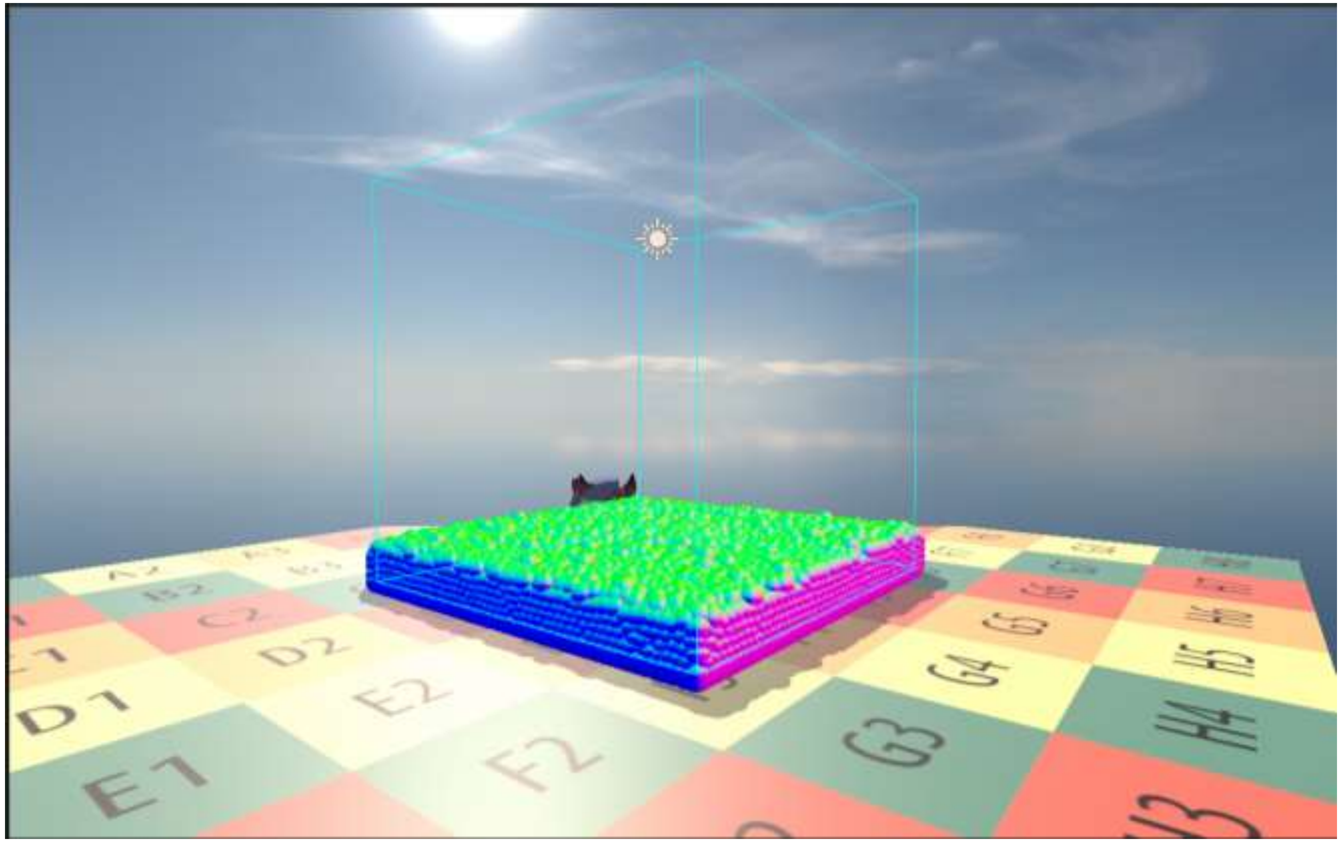
- **Tangent and Bi-tangent can be found by position difference**

$$tangent = \frac{\partial f}{\partial x} \approx \min\left(\frac{f(x+\Delta x, y) - f(x,y)}{\Delta x}, \frac{f(x,y) - f(x-\Delta x, y)}{\Delta x}\right)$$

$$bitangent = \frac{\partial f}{\partial y} \approx \min\left(\frac{f(x, y+\Delta y) - f(x,y)}{\Delta y}, \frac{f(x,y) - f(x, y-\Delta y)}{\Delta y}\right)$$



