Game Developers Conference® | **March 9-13, 2010** | Moscone Center | San Francisco, CA

GDC 10

Learn. Network. Inspire.

www.GDConf.com

Game Developers Conference®
March 9-13, 2010
Moscone Center
San Francisco, CA
www.GDConf.com

Screen Space Fluid Rendering for Games

Simon Green, NVIDIA

Overview

- Introduction
- Fluid Simulation for Games
- Screen Space Fluid Rendering
- Demo

Introduction

- DirectX 11 and DirectCompute enable physics effects to be computed and rendered directly on the GPU
- DirectCompute allows flexible general purpose computation on the GPU
    - sorting, searching
    - spatial data structures
- DirectX 11 has good interoperability between Compute shaders and graphics
    - can render results efficiently

Fluid Simulation for Games

- Fluids are well suited to GPU
  - data parallel
- Many different techniques
  - Eulerian (grid-based)
  - Lagrangian (particle-based)
  - Heightfield
- Each has its own strengths and weaknesses
- To achieve realistic results, games need to combine techniques

Particle Based Fluid Simulation

- Smoothed particle hydrodynamics (SPH)
- Good for spray, splashes
- Easy to integrate into games
  - no fixed domain
  - particles simple to collide with scene
- Simulation can be provided by
  - Physics middleware (e.g. Bullet, Havok, PhysX)
  - or custom DirectCompute or CPU code

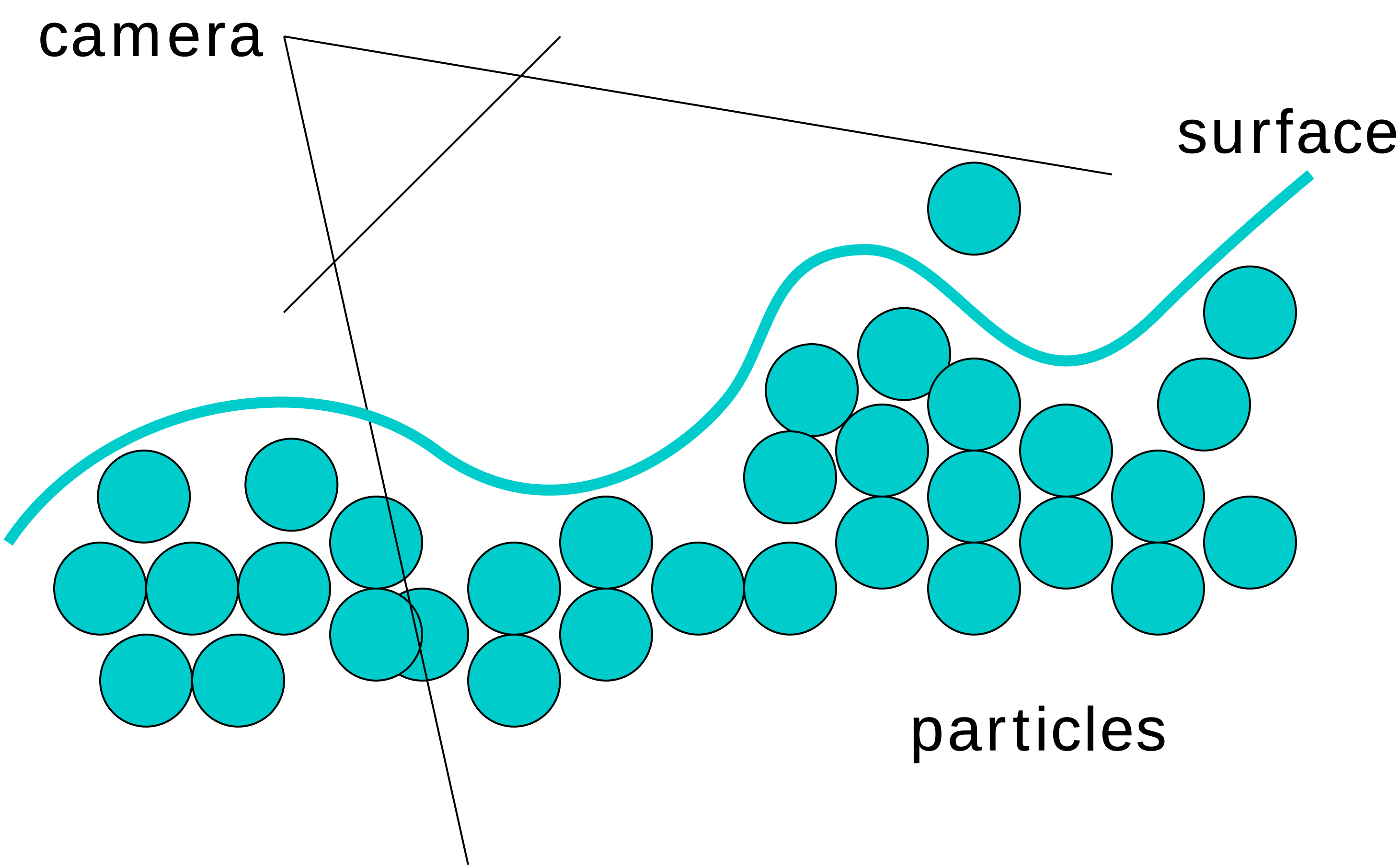
Fluid Rendering

- Rendering particle-based fluids is difficult
  - Simulation doesn't naturally generate a surface (no grid, no level set)
  - Just get particle positions and density
- Traditionally, rendering done using marching cubes
  - Generate density field from particles
  - Extract polygon mesh isosurface
  - Can be done on GPU, but very expensive

Screen Space Fluid Rendering

- Inspired by “Screen Space Meshes” paper (Müller et al)
- See: van der Laan *et al “Screen space fluid rendering with curvature flow”*, I3D 2009
- Operates entirely in screen-space
  - No meshes
- Only generates surface closest to camera

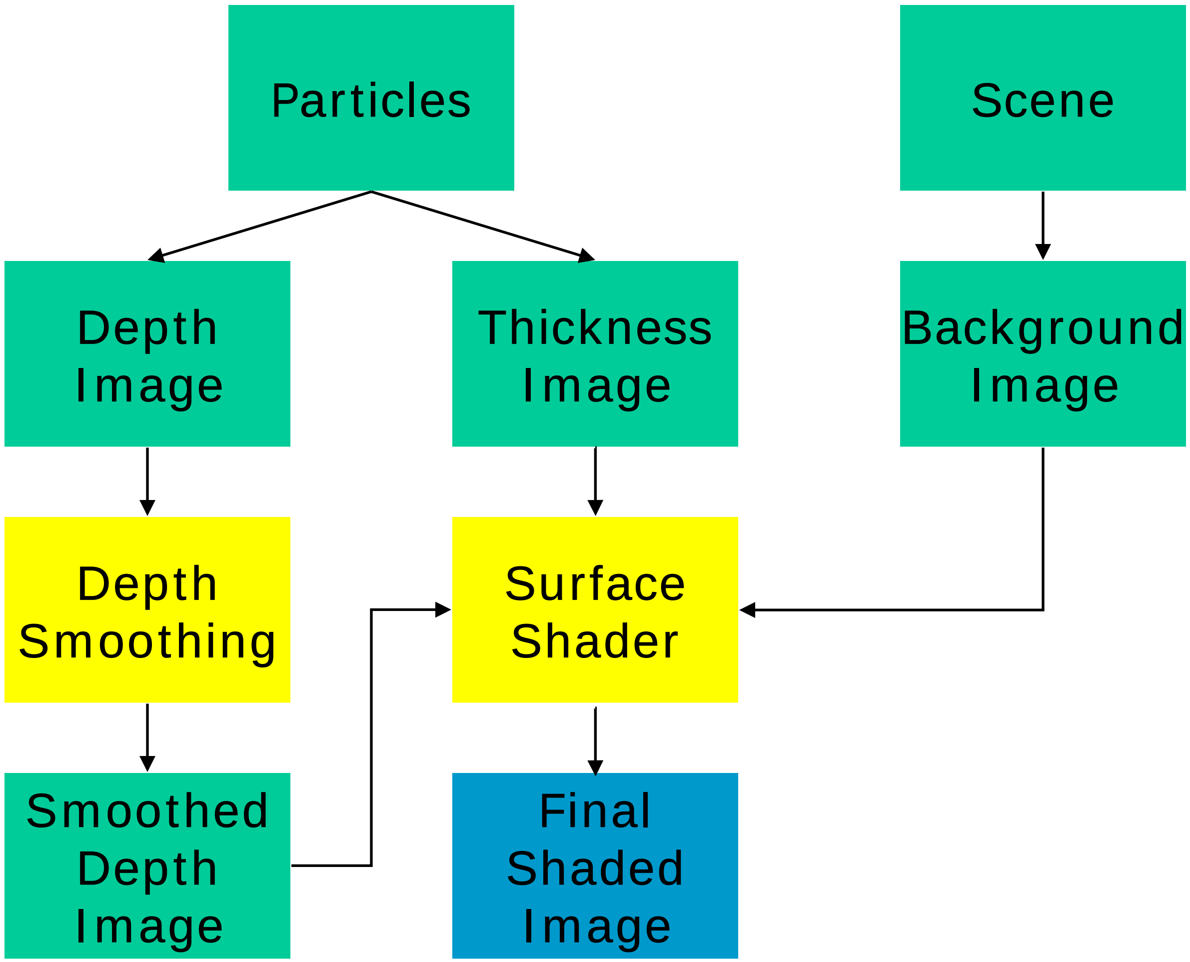
Screen Space Fluid Rendering



Screen Space Fluid Rendering - Overview

- Generate depth image of particles
  - Render as spherical point sprites
- Smooth depth image
  - Gaussian bilateral blur
- Calculate surface normals and position from depth
- Shade surface
  - Write depth to merge with scene

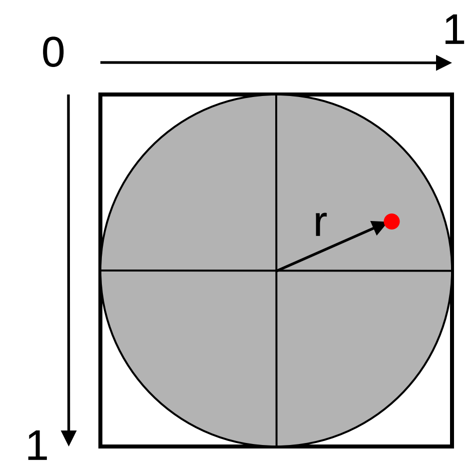
Screen Space Fluid Rendering



Rendering Particle Spheres

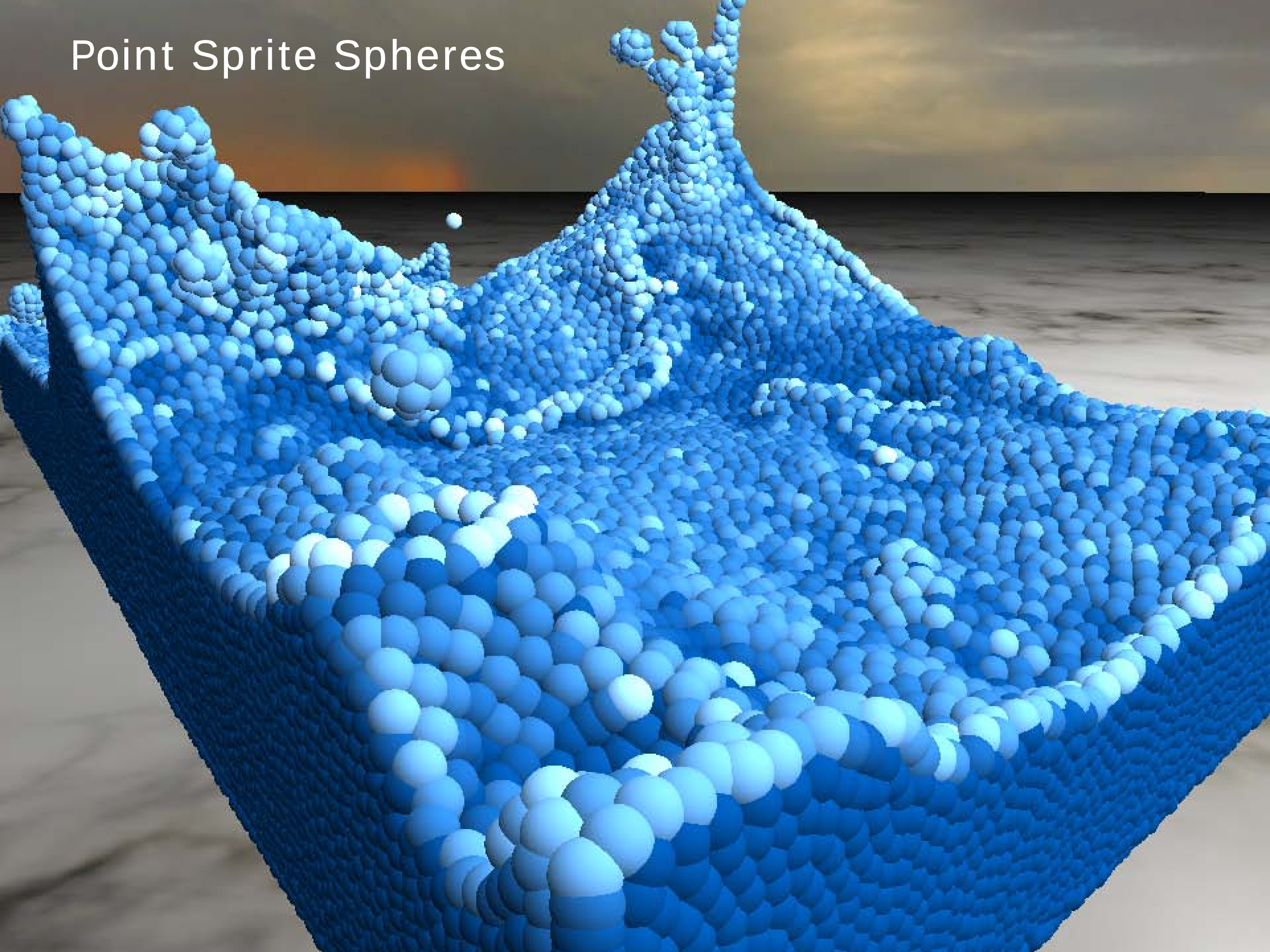
- Render as point sprites (quads)
- Calculate quad size in vertex shader (constant in world-space)
- Calculate sphere normal and depth in pixel shader
- Discard pixels outside circle
- Not strictly correct (perspective projection of a sphere can be an ellipsoid)
  - But works fine in practice

