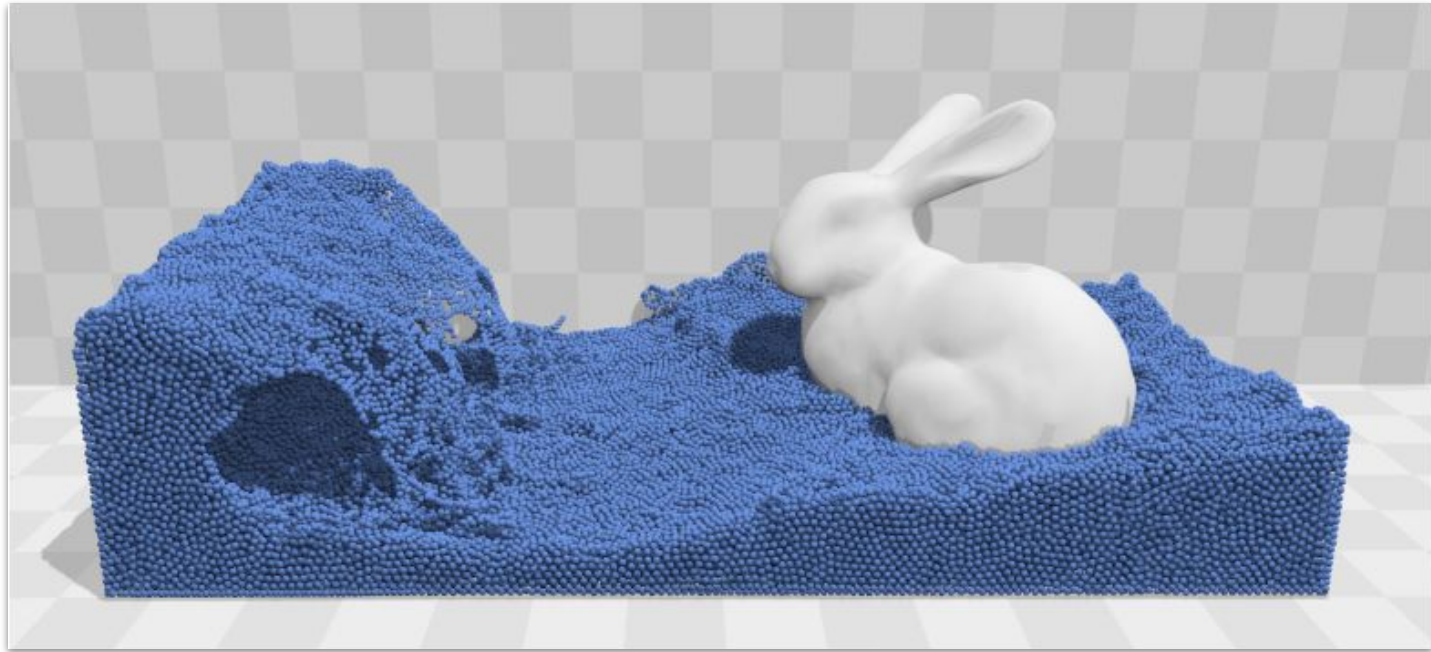
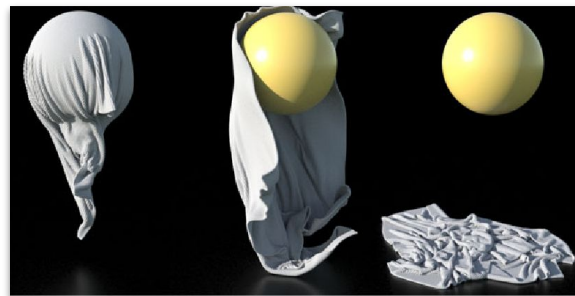
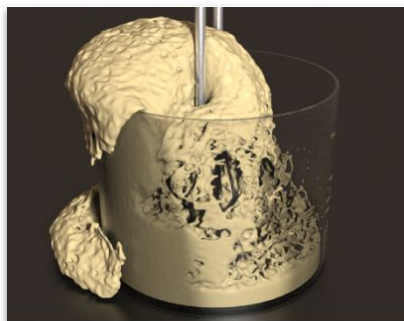
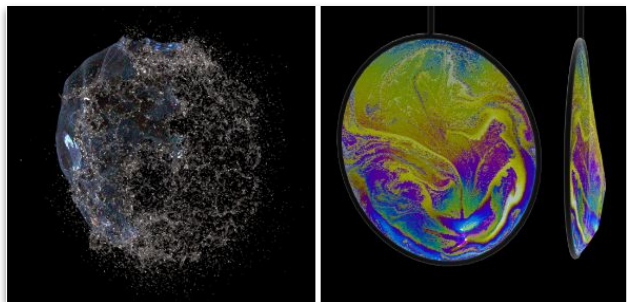
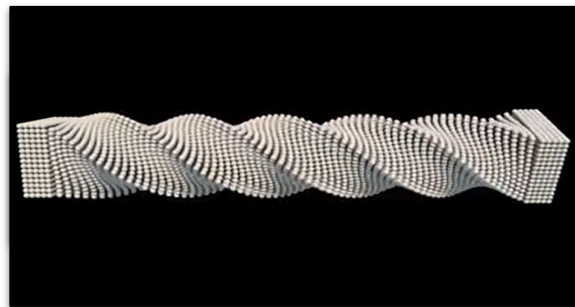
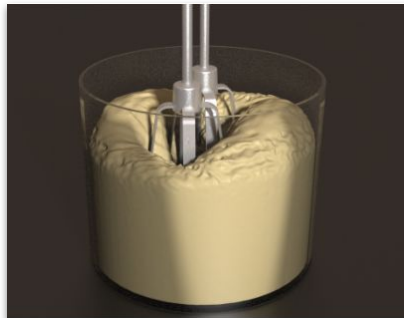
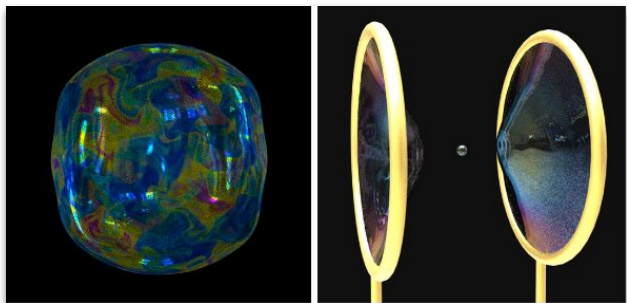


Position Based Fluid (2013)

Miles Macklin, Matthias Müller



SPH Fluid Simulation



Thin-Film SPH Fluid (2021)

**An Implicit Viscosity Formulation
for SPH Fluids (2015)**

**An Implicit SPH Formulation for
Incompressible Linearly Elastic Solids
(2019)**

Fluid Position Based Dynamics

Particle/Vertex $i \in [1, \dots, N]$

position : $\mathbf{x}_i$

velocity : $\mathbf{v}_i$

mass : m_i

density : ρ_0

Constraints $j \in [1, \dots, M]$

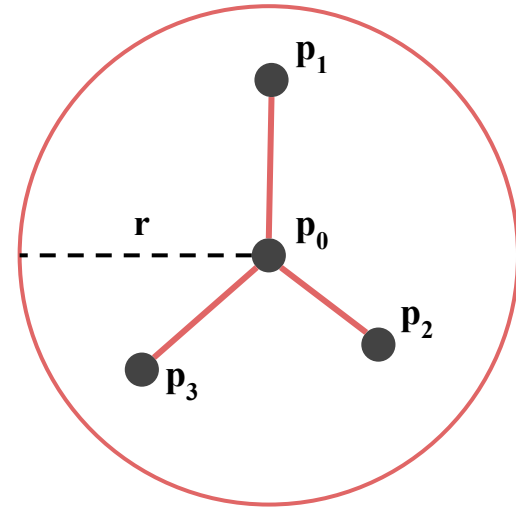
cardinality : n

indices : $\{i_1, \dots, i_{n_j}\}, i_k \in [1, \dots, N]$

fonction : $C_j(\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_{n_j}}) \rightarrow \mathbb{R}$

stiffness : $k \in [0, 1]$

condition : *equality* ($C = 0$) or *inequality* ($C \geq 0$)



$C(p_0, \{p_1, p_2, p_3\})$

Fluid PBD Solver

```
while(True)
  forall particles i do  $v_i \leftarrow v_i + \Delta t w_i f_{\text{ext}}$ 
  forall particles i do  $p_i \leftarrow x_i + \Delta t v_i$ 

  forall particles i do findNeighbours( $p_i$ )

  loop solverIterations times
    forall particles i do
      solveConstraints()
      checkCollisions() & solveCollisions()

    forall particles i do
       $p_i \leftarrow p_i + \Delta p_i$ 

  forall particles i
     $v_i \leftarrow (p_i - x_i) / \Delta t$ 
    applyVorticity() & applyViscosity()
     $x_i \leftarrow p_i$ 
endfor
```

Explicit Integrations

[Particle Simulation using CUDA](#) (2010)

[Loïs GDL]

Density Constraints

Collisions Constraints

Apply Corrections

Compute Velocities

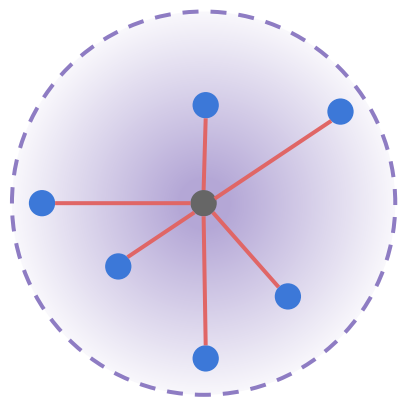
Fluid stuff

Fluid Constraint

Fluid constraint (clamped)

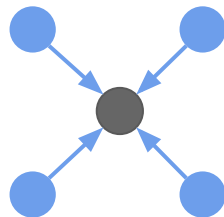
$$C(\mathbf{x}) = \frac{\rho_i}{\rho_0} - 1 \leq 0$$

$$\rho_i = \sum_j m_j W(\mathbf{p}_i - \mathbf{p}_j, \mathbf{h})$$

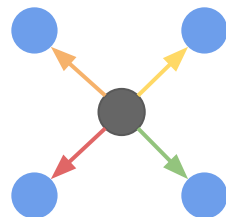


$$W \in [0,1]$$

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = i \\ -\nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = j \end{cases}$$



$$\mathbf{k} = \mathbf{i}$$



$$\mathbf{k} = \mathbf{j}$$

$$\lambda_i = -\frac{C_i(\mathbf{p}_1, \dots, \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2}$$