PBF: Fluid Normal



PBF: Bilateral Filter

- **Bilateral Filter:**

$$I^{filtered}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$$

$$W_p = \sum_{x_i \in \Omega} f_r(||I(x_i) - I(x)||) g_s(||x_i - x||)$$

- **Use both spatial and value weights**

$$w(i,j) = f_r(i,j) g_s(i,j) = \underbrace{\exp\left(-\frac{(\Delta z)^2}{2\sigma_r^2}\right)}_{Value} \underbrace{\exp\left(-\frac{i^2 + j^2}{2\sigma_s^2}\right)}_{Spatial}$$

PBF: Bilateral Filter Cont.

- **Why bilateral?**
  - **Keep fluid edges(foreground from bg)**
  - **Can be split into horizontal and vertical**
- **Can have artifacts but not that serious**

Algorithm 1: Bilateral Filter (1D Pass)

Data: Depth texture D_{in}, kernel radius R, constants σ_d, σ_r

Result: Smoothed depth texture D_{out}

foreach *pixel* (x, y) *in parallel* **do**

    $z_0 \leftarrow D_{\text{in}}(x, y)$;

    $sum \leftarrow 0$, $w_{sum} \leftarrow 0$;

    for $i = -R$ **to** R **do**

        if *horizontal pass* **then**

            $z_i \leftarrow D_{\text{in}}(x + i, y)$

        else if *vertical pass* **then**

            $z_i \leftarrow D_{\text{in}}(x, y + i)$

        end

        $g \leftarrow \exp(-i^2/2\sigma_d^2)$;

        $r \leftarrow \exp(-(z_i - z_0)^2/2\sigma_r^2)$;

        $w \leftarrow g \cdot r$;

        $sum \mathrel{+}= w \cdot z_i$;

        $w_{sum} \mathrel{+}= w$;

    end

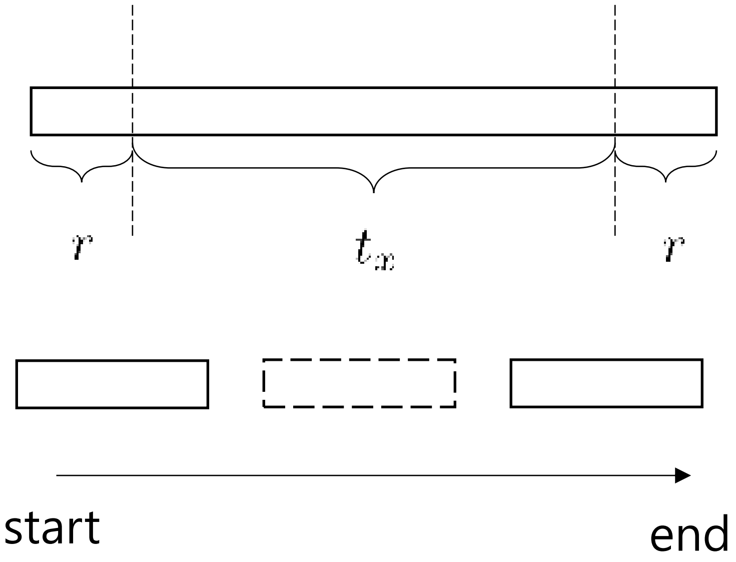
    $D_{\text{out}}(x, y) \leftarrow \frac{sum}{w_{sum}}$;