Rendering Particle Spheres

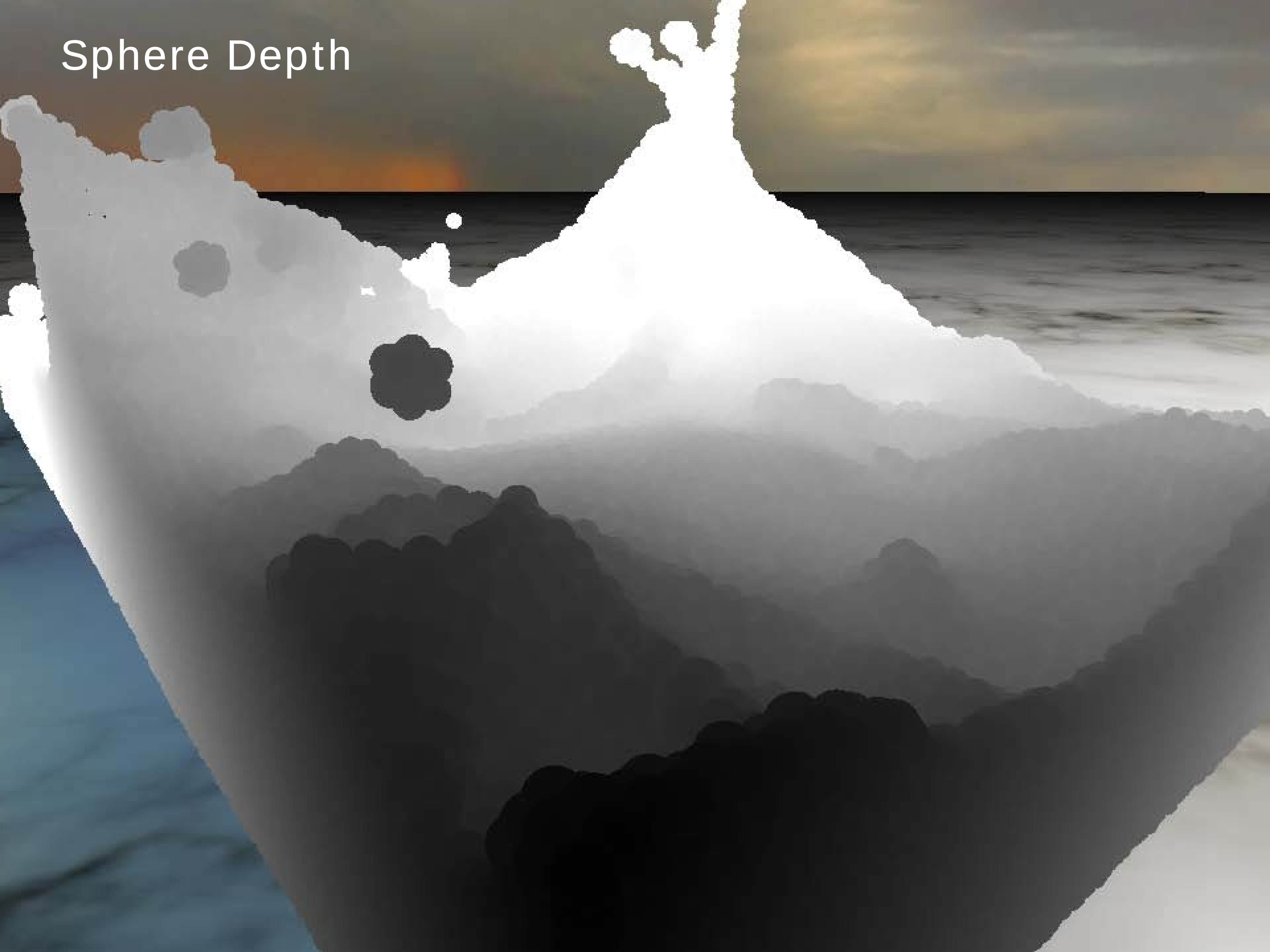



```
PSOutput particleSpherePS(
    float2 texCoord      : TEXCOORD0,
    float3 eyeSpacePos   : TEXCOORD1,
    float  sphereRadius  : TEXCOORD2,
    float4 color         : COLOR0)
{
    PSOutput OUT;

    // calculate eye-space sphere normal from texture coordinates
    float3 N;
    N.xy = texCoord*2.0-1.0;
    float r2 = dot(N.xy, N.xy);
    if (r2 > 1.0) discard;   // kill pixels outside circle
    N.z = -sqrt(1.0 - r2);

    // calculate depth
    float4 pixelPos = float4(eyeSpacePos + N*sphereRadius, 1.0);
    float4 clipSpacePos = mul(pixelPos, ProjectionMatrix);
    OUT.fragDepth = clipSpacePos.z / clipSpacePos.w;

    float diffuse = max(0.0, dot(N, lightDir));
    OUT.fragColor = diffuse * color;

    return OUT;
}
```

Point Sprite Spheres



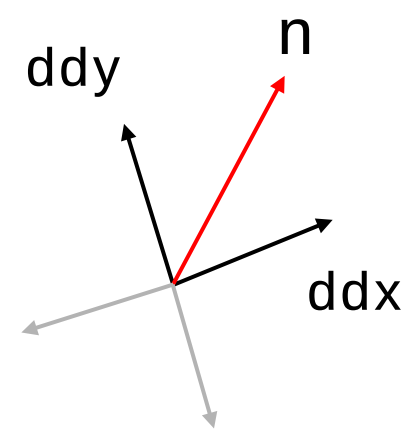
Sphere Depth



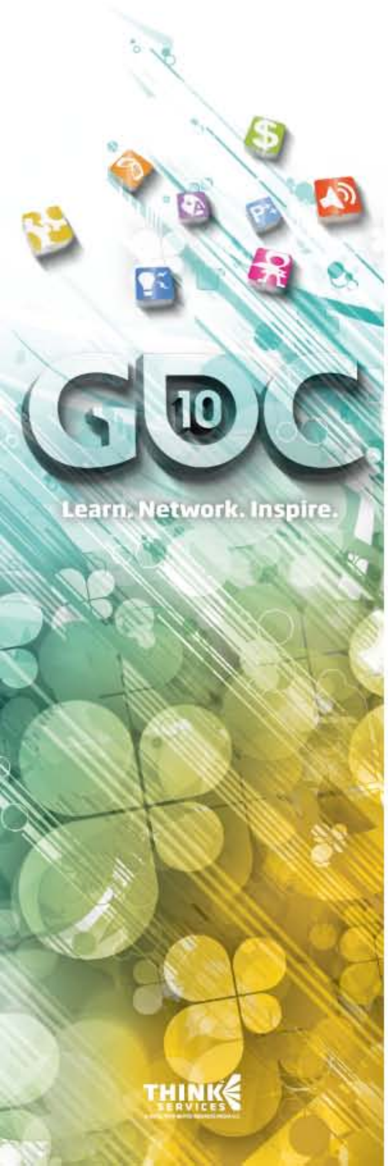
Calculating Normals

- Store eye-space sphere depth to floating point render target
- Can calculate eye-space position from UV coordinates and depth
- Use partial differences of depth to calculate normal
  - Look at neighbouring pixels
- Have to be careful at edges
  - Normal may not be well-defined
  - At edges, use difference in opposite direction (hack!)

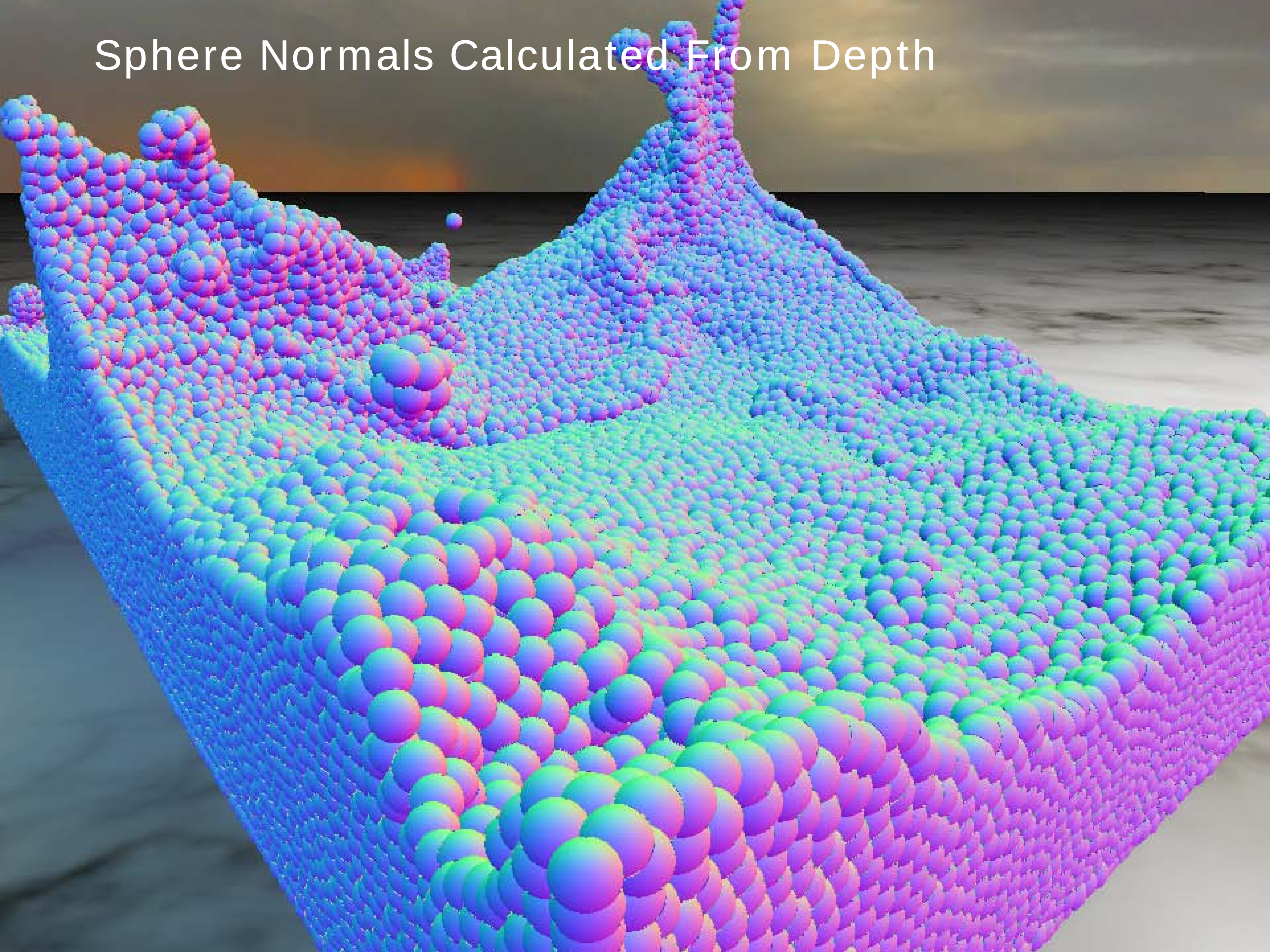
Calculating Normals (code)




```
// read eye-space depth from texture
float depth = tex2D(depthTex, texCoord).x;
if (depth > maxDepth) {
    discard;
    return;
}

// calculate eye-space position from depth
float3 posEye = uvToEye(texCoord, depth);

// calculate differences
float3 ddx = getEyePos(depthTex, texCoord + float2(texelSize, 0)) - posEye;
float3 ddx2 = posEye - getEyePos(depthTex, texCoord + vec2(-texelSize, 0));
if (abs(ddx.z) > abs(ddx2.z)) {
    ddx = ddx2;
}

float3 ddy = getEyePos(depthTex, texCoord[0] + vec2(0, texelSize)) - posEye;
float3 ddy2 = surfacePosEye - getEyePos(depthTex, texCoord + vec2(0, -texelSize));
if (abs(ddy2.z) < abs(ddy.z)) {
    ddy = ddy2;
}

// calculate normal
vec3 n = cross(ddx, ddy);
n = normalize(n);
```