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

Some fixes

Tensile Instability

$$s_{corr} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta \mathbf{q}, h)} \right)^n$$

$$\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

Add artificial pressure causing surface tension-like

Avoid lonely particles to blow up

Vorticity Confinement and Viscosity

Vorticity

$$\omega_i = \nabla \times \mathbf{v} = \sum_j \mathbf{v}_{ij} \times \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_j, h)$$

$$\mathbf{f}_i^{vorticity} = \varepsilon (\mathbf{N} \times \omega_i)$$

Viscosity

$$\mathbf{v}_i^{new} = \mathbf{v}_i + c \sum_j \mathbf{v}_{ij} \cdot \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

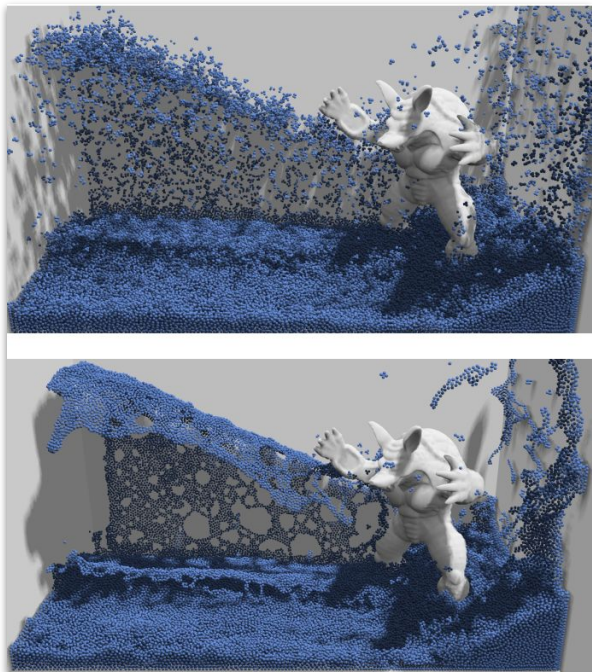
Add turbulent motion

Increase motion coherence

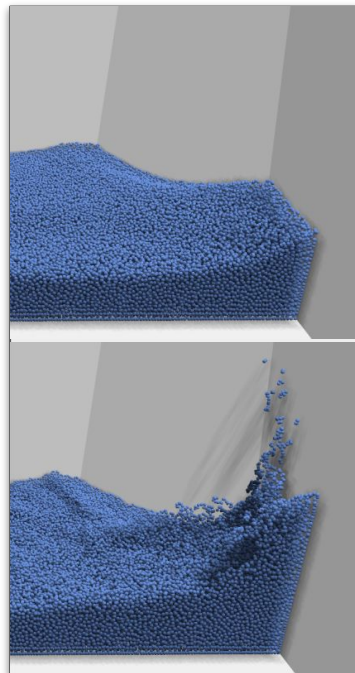
Good looking stuff

Results

Tensile Instability



Vorticity

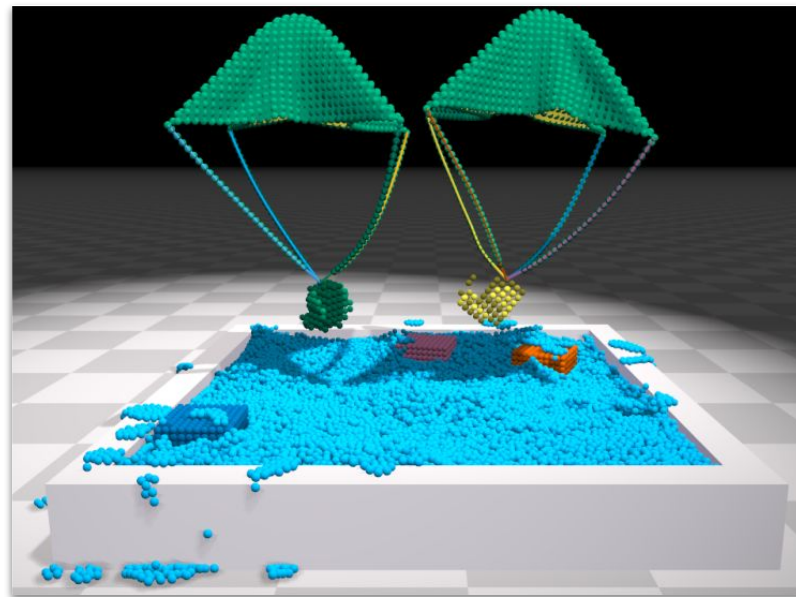
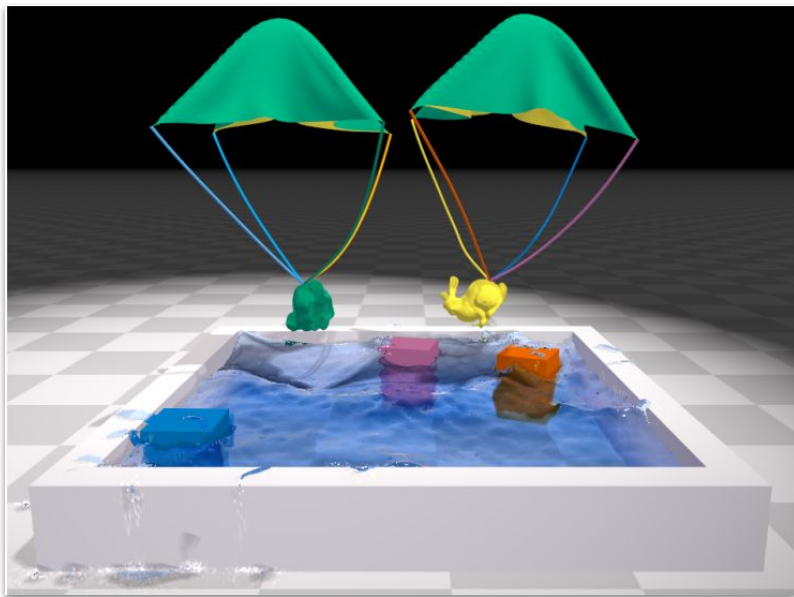


Rendering



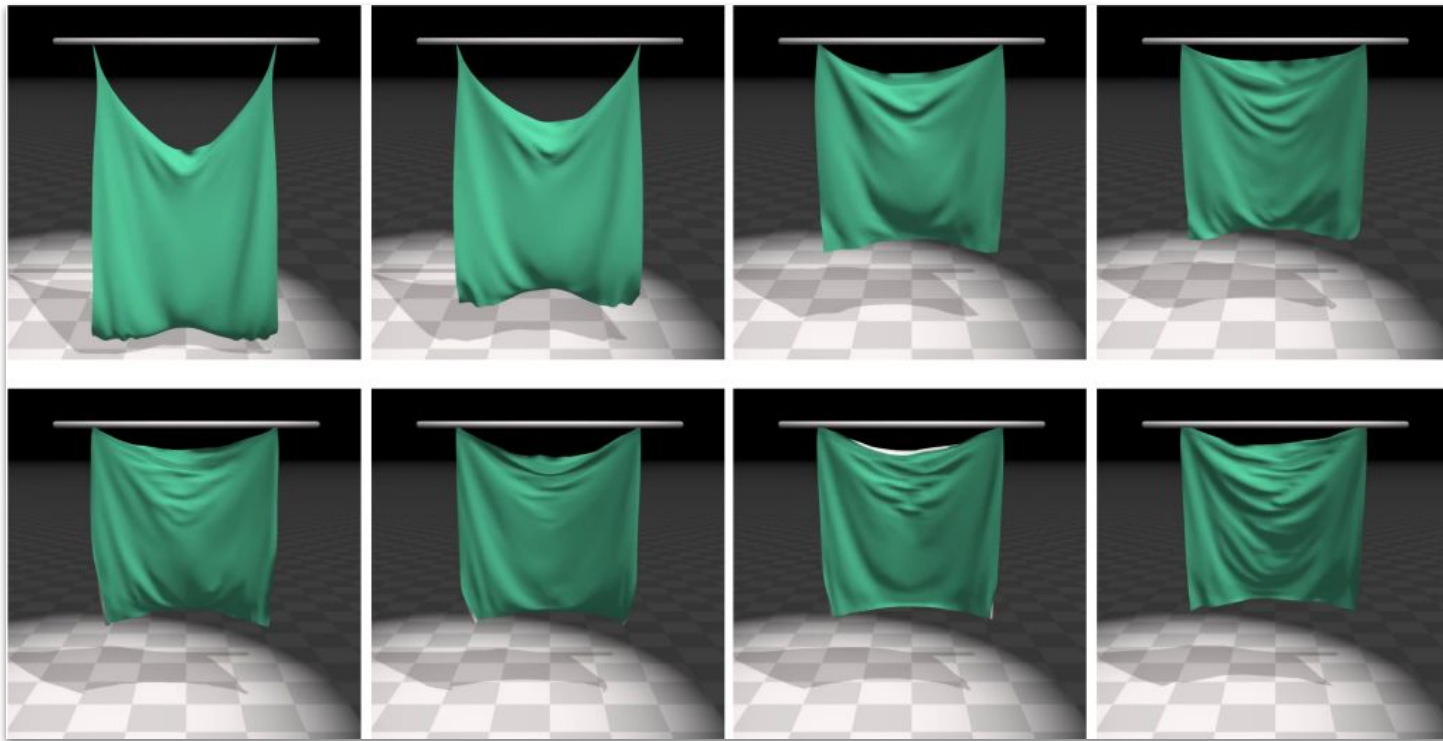
Unified Particle Physics for Real-Time Applications (2014)