end

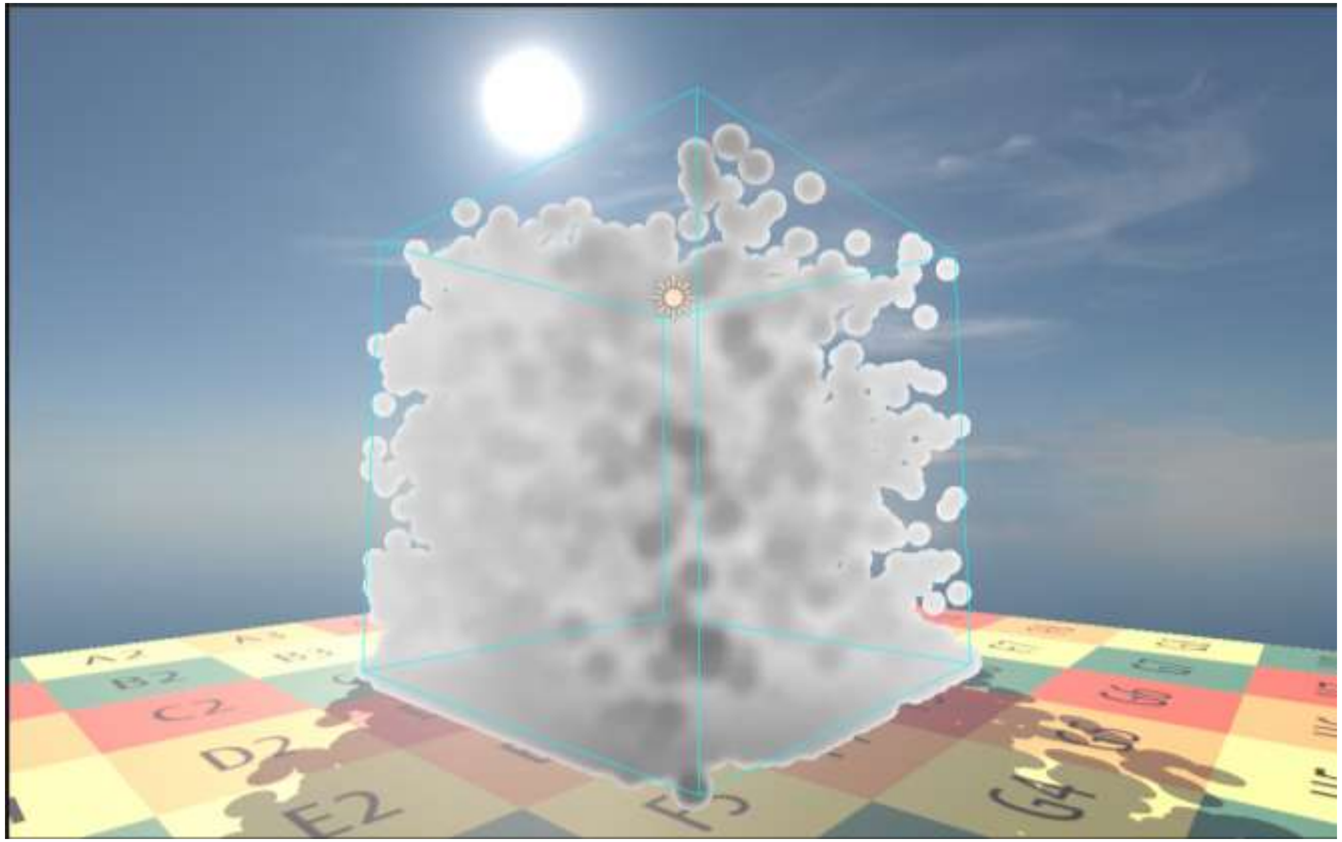
PBF: Cache Depth

- **Query samples frequently is costly**
  - **For each thread group cache the depth**
  - **GroupShared memory should be tx+2r**
  - **For caching do remap from 0~tx-1 to 0~tx+2r-1**
  - **For fetching do the reverse**