Sphere Normals Calculated From Depth

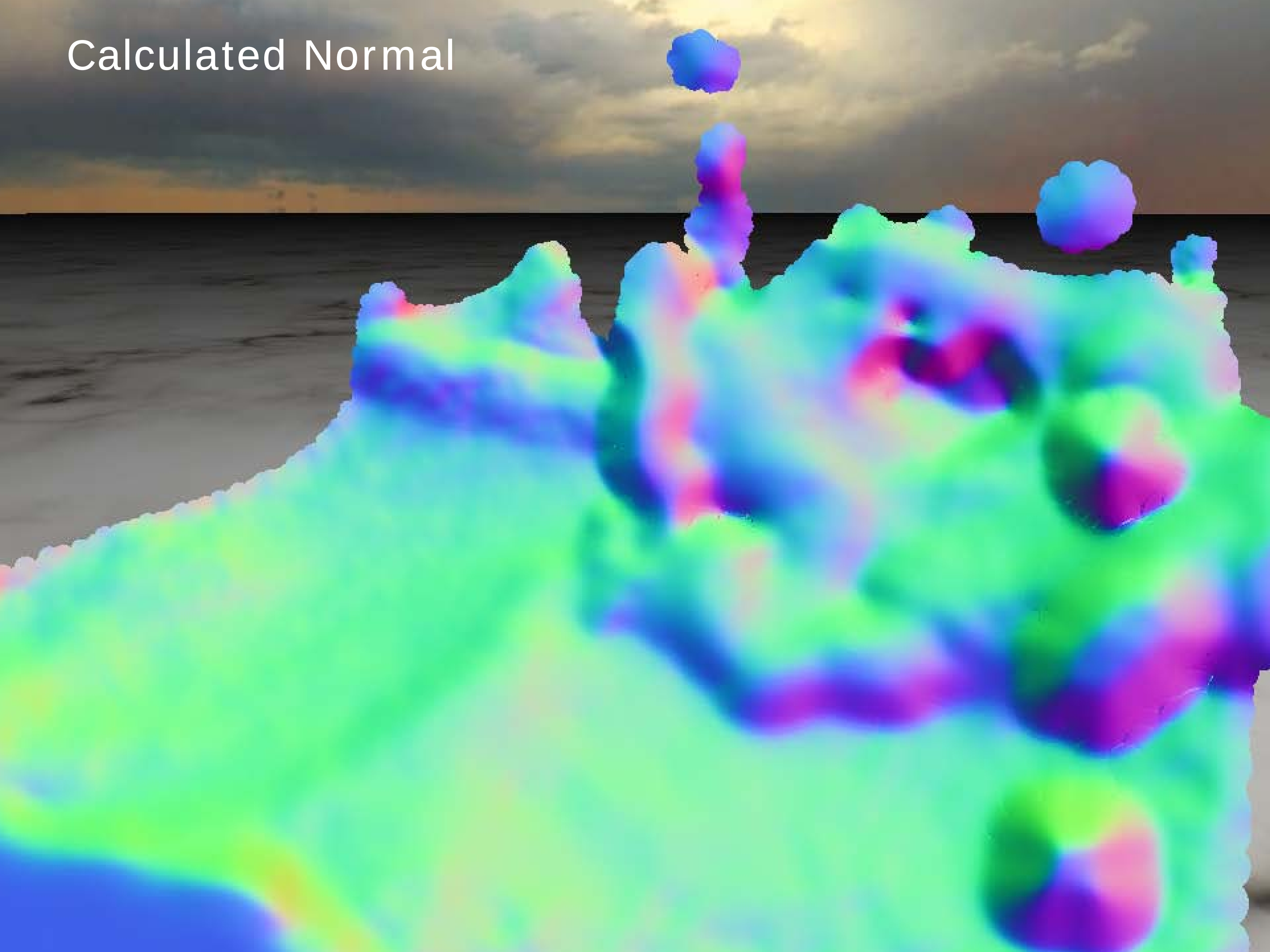


Smoothing

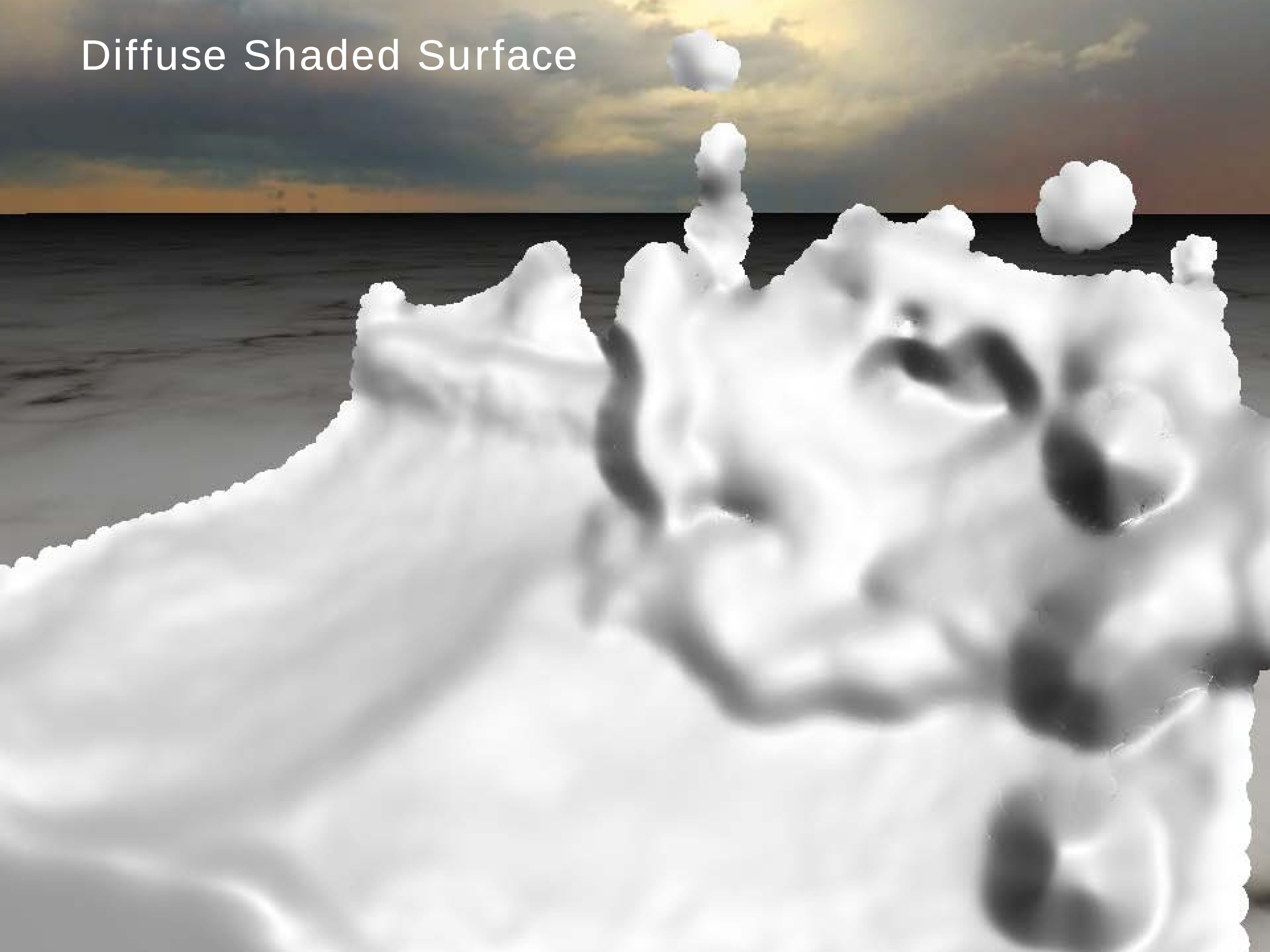
- By blurring the depth image, we can smooth the surface
- Use Gaussian blur
- Needs to be view-invariant
  - Constant width in world space
  - -> Variable in screen-space space
- Calculate filter width in shader
  - Clamped to maximum radius in screen space (e.g. 50 pixels) for performance

Sphere Depth


Naively Smoothed Depth


Calculated Normal

Diffuse Shaded Surface



Bilateral Filter

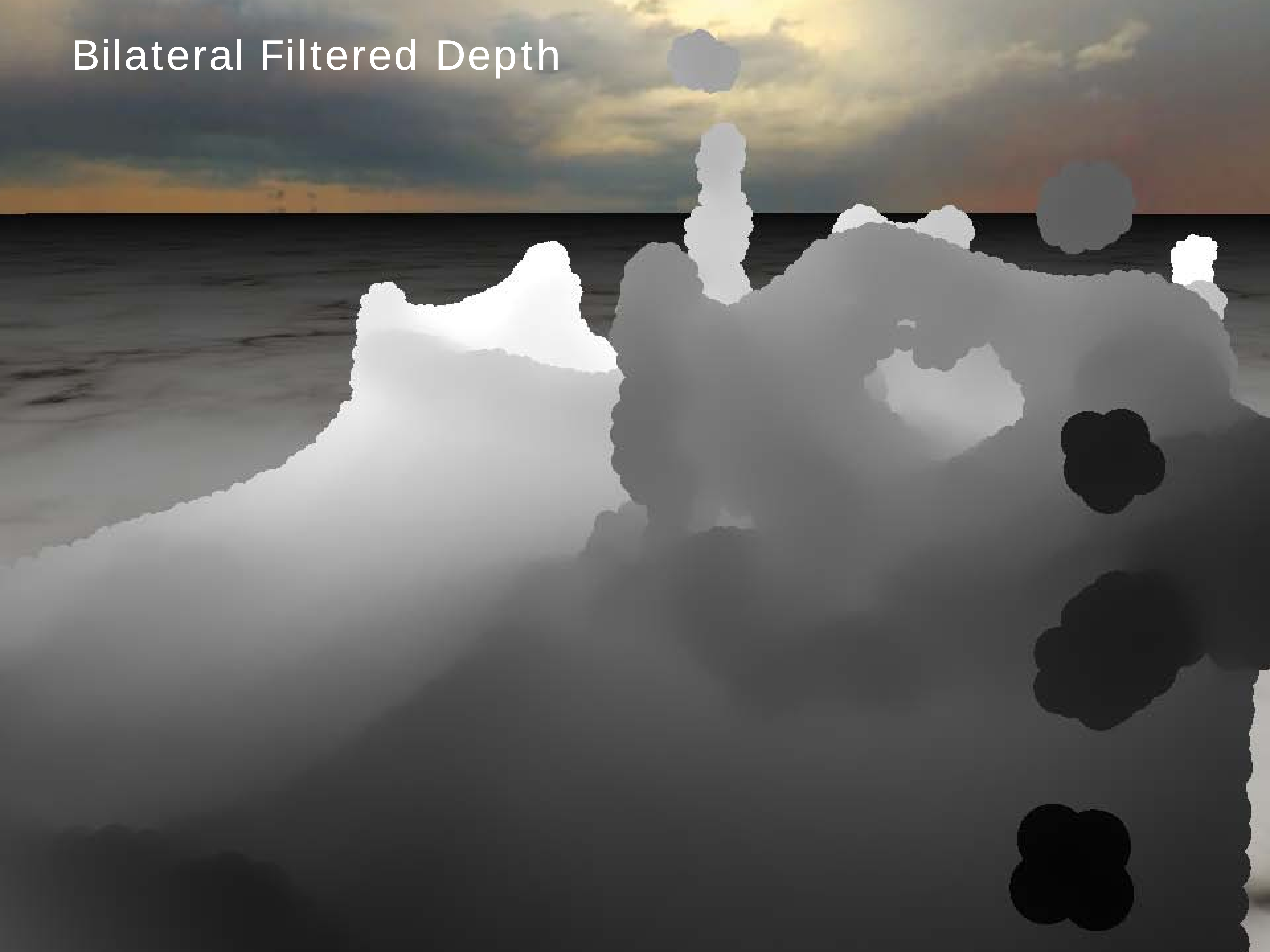
- Problem: we want to preserve the silhouette edges in depth image
  - So particles don't get blended into background surfaces
- Solution: Bilateral Filter
  - Edge-preserving smoothing filter
  - Called "Surface Blur" in Photoshop
  - Regular Gaussian filter is based only on only distance in image domain
  - Bilateral filter also looks at difference in range (image values)
  - Two sets of weights

Bilateral Filter Code

```
float depth = tex2D(depthSampler, texcoord).x;

float sum = 0;
float wsum = 0;
for(float x=-filterRadius; x<=filterRadius; x+=1.0) {
    float sample = tex2D(depthSampler, texcoord + x*blurDir).x;

    // spatial domain
    float r = x * blurScale;
    float w = exp(-r*r);

    // range domain
    float r2 = (sample - depth) * blurDepthFalloff;
    float g = exp(-r2*r2);

    sum += sample * w * g;
    wsum += w * g;
}

if (wsum > 0.0) {
    sum /= wsum;
}
return sum;
```

Note – not optimized!