Miles Macklin Matthias Müller Nuttapong Chentanez Tae-Yong Kim



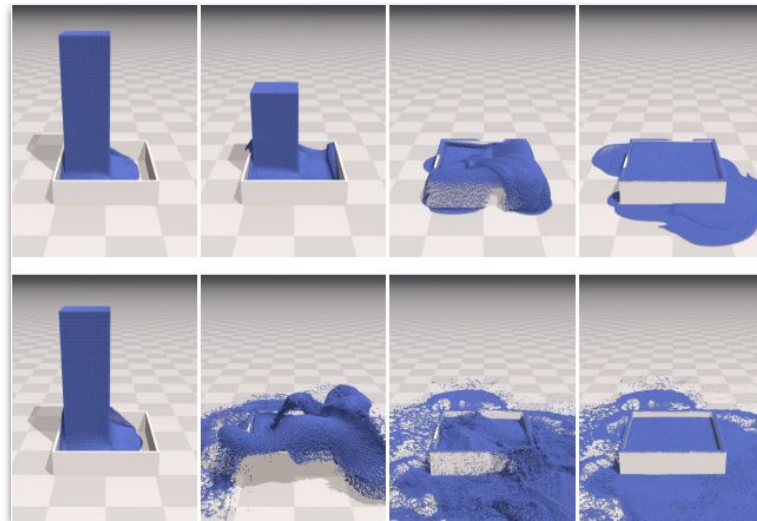
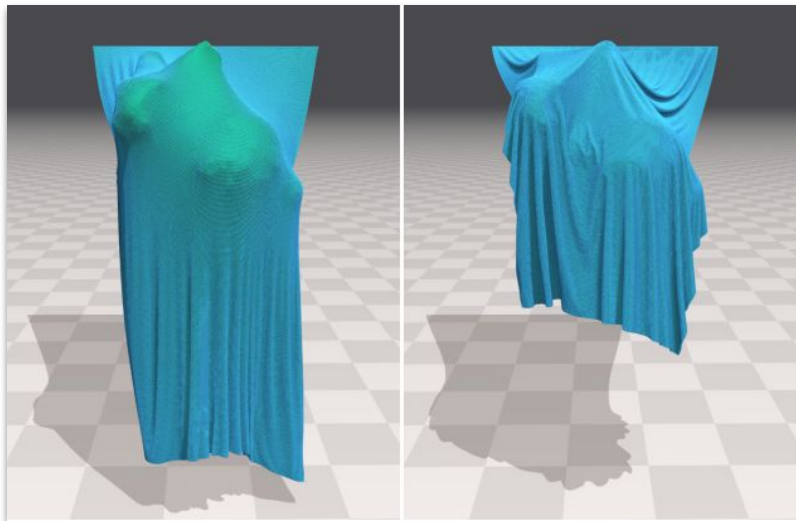
Extended Position Based Dynamics (2016)

Miles Macklin, Matthias Muller, Nuttapong Chentanez



Small Steps in Physics Simulation (2019)

*Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman,
Nuttapong Chentanez, Stefan Jeschke, Matthias Müller*



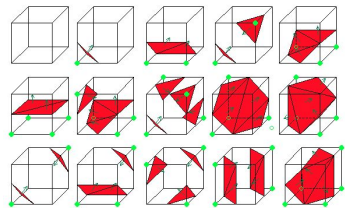
SPH Based Fluid Rendering



SPH Based Fluid Rendering

Mesh Based

Marching Cube



Screen Space Mesh



Blurring/Kernels

Gaussian Blur

Anisotropic Kernel

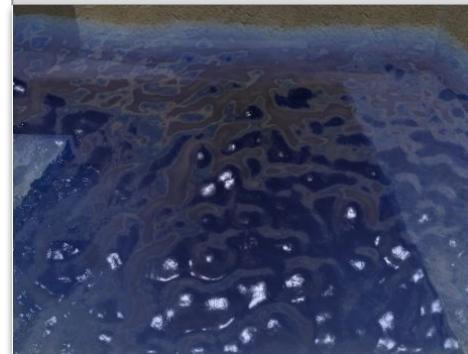
Bilateral Gaussian Filter

1D Filters



Optimisation

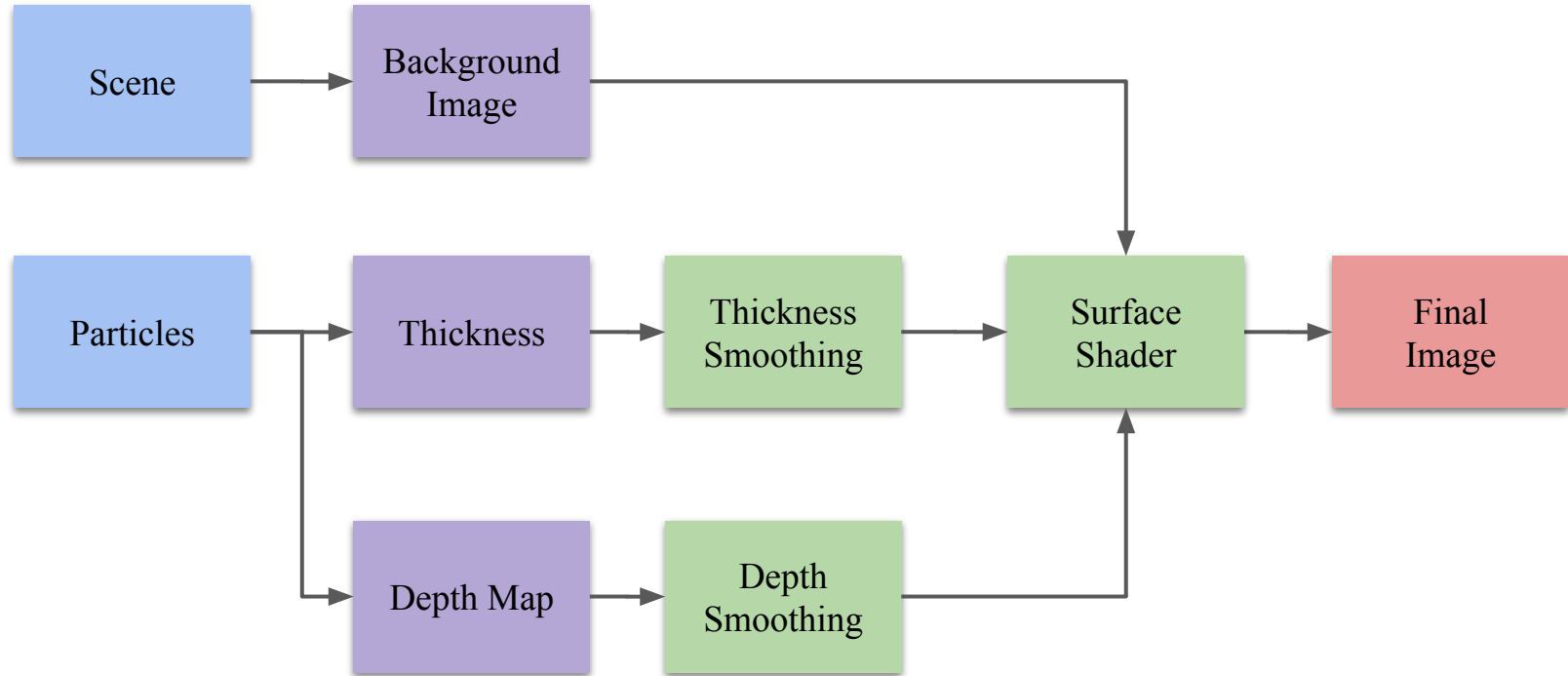
Curvature Flow



Implicit surface

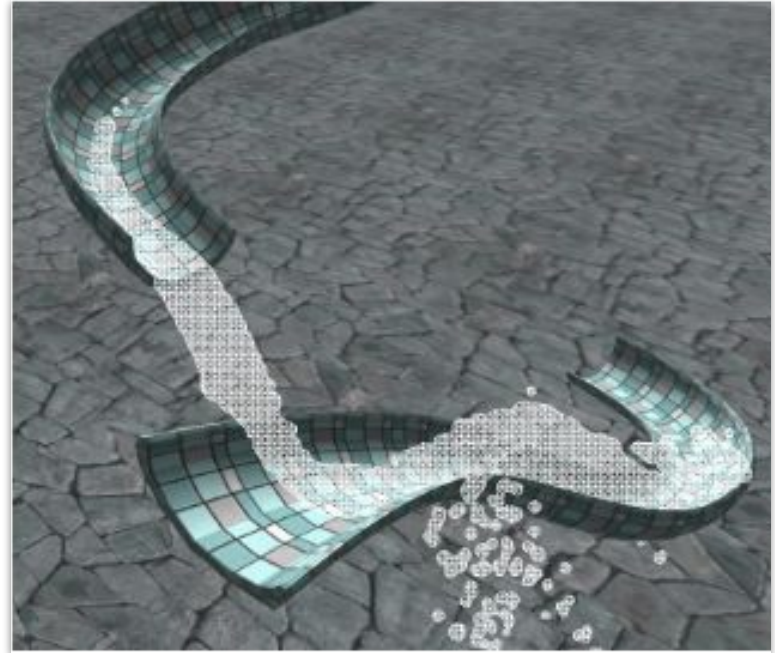
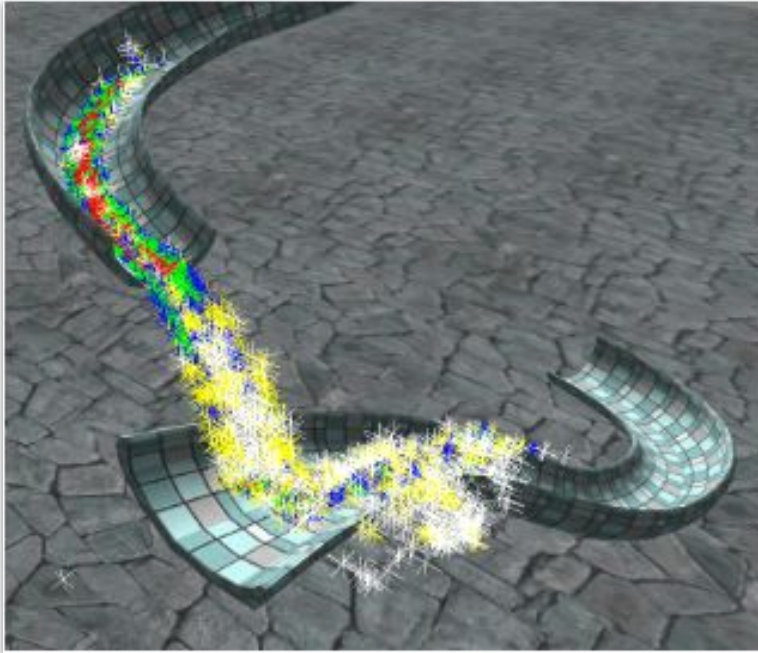
Raytracing