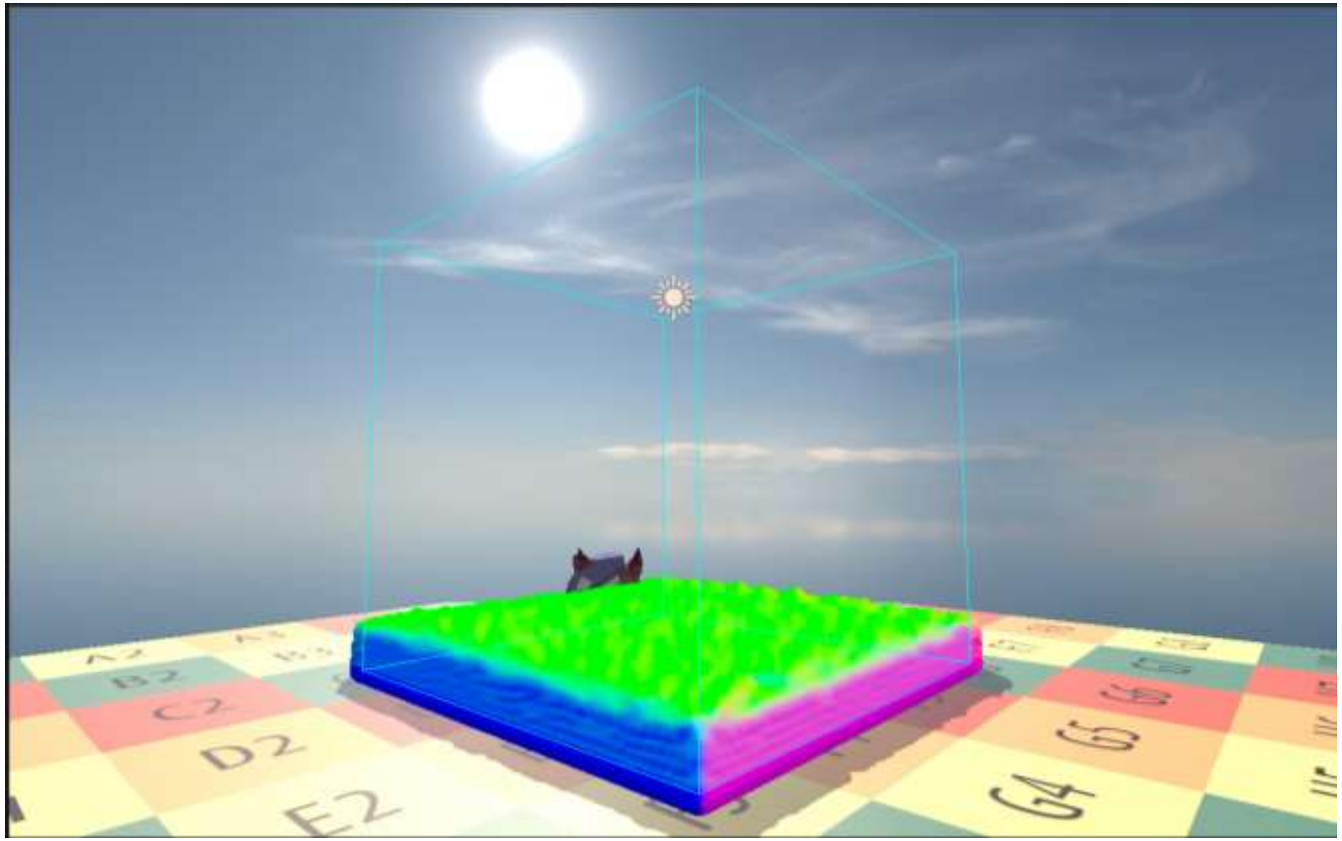
$$\text{threadGroupSize} = (t_x, 1, 1) \quad \text{kernelSize} = 2r + 1$$