Sphere Depth

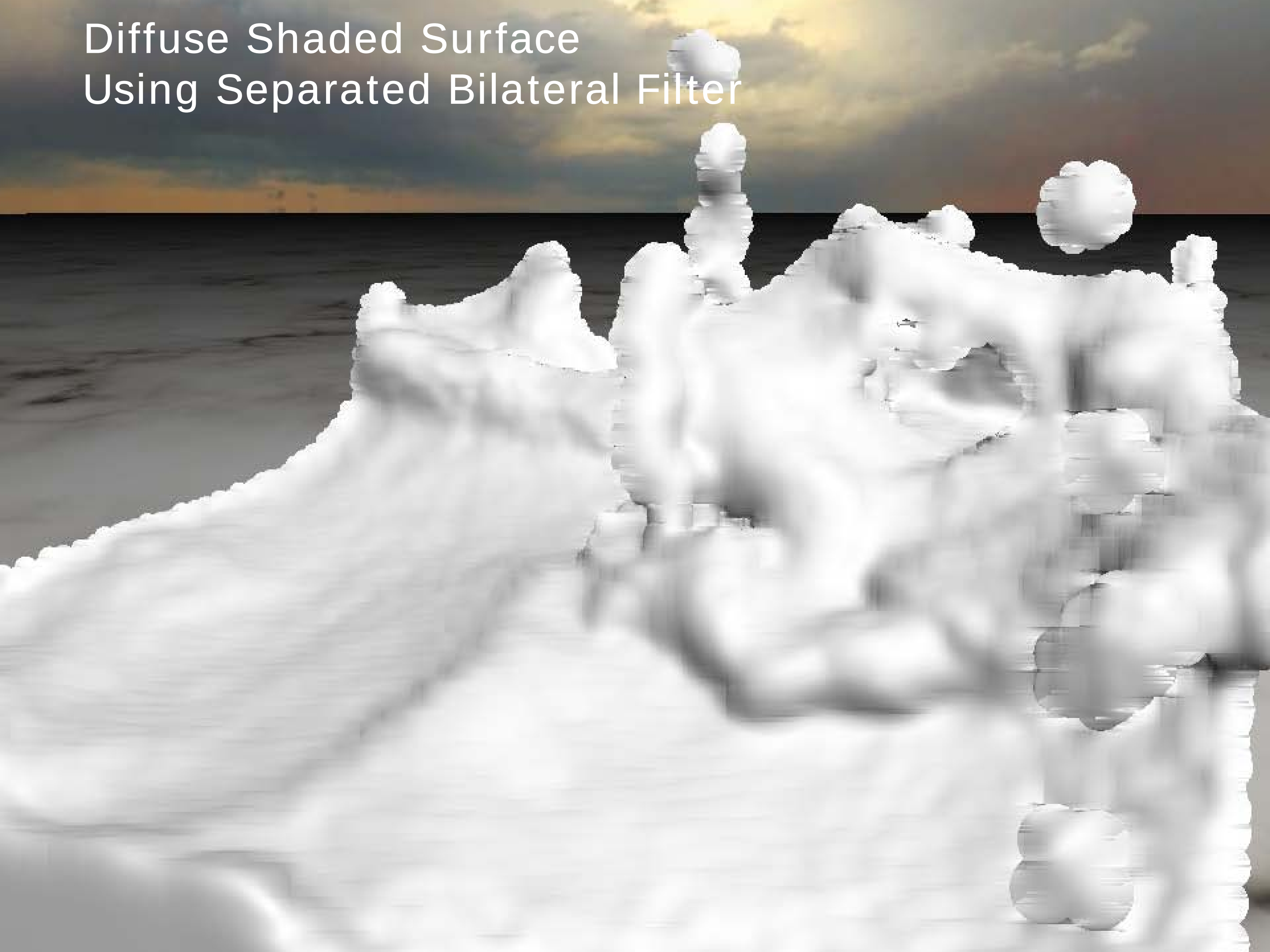

Bilateral Filtered Depth

Diffuse Shaded Surface

Bilateral Filter

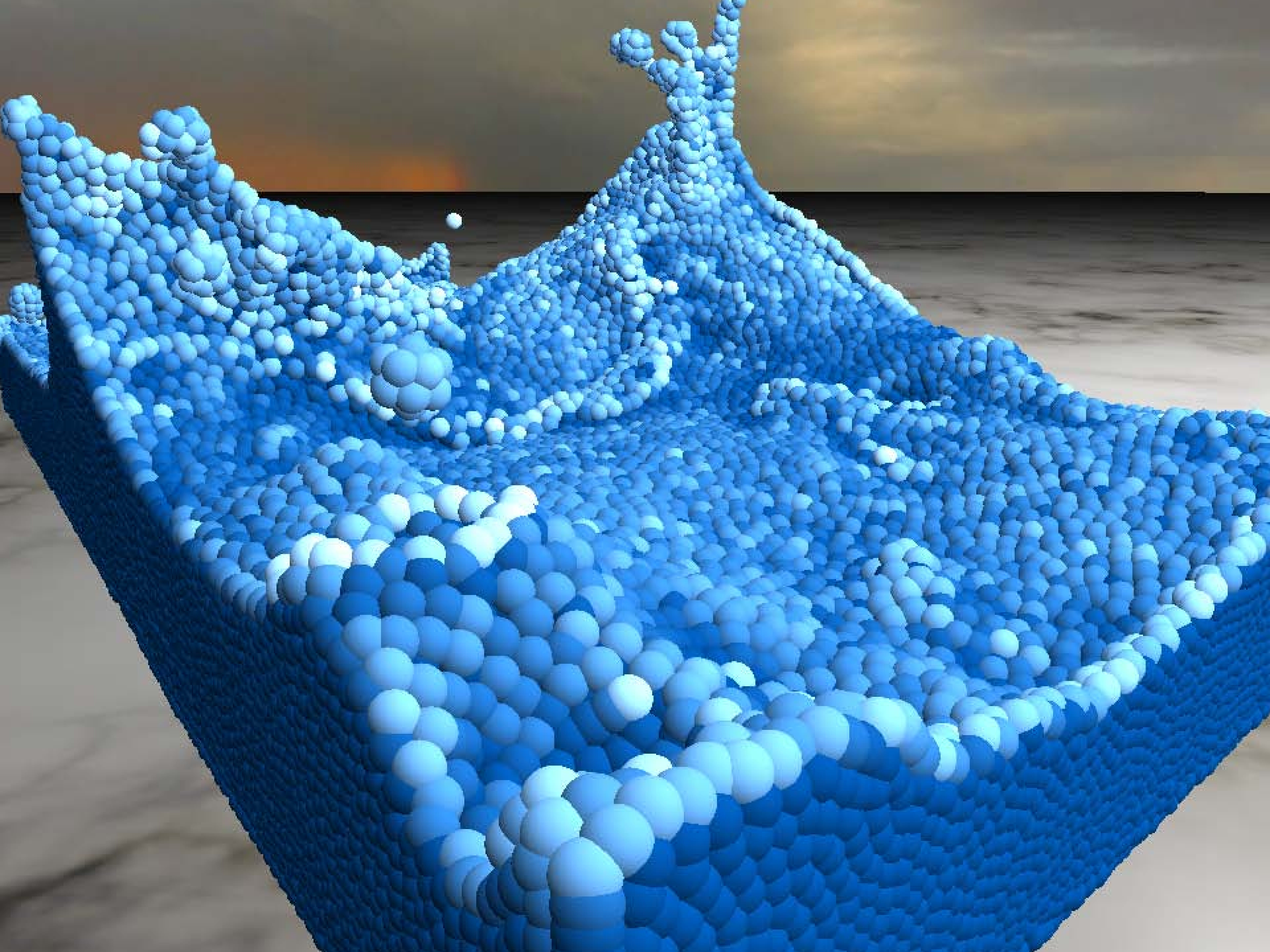
- Bilateral filter is not strictly separable
  - Can't separate into X and Y blur passes
  - Non-separable 2D filter is very expensive
- But we can get away with separating, with some artifacts
  - Artifacts not very visible once other shading added

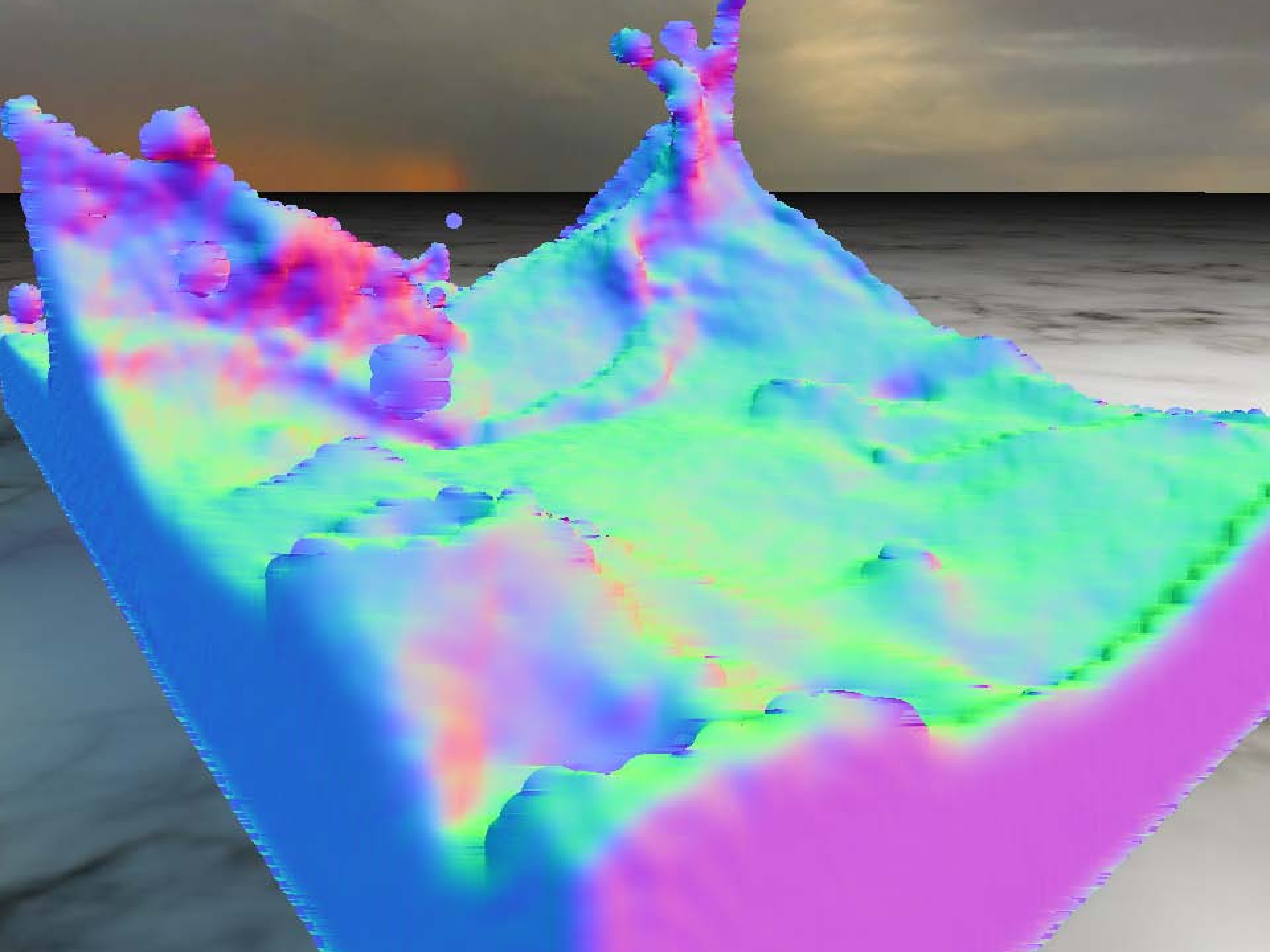
Diffuse Shaded Surface Using Separated Bilateral Filter