SPH Based Fluid Rendering



Screen Space Meshes (2007)

Matthias Müller, Simon Schirm, Stephan Duthaler



Method

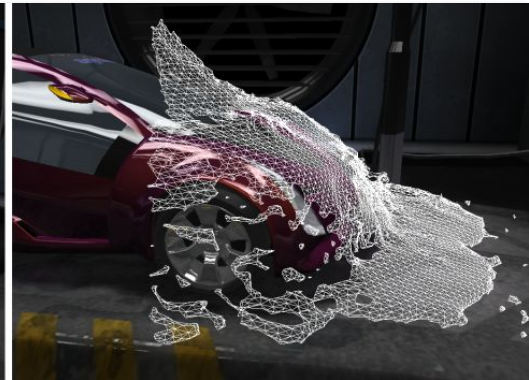
1. Setup regular depth map
2. Find internal and external silhouettes
3. Smooth depth values
4. Generate a 2D triangle mesh using our Marching Squares related procedure
5. Smooth silhouettes
6. Transform mesh back into world space
7. Render the 3D triangle mesh

Square marching
is 2D cube marching

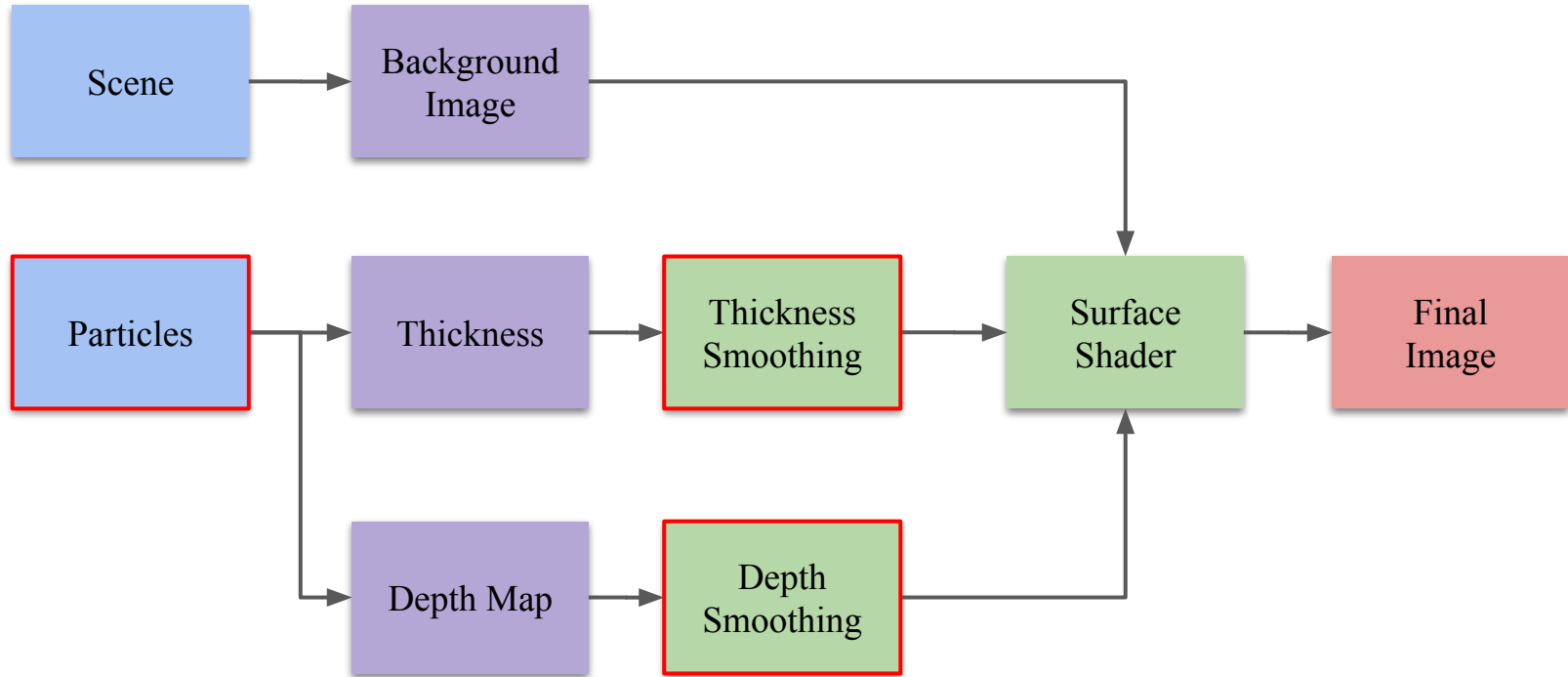
Mesh is bad, bad, bad

No fluid thickness

Real time though



SPH Based Fluid Rendering



Rendering Particle Spheres

Billboards

(Camera oriented quad)

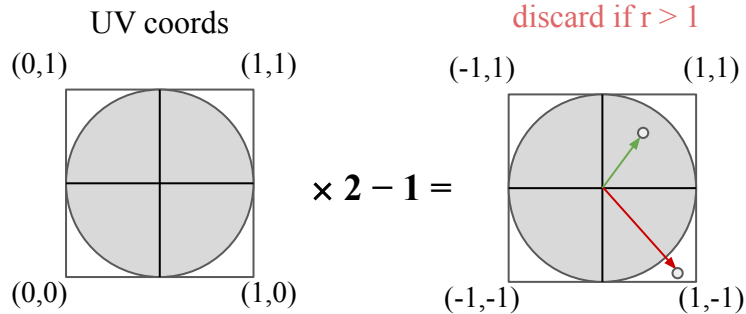
```
in vec2 texCoord;
in vec3 viewPosition; // billboard center position in camera referential

uniform mat3 P; // projection matrix
uniform vec4 color;
uniform float radius;
uniform vec3 lightDir;

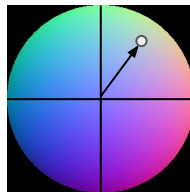
out float f_depth;
out vec4 f_color;

void main() {
    // compute normal
    vec3 n = texCoord * 2.0 - 1.0;
    float r2 = dot(n.xy, n.xy);
    if(r2 > 1.) discard;
    n.z = -sqrt(1. - r2);

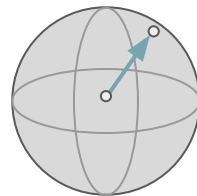
    // compute depth
    vec4 pixelPos = vec4(viewPosition + n * radius, 1.0);
    vec4 projectedPosition = P * pixelPos;
    f_depth = projectedPosition.z / projectedPosition.w; // normalize
}
```



Point Normal
 $z = -\sqrt{1. - r2}$



Point position (view)



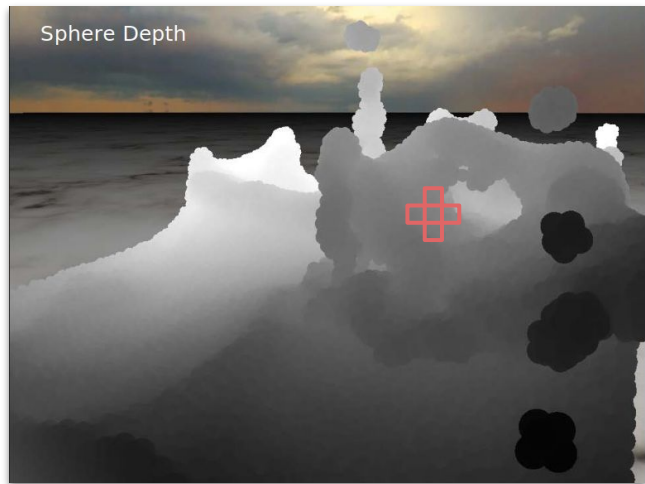
Apply projections



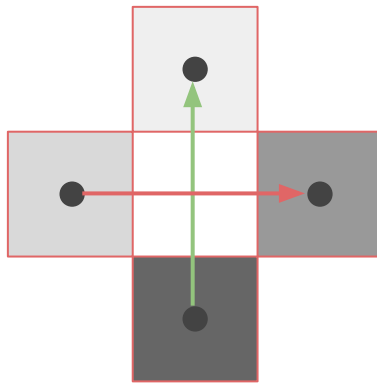
Compute frag depth

Normals with Depth Map

Depth map sampling

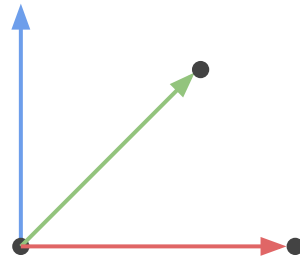


Points
(projection referential)



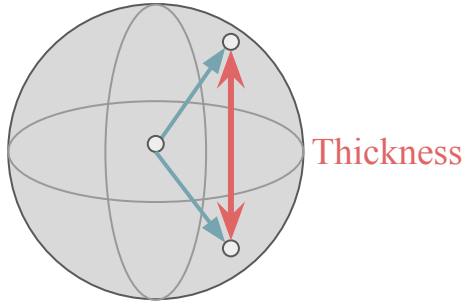
Compute normal
(view referential)

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



$$\begin{aligned} \mathbf{n}(x, y) &= \frac{\partial \mathbf{P}}{\partial x} \times \frac{\partial \mathbf{P}}{\partial y} \\ &= \begin{pmatrix} C_x z + W_x \frac{\partial z}{\partial x} \\ W_y \frac{\partial z}{\partial x} \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} W_x \frac{\partial z}{\partial y} \\ C_y z + W_y \frac{\partial z}{\partial y} \\ \frac{\partial z}{\partial y} \end{pmatrix} \approx \begin{pmatrix} C_x z \\ 0 \\ \frac{\partial z}{\partial x} \end{pmatrix} \times \begin{pmatrix} 0 \\ C_y z \\ \frac{\partial z}{\partial y} \end{pmatrix} = \begin{pmatrix} -C_y \frac{\partial z}{\partial x} \\ -C_x \frac{\partial z}{\partial y} \\ C_x C_y z \end{pmatrix} z \end{aligned}$$