PBF: Fluid Smoothed Depth

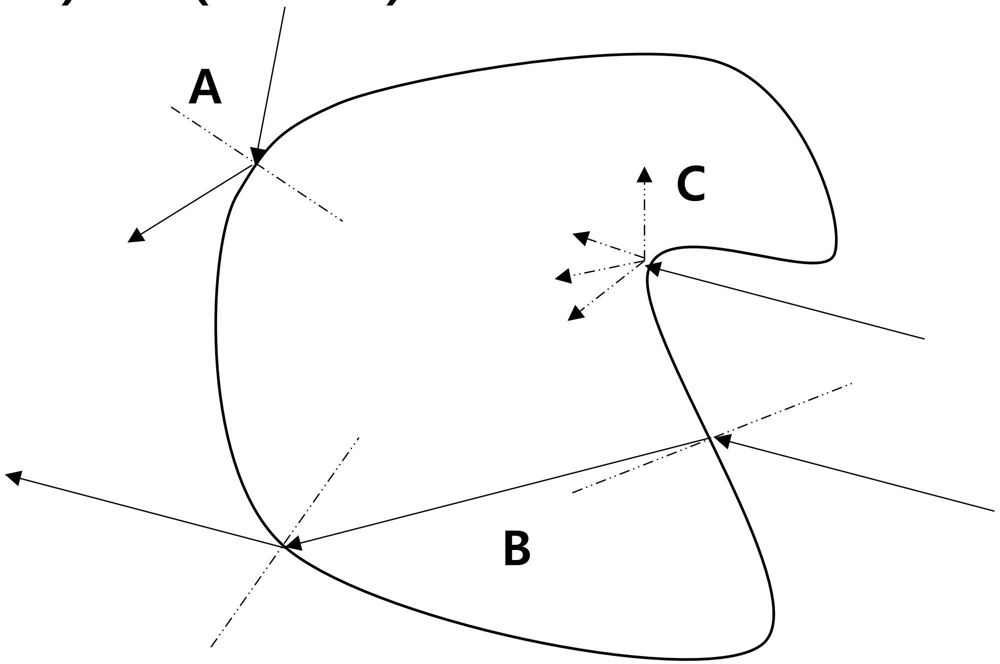


PBF: Fluid Smoothed Normal



PBF: Composite

- Shading Translucent Water : A(Reflect) + B(Refract) + C(Scatter)
  - Reflect: Cube map reflection
  - Refract: Scene texture
  - Shadows: Shadow mapping
  - Caustics: Photon mapping
  - Foam: Wait for solution!



PBF: Reflect & Refract

- **Shading Translucent Water**
  - **Reflect: Cube map reflection**

```
float3 O = reflect(I, N);
float3 reflectColor = texCUBE(_SkyboxTexture, O).rgb;
```

  - **Refract: Scene texture refraction**

```
float3 R = refract(I, N, eta);
float2 refractCoord = screenUV + R.xy*refractScalar;
float3 refractColor =
SAMPLE_TEXTURE2D_X(_CameraOpaqueTexture,sampler_CameraOpaqueTexture,
refractCoord).xyz * absorptionFactor;
```