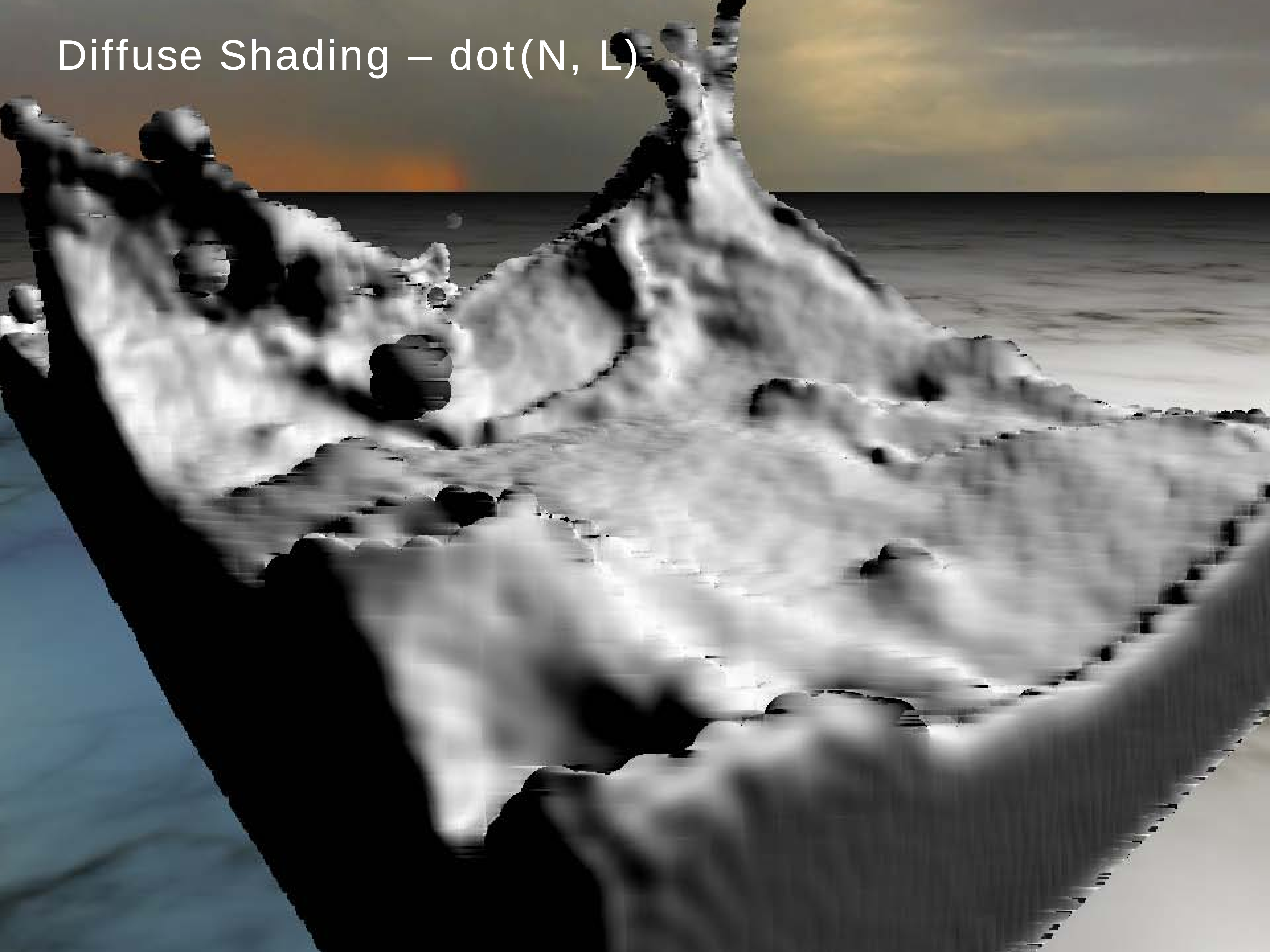
Surface Shading

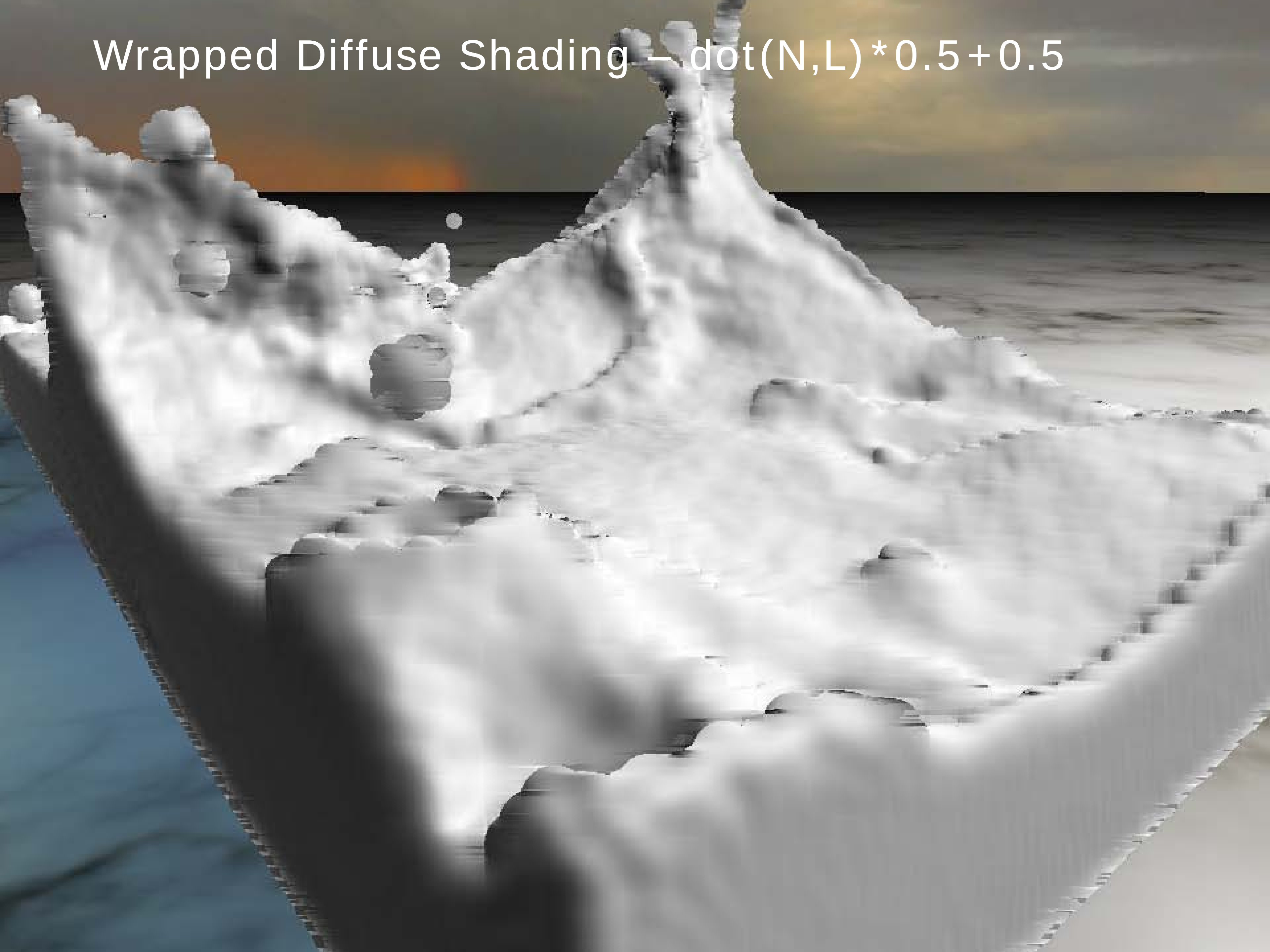
- Why not just blur normals?
- We also calculate eye-space surface position from the smoothed depth
  - Important for accurate specular reflections
- Once we have a per-pixel surface normal and position, can shade as usual

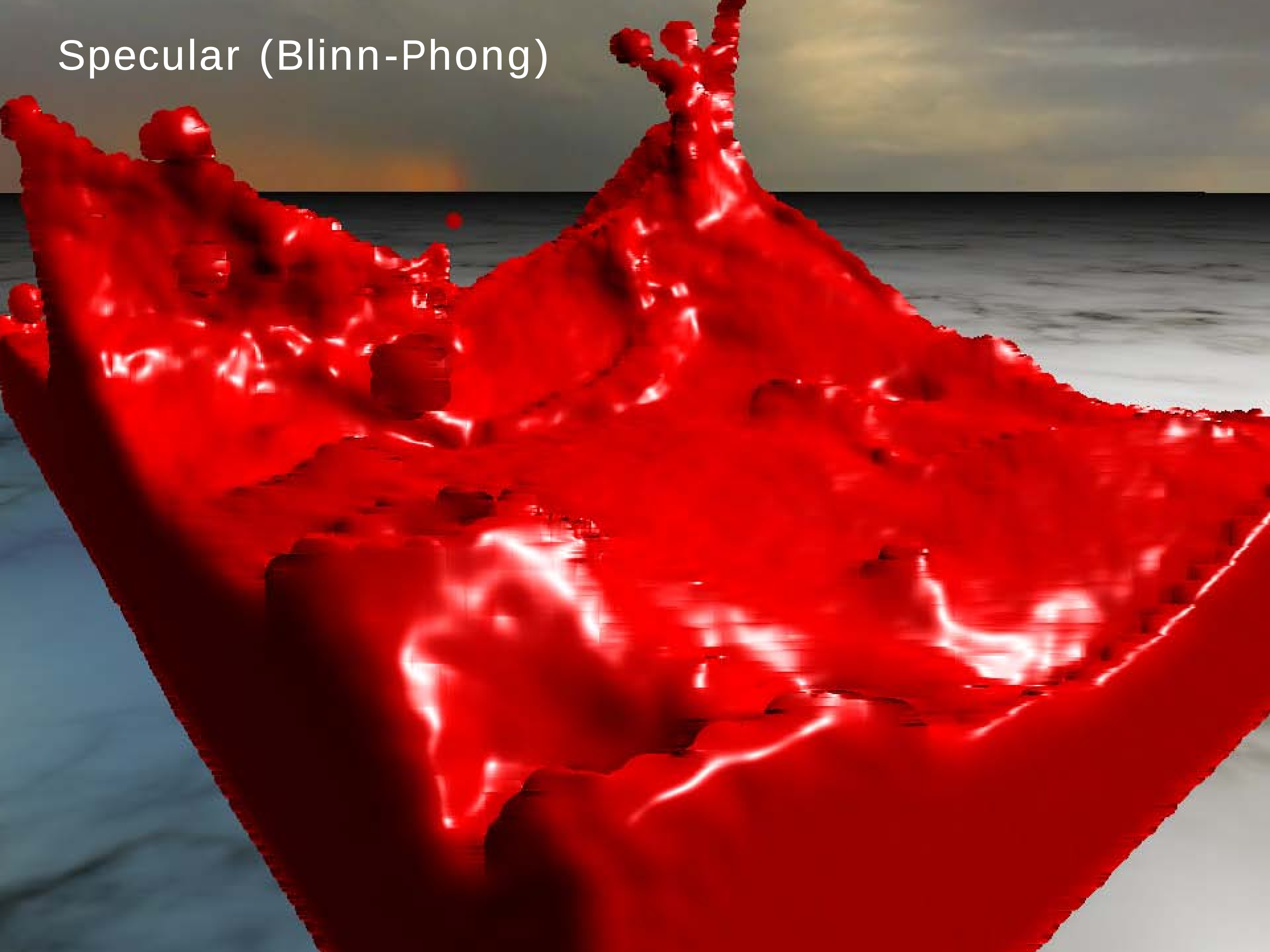




Diffuse Shading – dot(N, L)




Wrapped Diffuse Shading – dot(N,L)*0.5+0.5


Specular (Blinn-Phong)