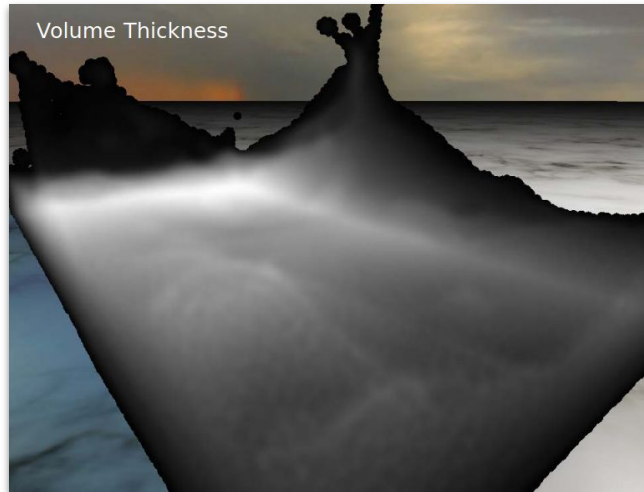
Fluid Thickness



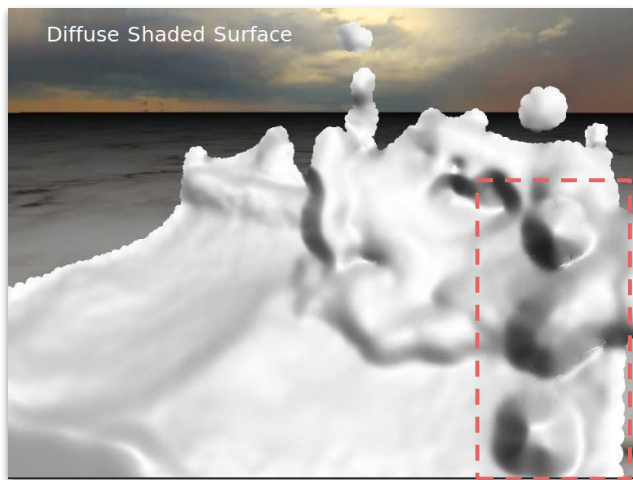
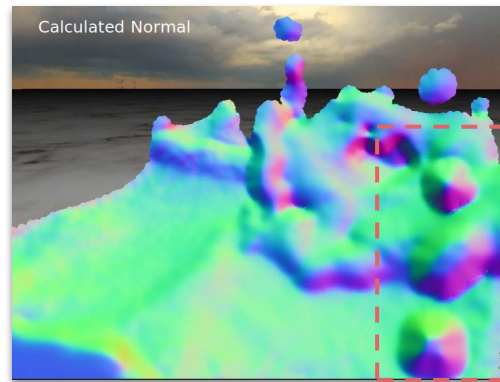
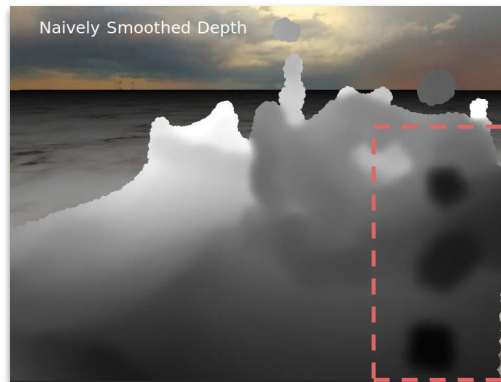
Very close to depth rendering

with additive color rendering (no depth test)

true if no interpenetration (good enough)



Depth Map Blurring



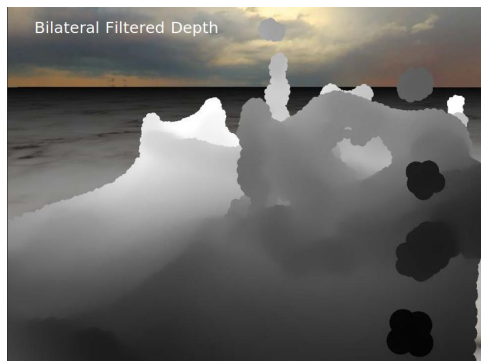
Naive Blur does not preserve edges in depth map !

Bad normals, Bad surfaces construction

Damn, that's ugly (but fast ...)

Bilateral Filter

Bilateral Blur



2D kernel



Preserve edges

Slow ?

Noisy surface

1D kernel (approximation)



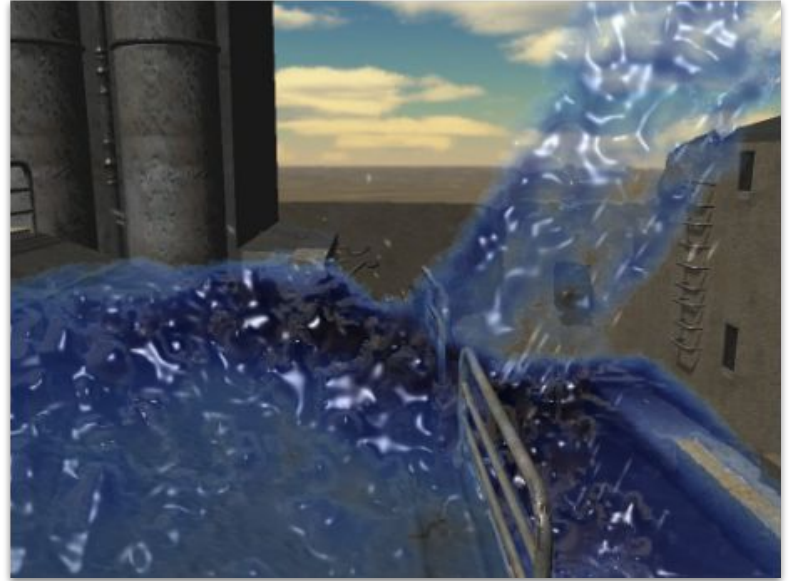
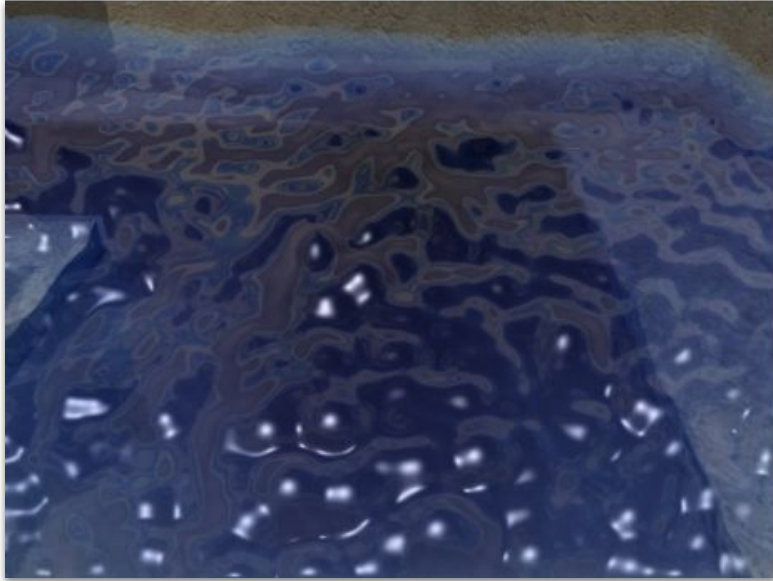
Preserve edges (kinda)

Fast

Look terribly bad
(ok, not that much with shaders)

Screen Space Fluid Rendering with Curvature Flow (2009)

Wladimir J. van der Laan, Simon Green, Miguel Sainz



Curvature Minimisation

$$\boxed{\frac{\partial z}{\partial t} = H} \quad 2H = \frac{\partial \hat{n}_x}{\partial x} + \frac{\partial \hat{n}_y}{\partial y} = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}}$$

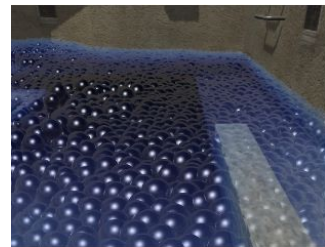
$$D = C_y^2 \left(\frac{\partial z}{\partial x} \right)^2 + C_x^2 \left(\frac{\partial z}{\partial y} \right)^2 + C_x^2 C_y^2 z^2$$

$$C_x = \frac{2}{V_x F_x}, C_y = \frac{2}{V_y F_y}$$

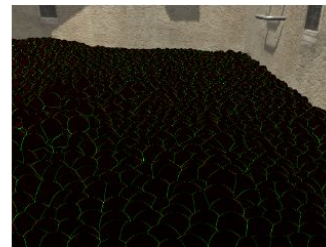
F_x, F_y : focal length V_x, V_y : viewport dimensions

Derivatives computed with finite differences

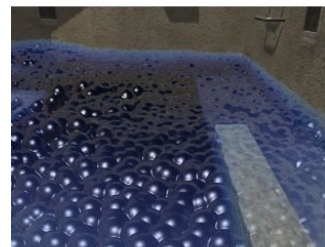
Explicit Integration Scheme
dt has to be small !