PBF: Color Absorption

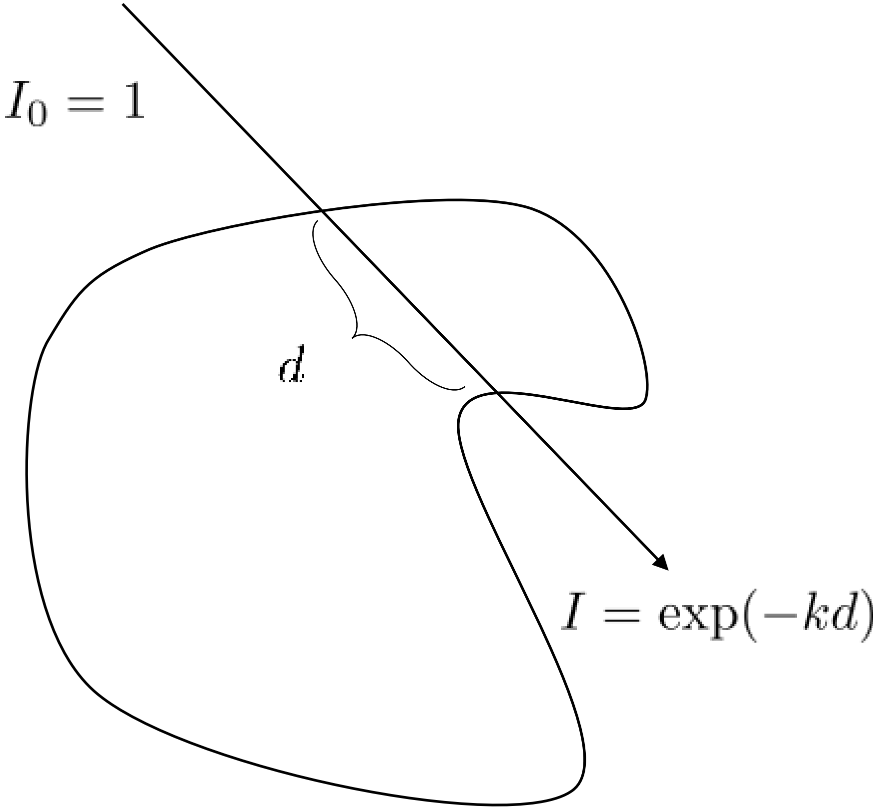
- **Beer-Lambert's law**
- **Light decays exponentially with distance:**

$$I = I_0 \exp(-kd)$$

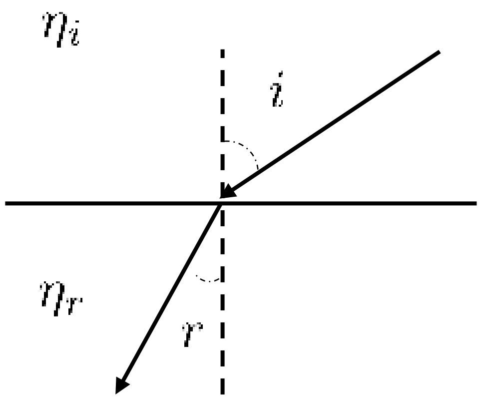
$$* : k \ the \ absorption \ factor, \ d \ the \ thickness$$

- **For each color component can use different k**

$$k = (k_r, k_g, k_b)$$



PBF: Transmit



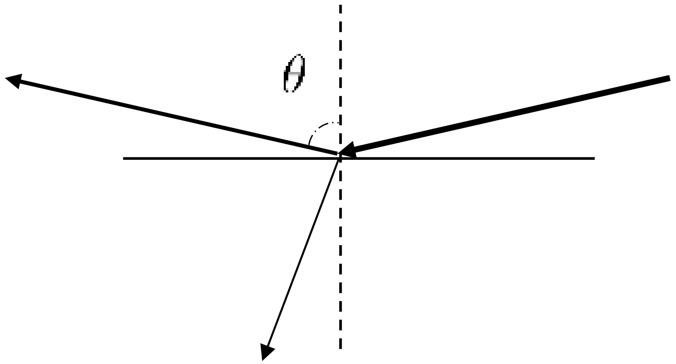
- **Transmit = Refract + Scatter**
- **Refract: Snell's law**

$$\frac{\eta_i}{\eta_r} = \frac{\sin\theta_r}{\sin\theta_i} = \eta$$

- **Eta the reciprocal of IOR, for water is around 1/1.33**
- **Add a factor to control scatter**