Fresnel

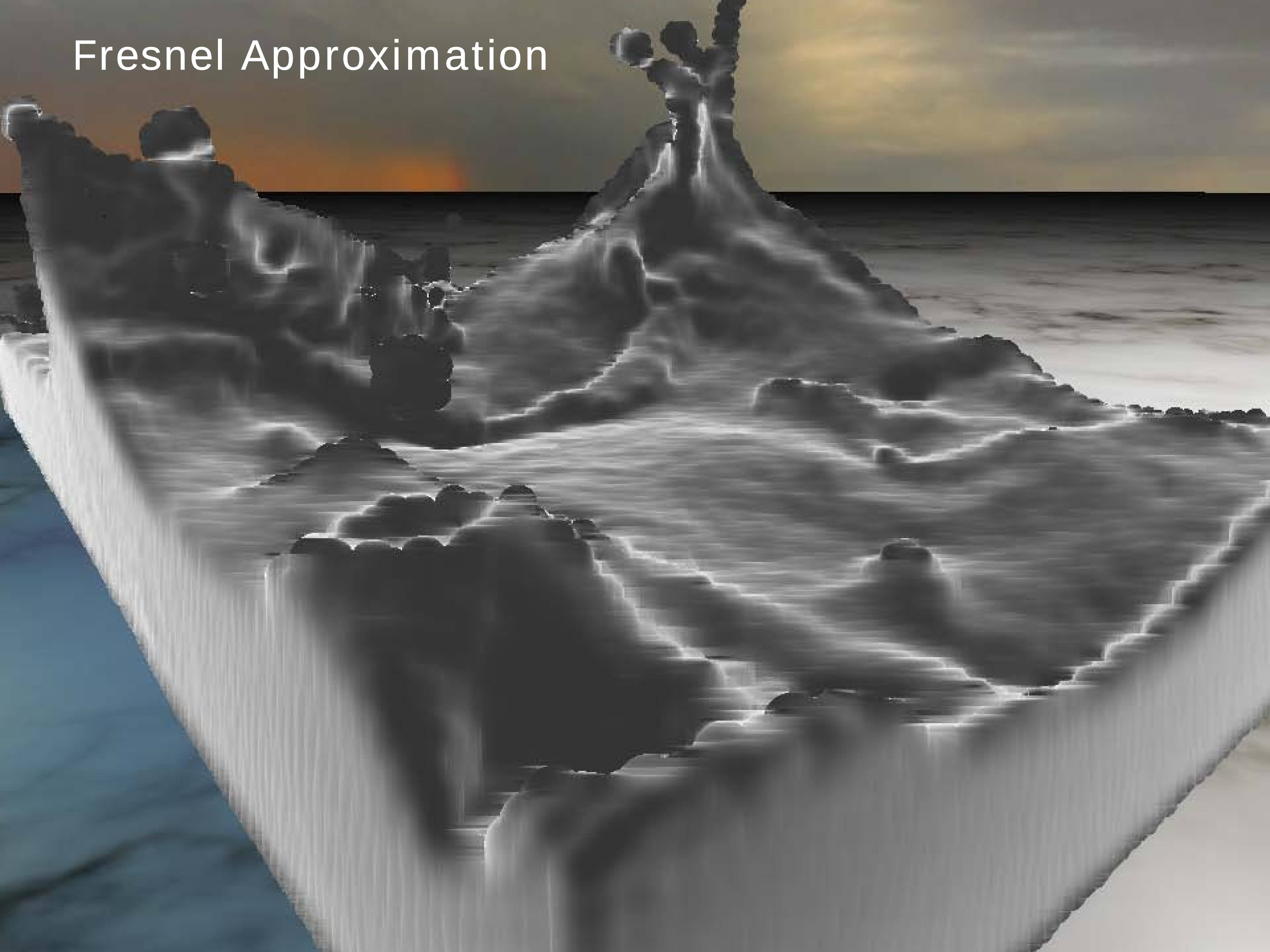
- Surfaces are more reflective at glancing angles
- Schlick's approximation

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

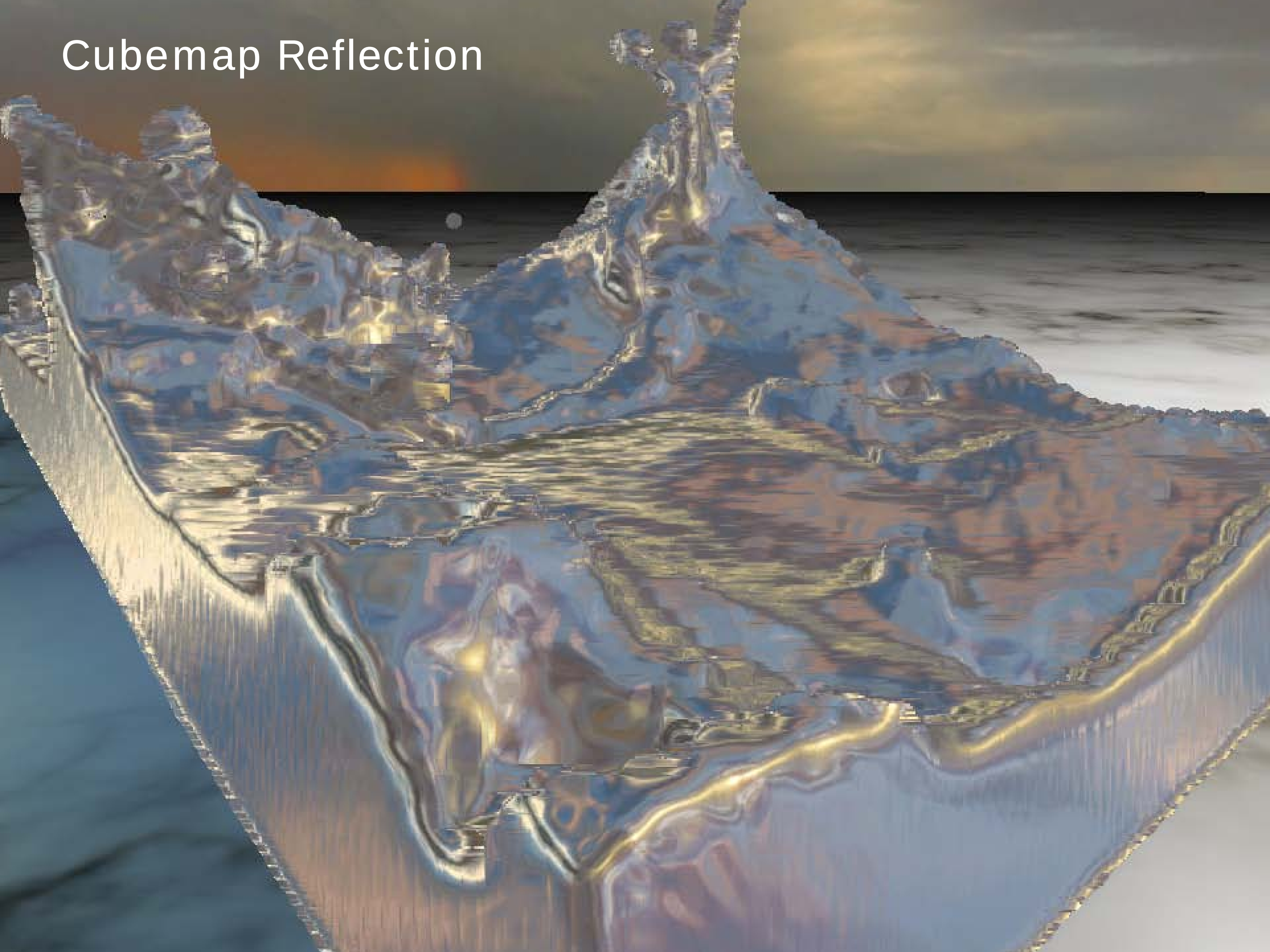
- θ is incident angle

  cos(θ) =dot(N, V)
- R_0 is the reflectance at normal incidence
- Can vary exponent for visual effect