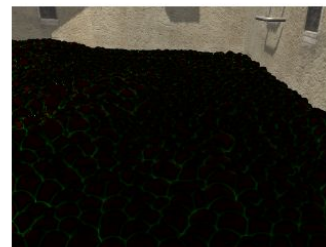
(a) 0 iterations, image



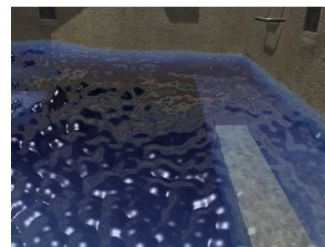
(b) 0 iterations, curvature



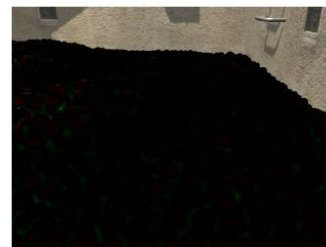
(c) 40 iterations, image



(d) 40 iterations, curvature



(e) 60 iterations, image



(f) 60 iterations, curvature

Noise and Foam

Billboard noise rendering

$$I(x, y) = \text{noise}(x, y) * e^{-x^2 - y^2 - (\mathbf{p}_z(x, y) - d(x, y))^2}$$

Exponential fall-off based on the depth (below surface)

Particles contribute less as they submerge

$\mathbf{p}$: view-space position d : depth $x, y \in [-1, 1] : uv * 2 - 1$

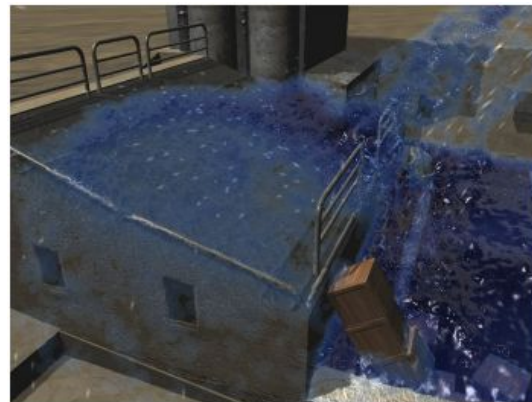
Summed on the screen to get a noise value at every pixel

Add some surface details (can depend on velocity)

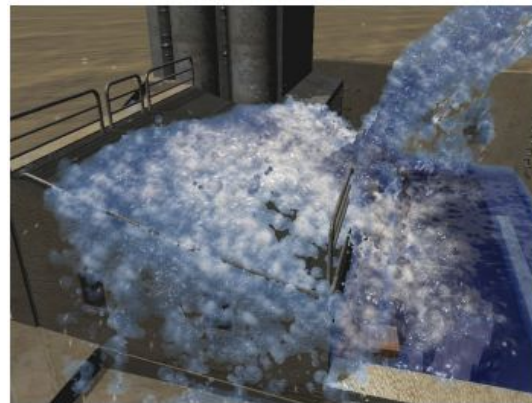
Foam

(just a threshold)

$$|\mathbf{v}_i(t) - \mathbf{v}_i(t - 1)| > \tau$$



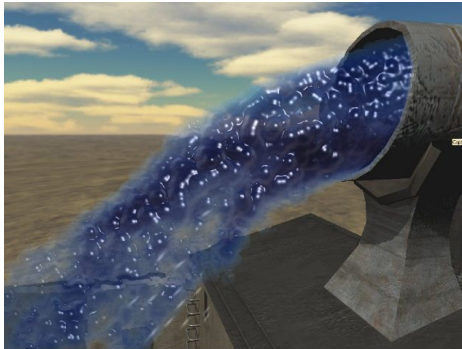
(a) Without foam



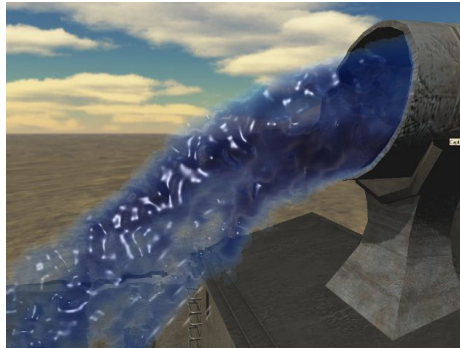
(b) With foam

Result

Gaussian Smoothing



Curvature flow

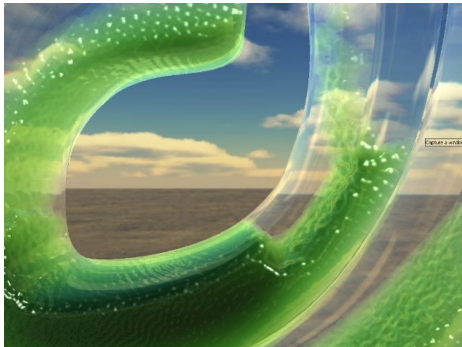


Preserve edges

Good smoothing

Fast^{nt}

(small kernel but need lot of iterations)

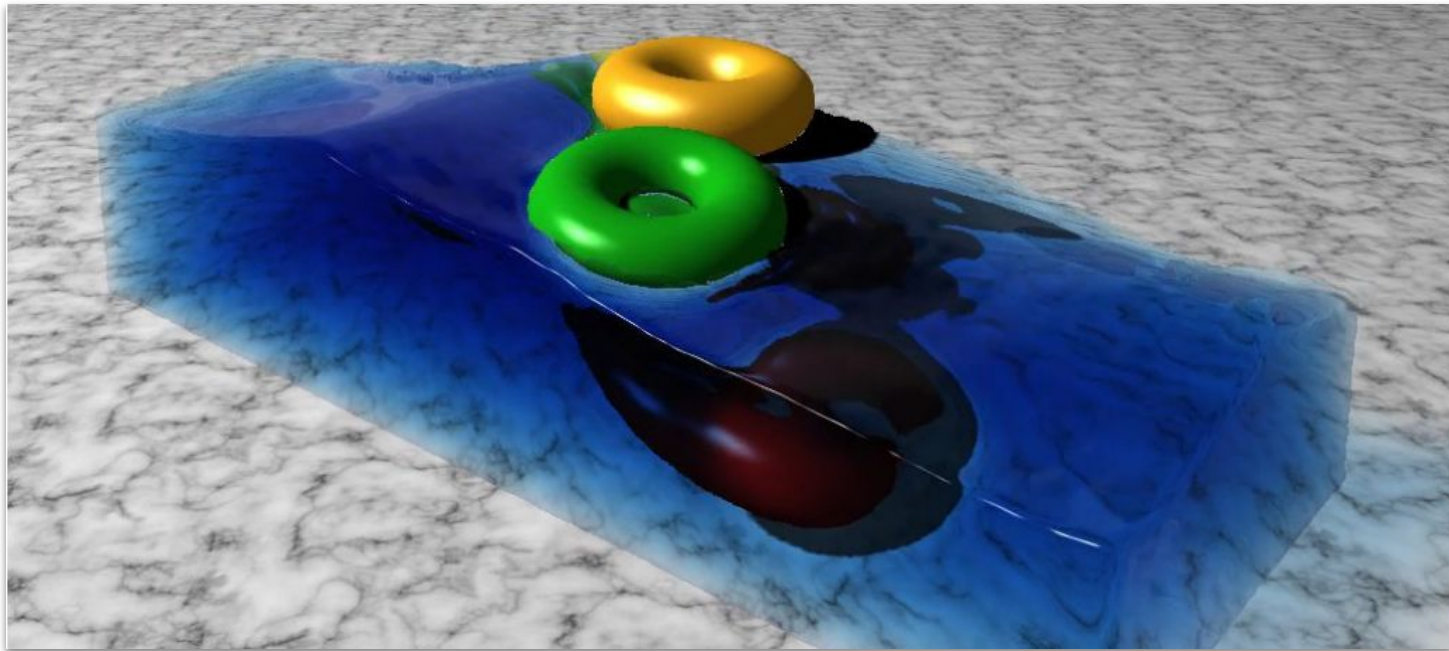


Performance

Corridor dataset	Frame (ms)
Bilateral Gaussian smoothing	18.1
Quarter res., 15 iterations	17.5
Half res., 40 iterations	19.6
Full res., 100 iterations	50.0
Foam+noise, quarter res, 15 iterations	30.0
Logo dataset	
Bilateral Gaussian smoothing	22.7
Quarter res., 15 iterations	23.3
Half res., 40 iterations	28.6
Full res., 100 iterations	50.0

A Narrow-Range Filter for Screen-Space Fluid Rendering (2018)

Nghia Truong, Cem Yuksel



Narrow-Range Filter ?

It's a Gaussian Filter

Gaussian filter ? Really ?

With a threshold on depth !



Was already a thing before ... so what ?
(back in 2007)

Threshold not good because its constant

What about a dynamic range ?



Gaussian kernel with threshold

Kernel value :
$$\omega_{ij} = \begin{cases} 0, & \text{if } z_j > z_i + \delta \\ G(\mathbf{p}_i, \mathbf{p}_j, \sigma_i), & \text{otherwise,} \end{cases}$$

$$G(\mathbf{p}_i, \mathbf{p}_j, \sigma_i) = e^{-|\mathbf{p}_j - \mathbf{p}_i|^2 / 2\sigma_i^2} \quad \sigma_i = \left\lceil \frac{H \sigma}{2 |z_i| \tan(\alpha/2)} \right\rceil$$

H : vertical screen resolution,

p : pixel coordinates

σ : world space filter size

δ : threshold value

α : camera's field of view angle

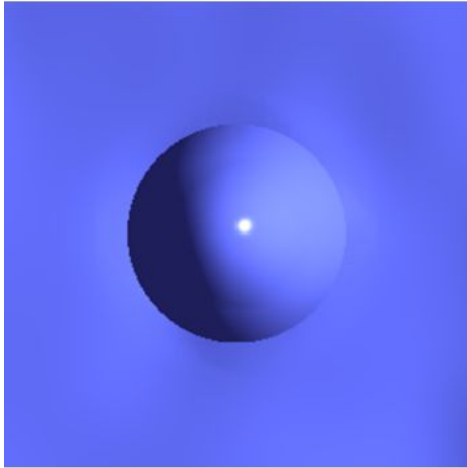
z : depth value

kernel scene size = $3\sigma_i$
(nb sample is constant)

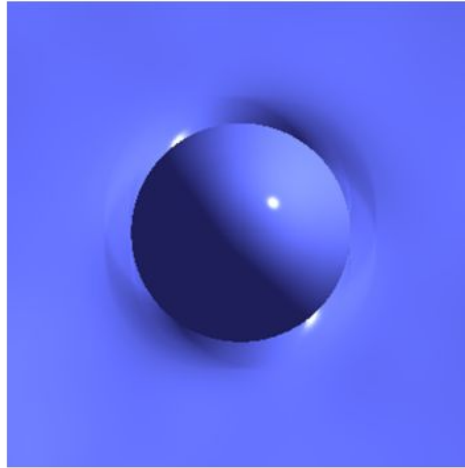
Create a different kernel for each fragment