$$\text{Transmit} = lerp(refractColor, scatterColor, \text{Turbidity})$$

PBF: Fresnel



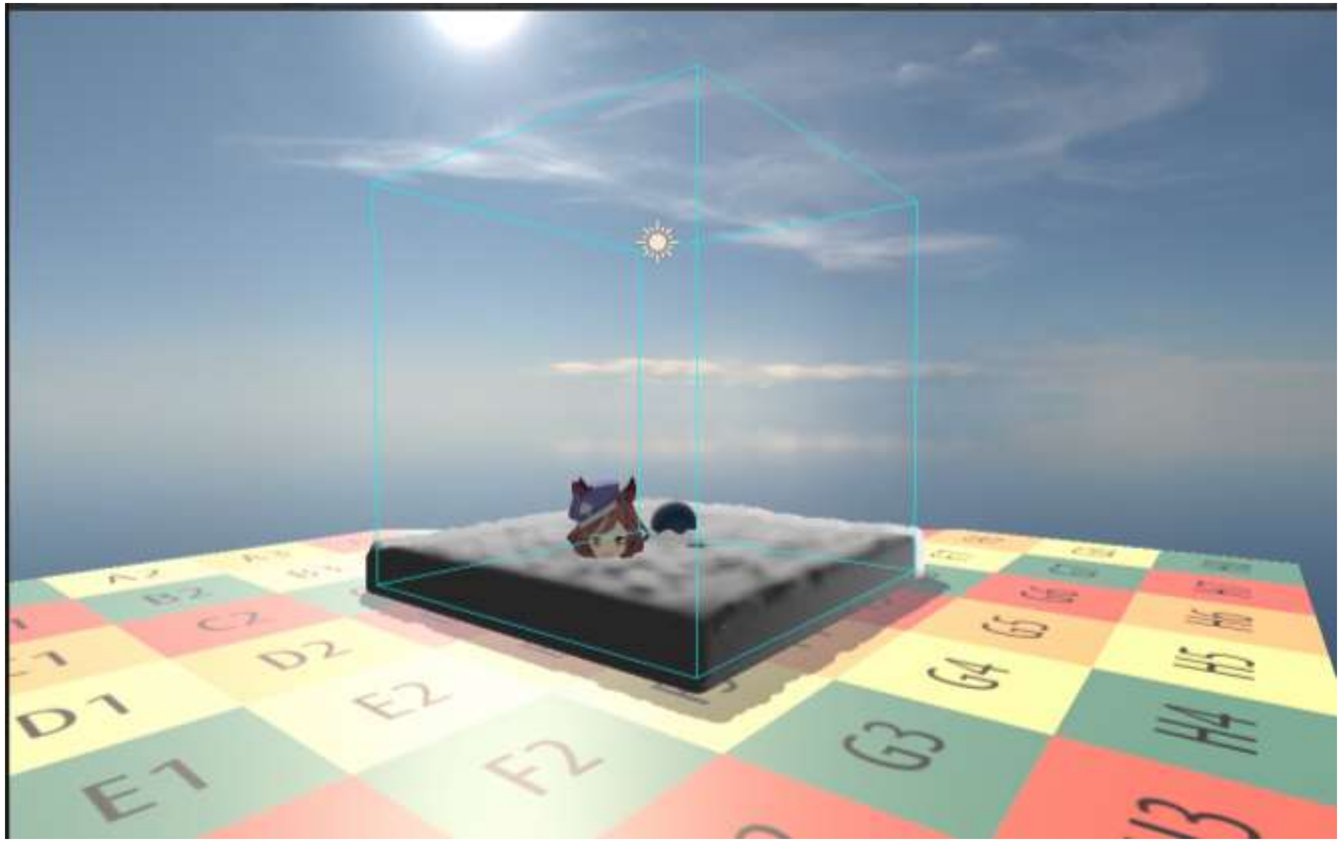
- More light reflected close to grazing angles
- Schlick's approximation