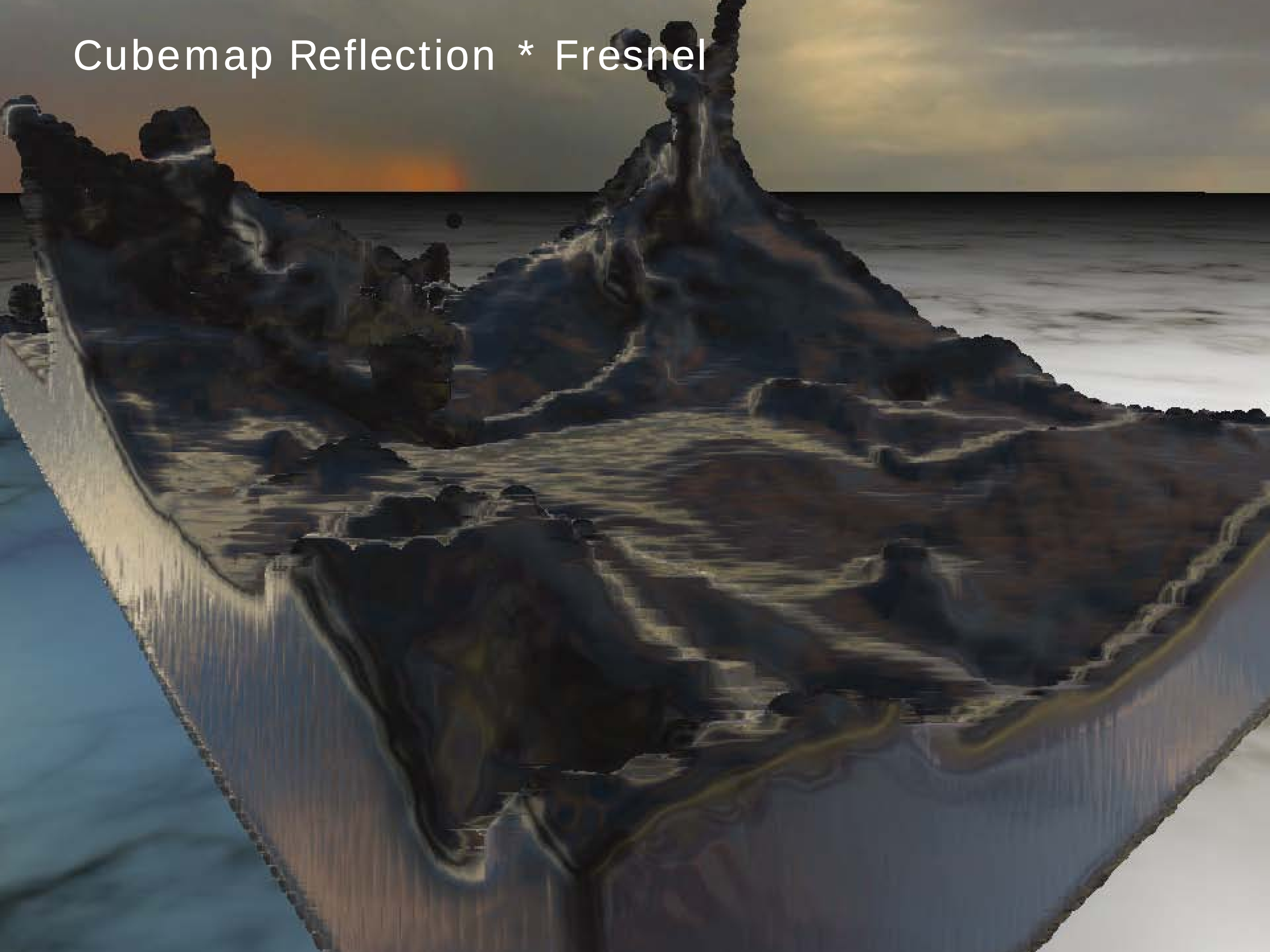
Fresnel Approximation



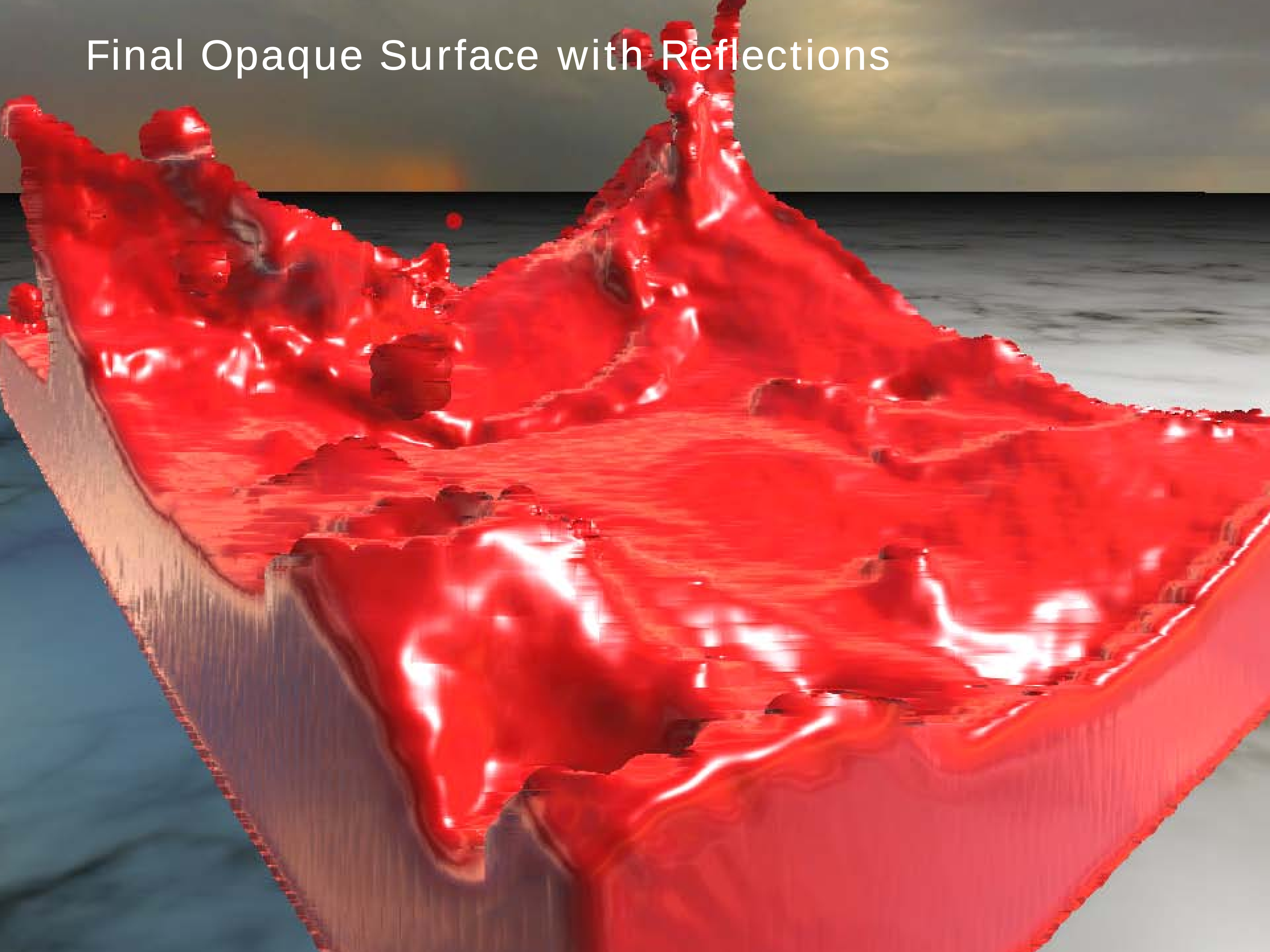
Cubemap Reflection



Cubemap Reflection * Fresnel



Final Opaque Surface with Reflections



Thickness Shading

- Fluids are often transparent
- Screen-space surface rendering only generates surface nearest camera
  - Looks strange with transparency
  - Can't see surfaces behind front
- Solution – shade fluid as semi-opaque using thickness through volume to attenuate color

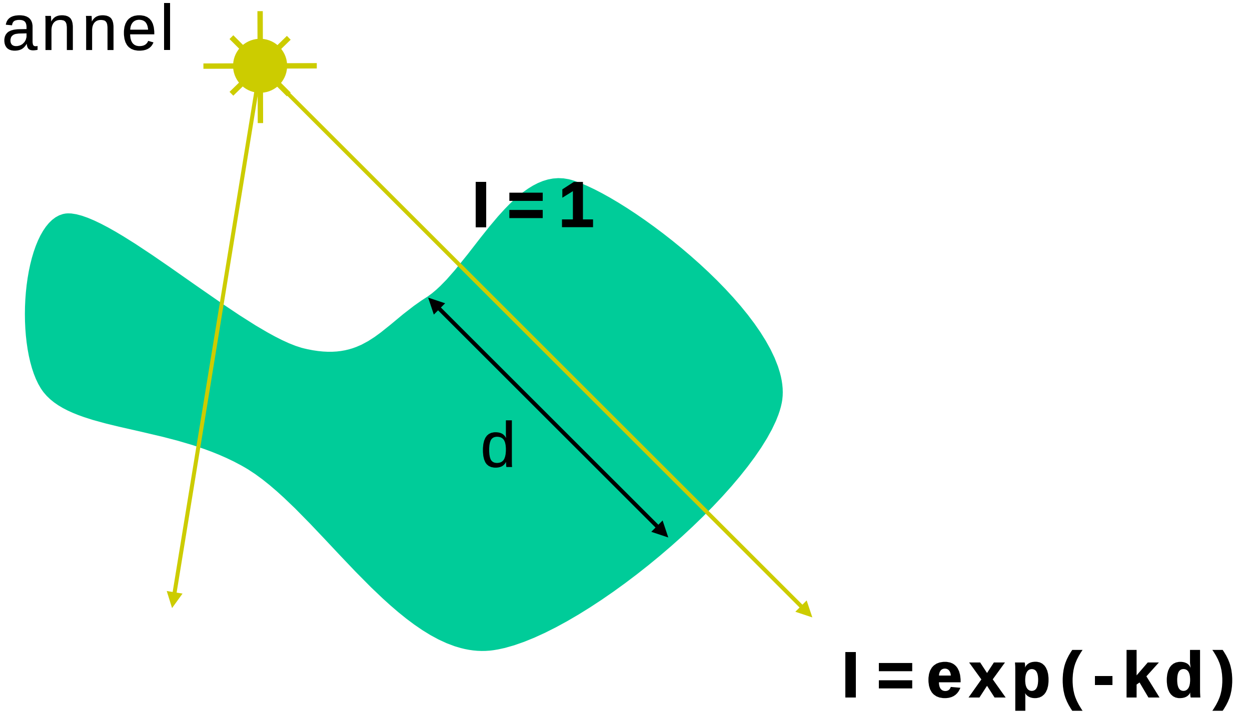
Generating Thickness

- Render particles using additive blending (no depth test)
    - Store in off-screen render target
    - Render smooth Gaussian splats
    - or just discs, and then blur
- Only needs to be approximate
- Very fill-rate intensive
    - Can render at lower resolution

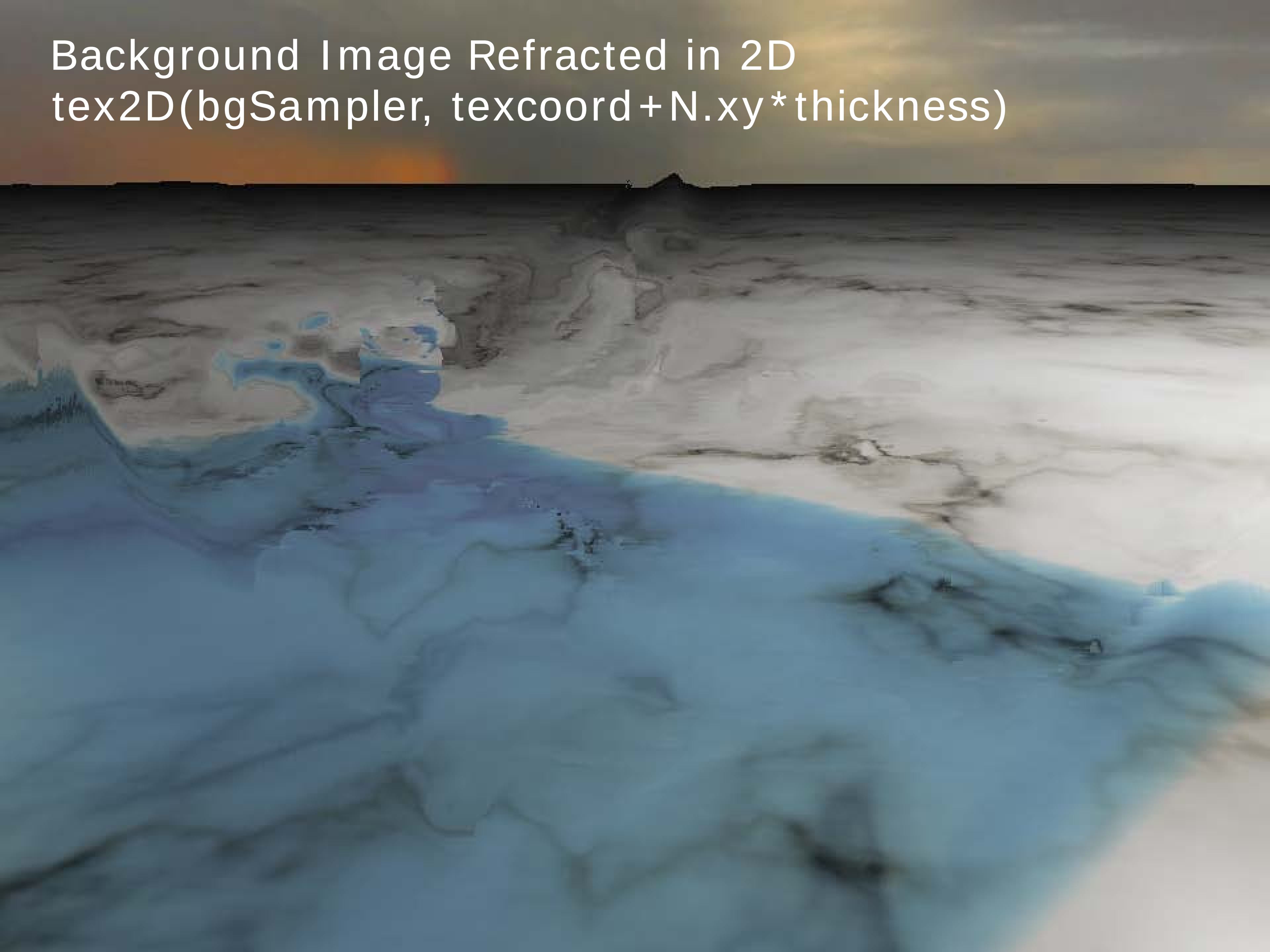
Volume Thickness

Volumetric Absorption

- Beer's Law
- Light decays exponentially with distance
- Use different constant k for each color channel