There is still artifacts on edges

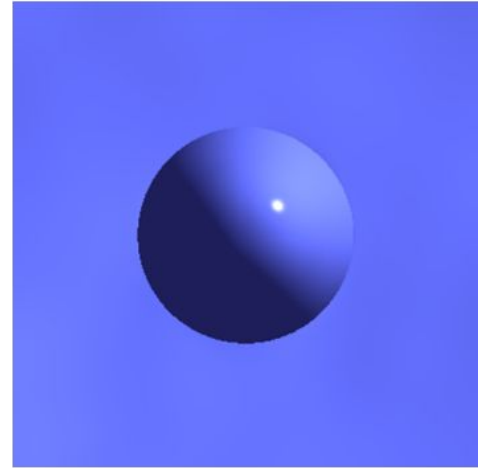
Threshold induces error on edges



(a) Perpendicular
Surface



(b) Inclined
Surface

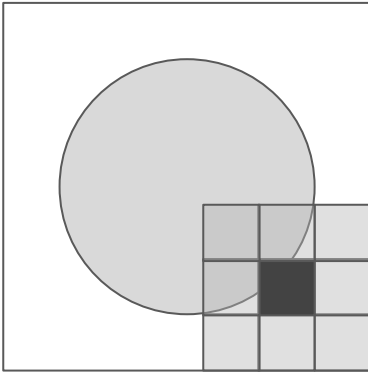


(c) Bias
Correction

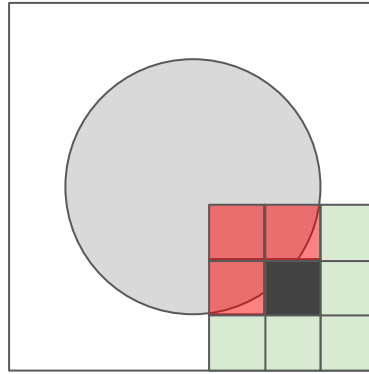
Threshold creates asymmetrical kernels

Bias correction

Kernel 3x3



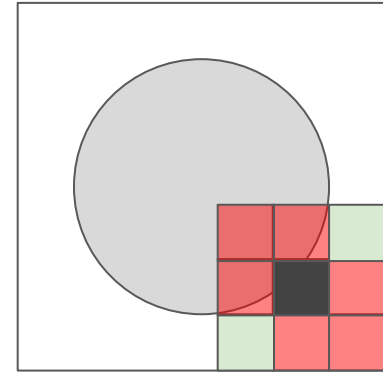
Threshold



Remove if

$$|z_j - z_i| < \delta$$

Bias Correction

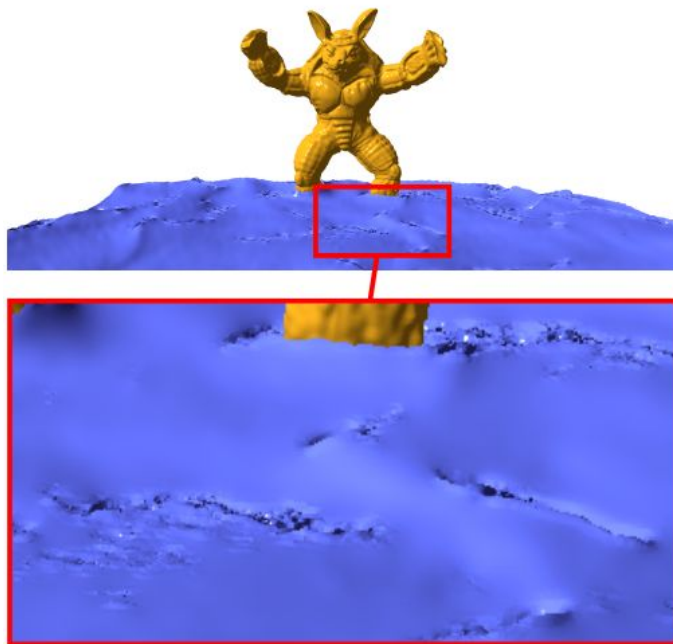


if j removed

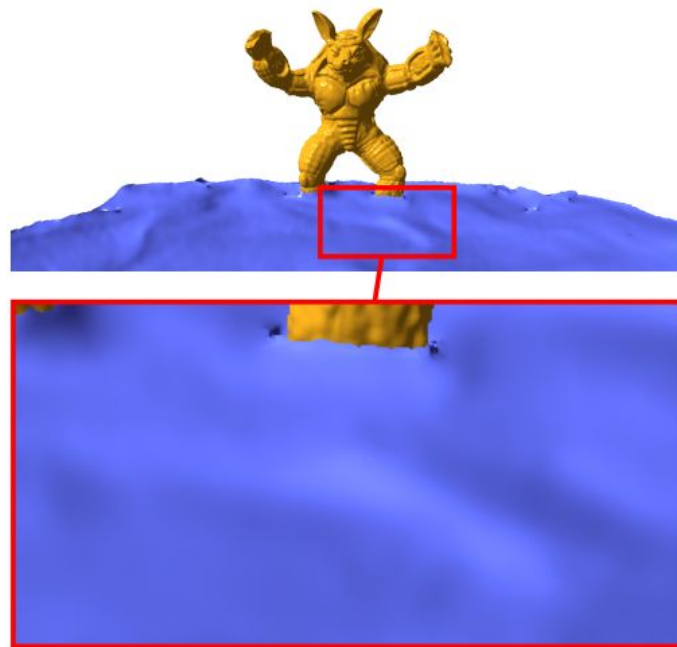
remove $k = 2i - j$

Constant threshold still create visual artifacts

Dynamic Range



(a) Static Range (δ)



(b) Dynamic Range ($\delta_{low,high}$)

Dynamic Range

$$z_i + \delta_{high} \geq z_j \geq z_i - \delta_{low}$$

$$\delta_{low} \leftarrow \max(\delta_{low}, z_i - z_j + \delta)$$

$$\delta_{high} \leftarrow \max(\delta_{high}, z_j - z_i + \delta)$$

δ_{low} and $\delta_{high} = \delta$ at start

adjusted when compute kernel

Clamping function

Final Depth

$$z'_i = \frac{\sum_j \omega_{ij} f(z_i, z_j)}{\sum_j \omega_{ij}}$$

Clamping Function

$$f(z_i, z_j) = \begin{cases} z_j, & \text{if } z_j \geq z_i - \delta \\ z_i - \mu, & \text{otherwise,} \end{cases}$$

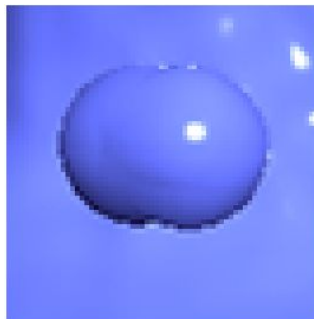
1D and clean up

Gaussian filter isn't separable

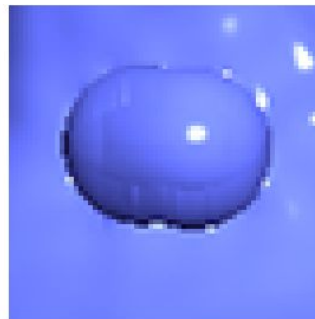
1D cause artifacts

Must add clean up path

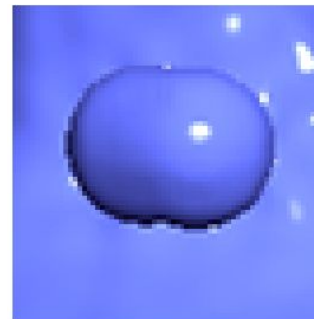
Better perf
Not that bad



(a) 2D Filter
Only

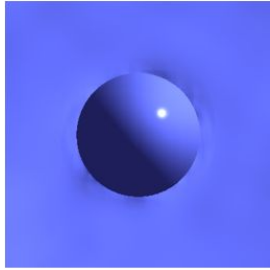


(b) 1D Filters
Only

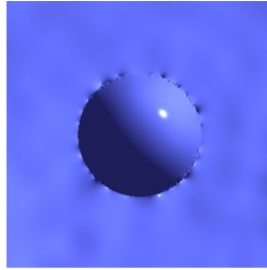


(c) 1D Filters
+ Clean-up

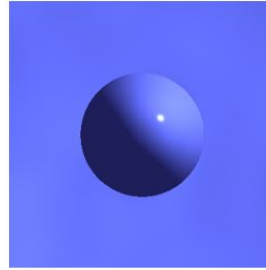
Results



Plane
Fitting



Screen-Space
Curvature Flow



**Narrow-Range
Filter**

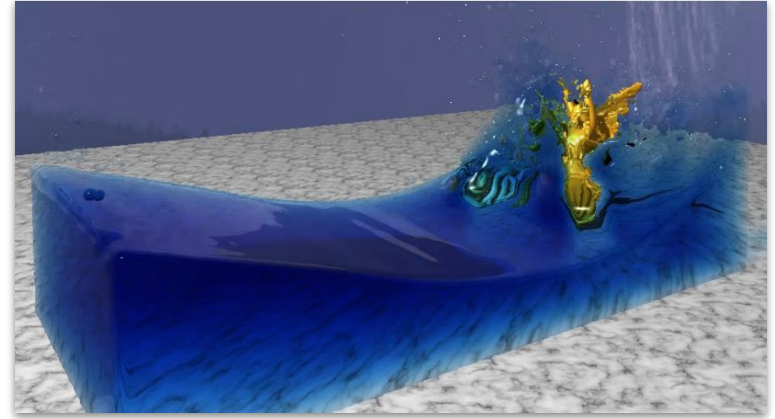
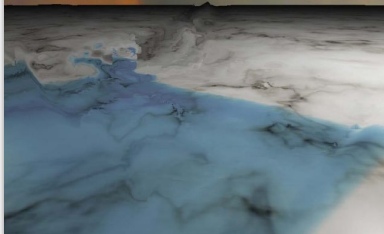