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

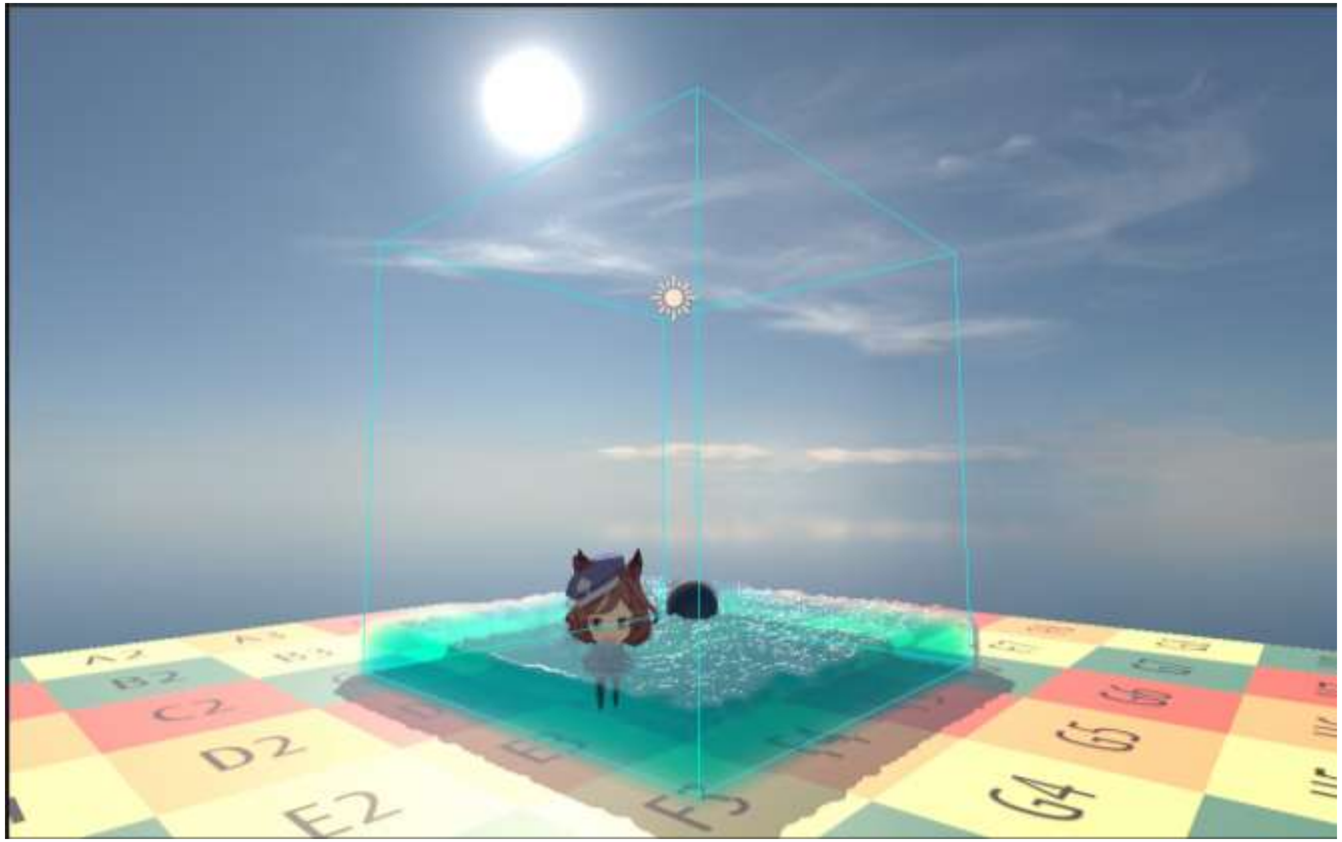
- F0: the albedo at normal incidence, for water is around 0.02
- Compose reflect and transmit with Fresnel

$$\text{Luminance} = lerp(\text{Transmit}, \text{Reflect}, F)$$

PBF: Fluid Fresnel



PBF: Fluid Composite



Expectation

- Support 100,000 particles
- Realtime: At least run at 30 fps
- Able to interact with user and scene