Table 1: Performance Results

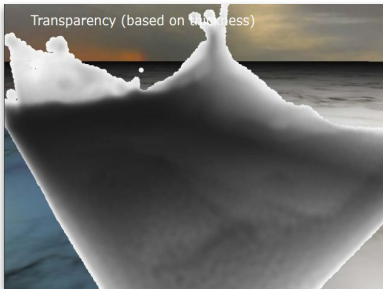
	Figure 1 Armadillo 368K	Figure 1 Bunny 333K	Figure 1 Emitter 350K	Figure 1 Drop 65K	Figure 7 Lucy 135K	Figure 7 Tori 135K	Figure 7 Balls 180K
Number of Particles							
Bilateral Gaussian Filter	0.40 ms	0.46 ms	1.03 ms	0.68 ms	0.48 ms	0.38 ms	0.62 ms
Plane Fitting	0.70 ms	0.90 ms	1.95 ms	1.51 ms	0.93 ms	0.72 ms	1.32 ms
Screen-Space Curvature Flow	5.02 ms	5.50 ms	6.05 ms	5.30 ms	4.34 ms	4.82 ms	5.80 ms
Our Narrow-Range Filter 2D	0.53 ms	0.68 ms	1.58 ms	1.23 ms	0.72 ms	0.53 ms	1.04 ms
Our Narrow-Range Filter 1D	0.51 ms	0.58 ms	1.25 ms	0.62 ms	0.62 ms	0.50 ms	0.67 ms
Our Narrow-Range Filter 1D & Clean-up	0.46 ms	0.52 ms	1.21 ms	0.54 ms	0.56 ms	0.46 ms	0.58 ms

Result in GDC 2010

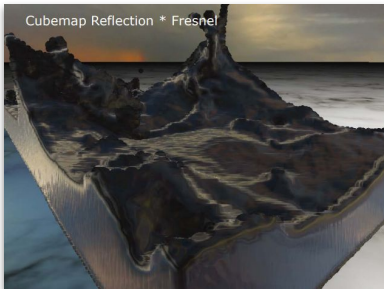
Background Image Refracted in 2D
`tex2D(bgSampler, texcoord+N.xy*thickness)`



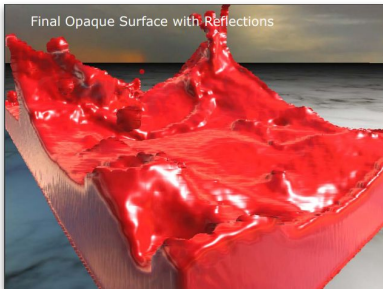
Transparency (based on thickness)



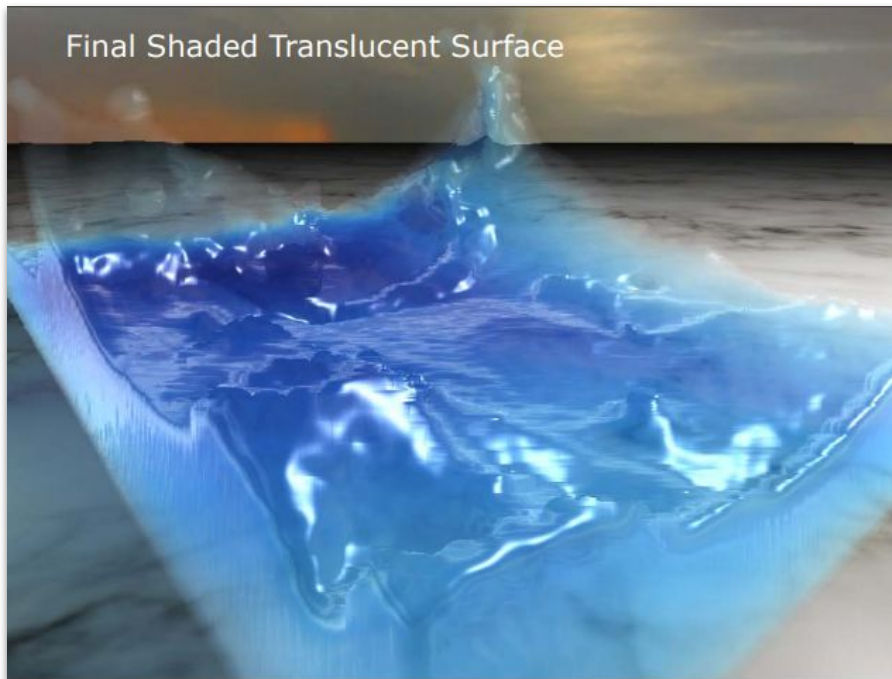
Cubemap Reflection * Fresnel



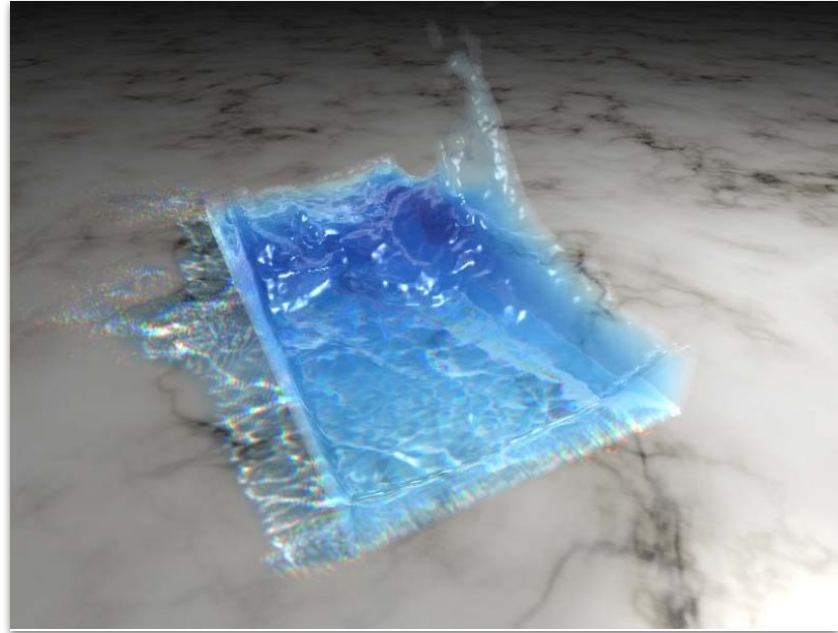
Final Opaque Surface with Reflections



Final Shaded Translucent Surface

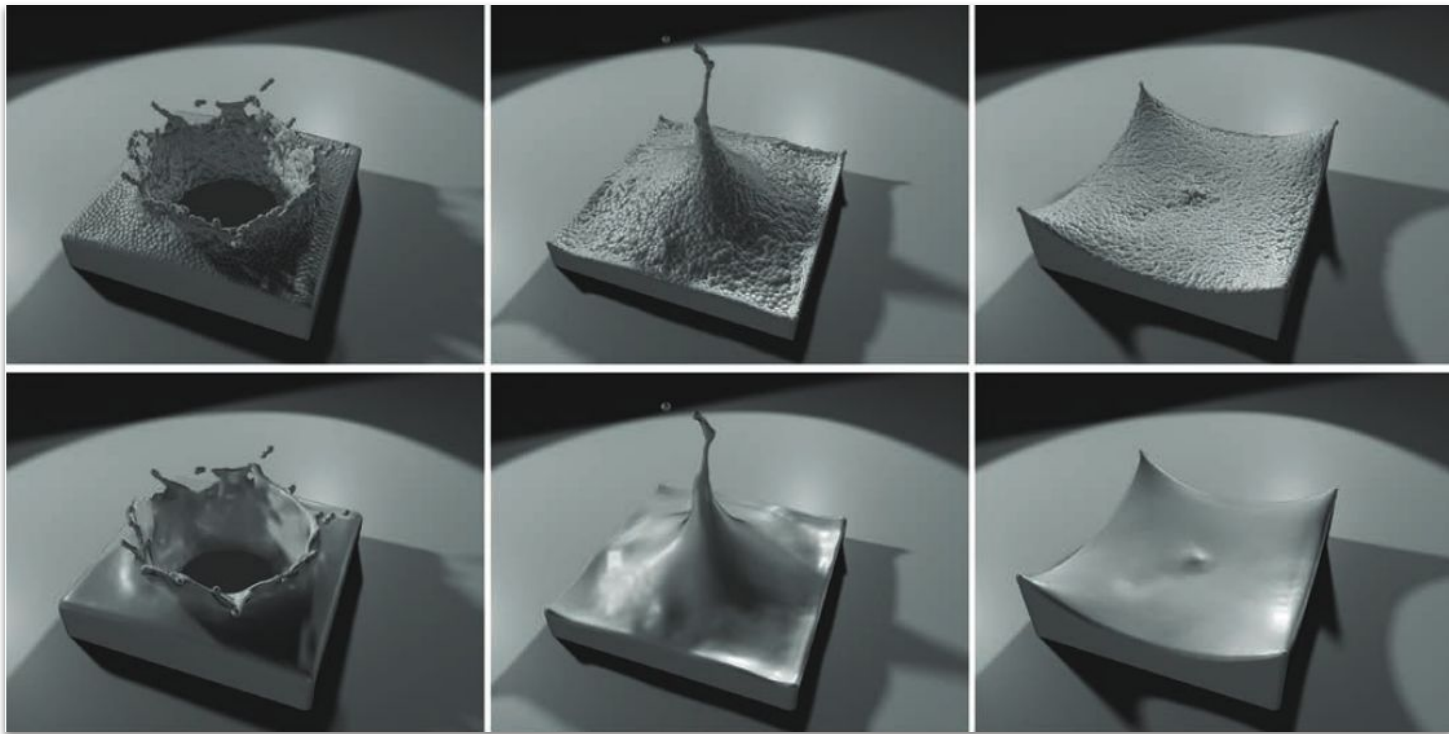


Result in GDC 2010



Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels (2013)

Jihun Yu, Greg Turk



Bibliography

[\[2019\] A Narrow-Range Filter for Screen-Space Fluid Rendering](#)

[\[2016\] Extended Position Based Dynamics](#)

[\[2016\] Small Steps in physics Simulation](#)

[\[2014\] Unified Particle Physics for Real-Time Applications](#)

[\[2013\] Position Based Fluids](#)

[\[2013\] Reconstructing surfaces of particle-based fluids using anisotropic kernels](#)

[\[2010\] GDC : Screen Space Fluid Rendering for Games \(NVidia\)](#)

[\[2009\] Screen Space Fluid Rendering with Curvature Flow](#)

[\[2007\] Screen Space Mesh](#)