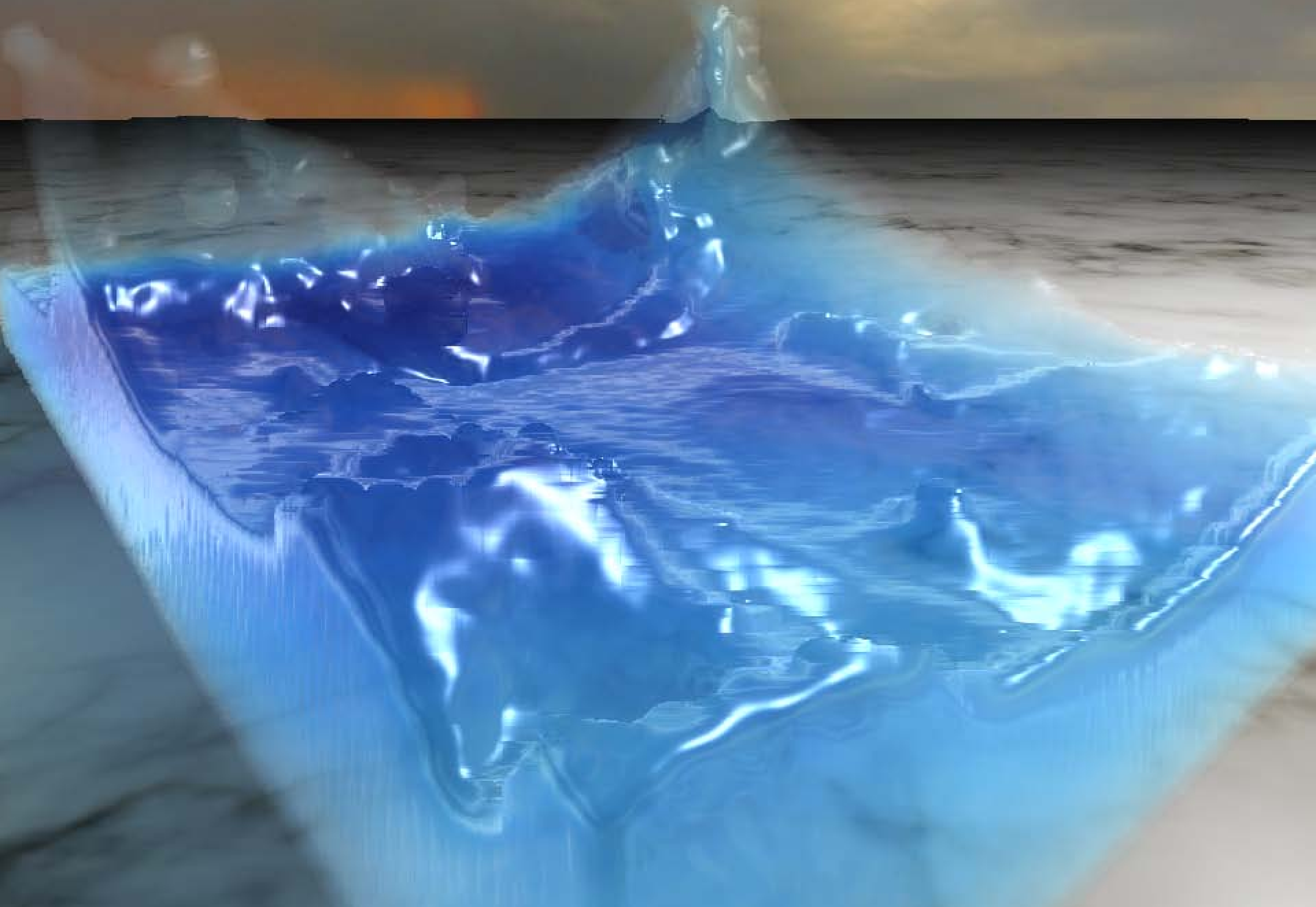


Color due to Absorption


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

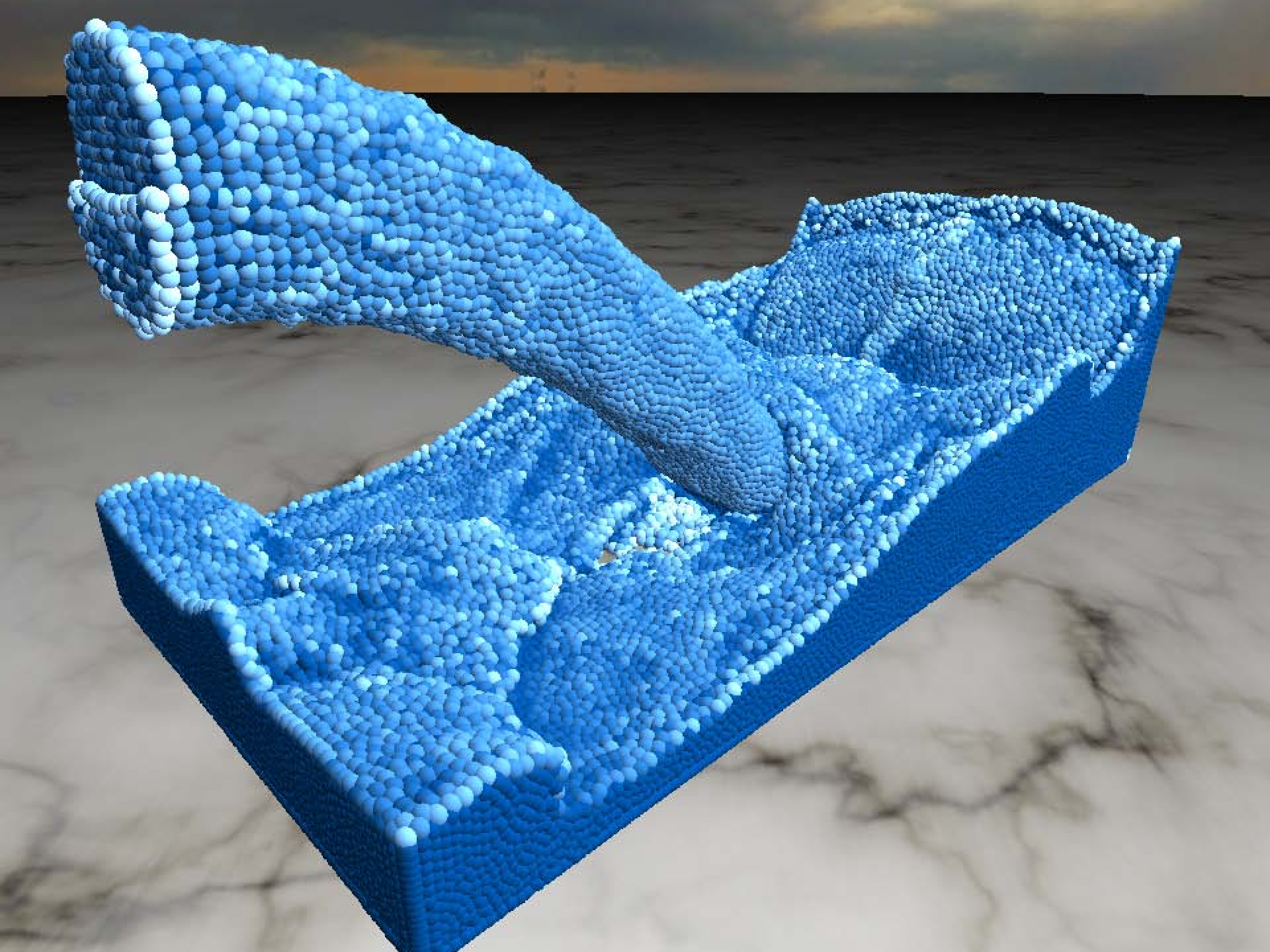
Transparency (based on thickness)

Final Shaded Translucent Surface



Shadows

- Since fluid is translucent, we expect it to cast coloured shadows
- Solution - render fluid surface again (using same technique), but from light's point of view
- Generate depth (shadow) map and color map (thickness)
- Project onto receivers (surface and ground plane)



No Shadows


Shadow Map


With Shadows

Problems

- Only generates surface closest to camera
  - Hidden somewhat by thickness shading
- Could be correctly rendered using ray tracing
  - Multiple refractions, reflections
- Possible to ray trace using the same uniform grid acceleration structure used for simulation
  - But still quite slow today

Artifact – can’t see further surfaces through volume

Caustics

- Refractive caustics are generated when light shines through a transparent and refractive material
- Light is focused into distinctive patterns

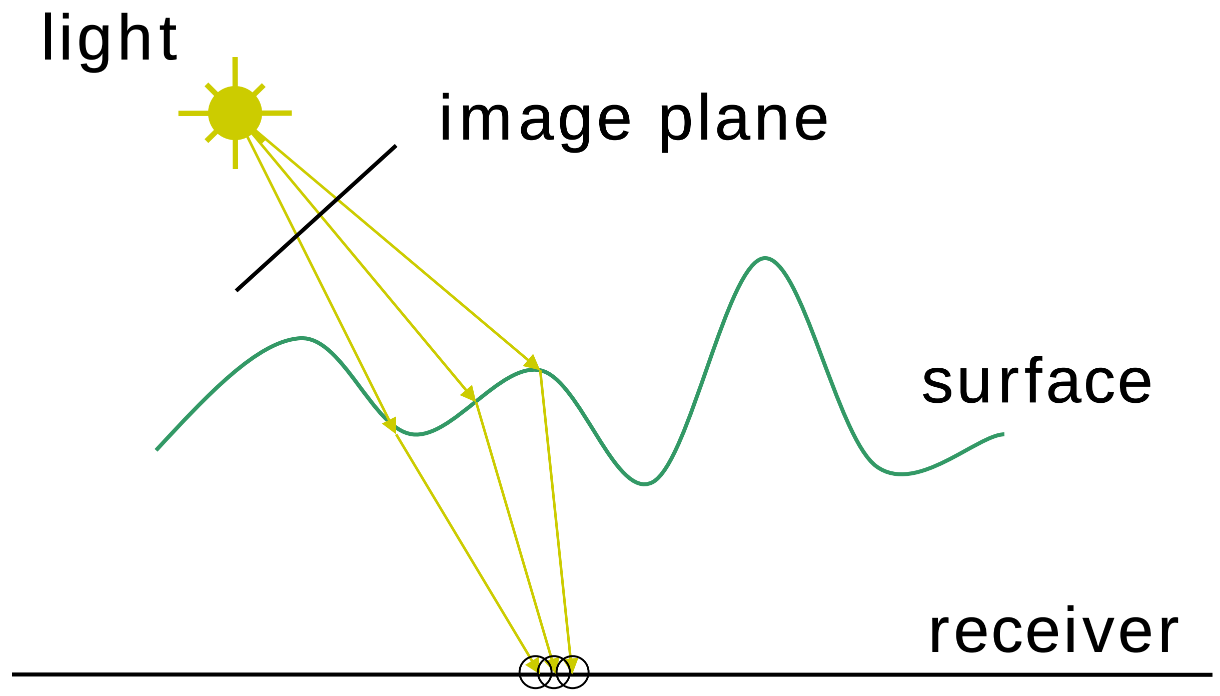
Caustics

Image by Rob Ireton

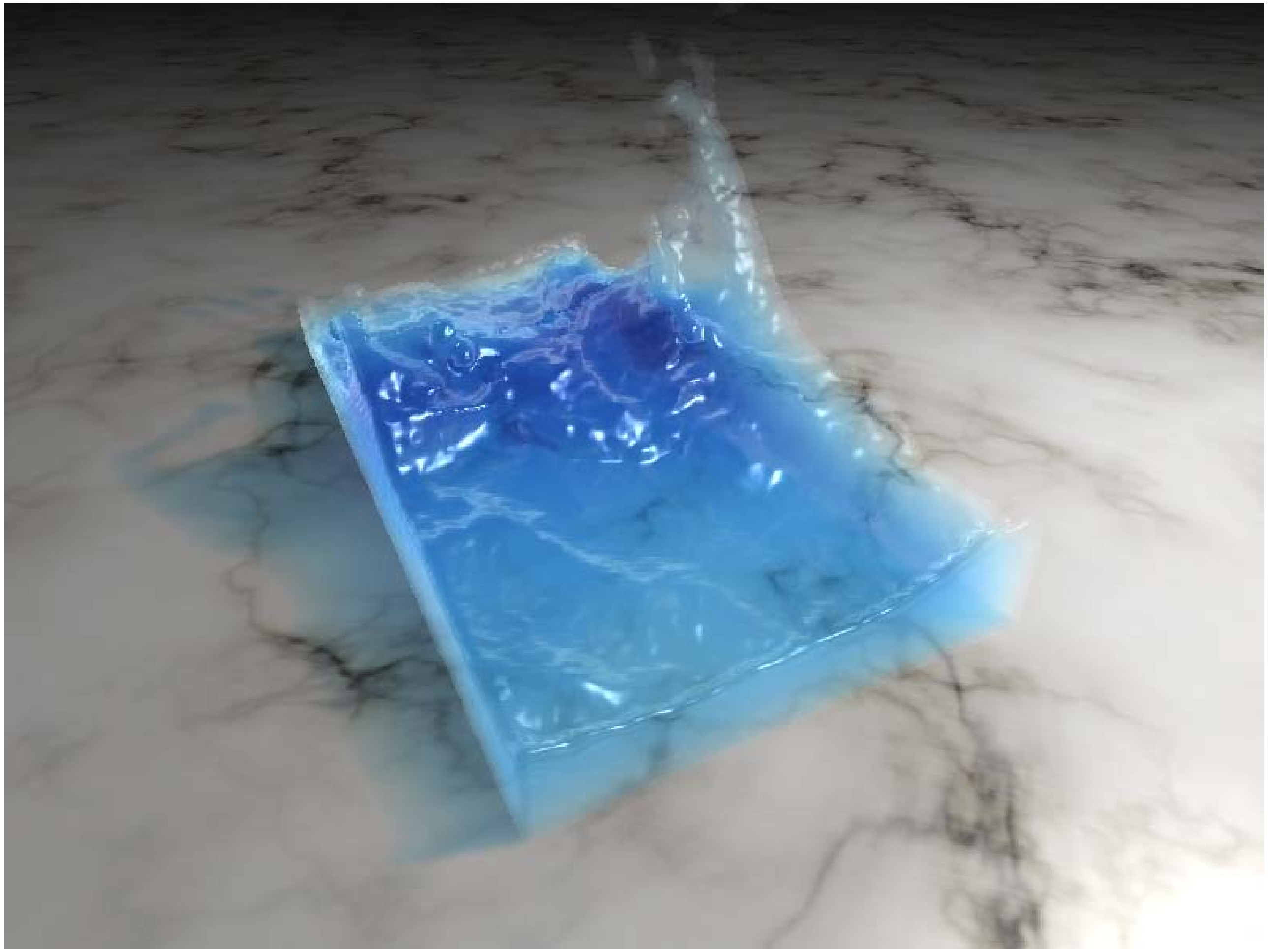
Caustics Algorithm

- We use a simple image-space technique
  - Similar to Wyman et al (see refs.)
- For each point in light view, calculate ray refracted through surface from light
  - uses surface position and normal
- Intersect ray with ground plane
- Render point splats ("photons") with additive blending

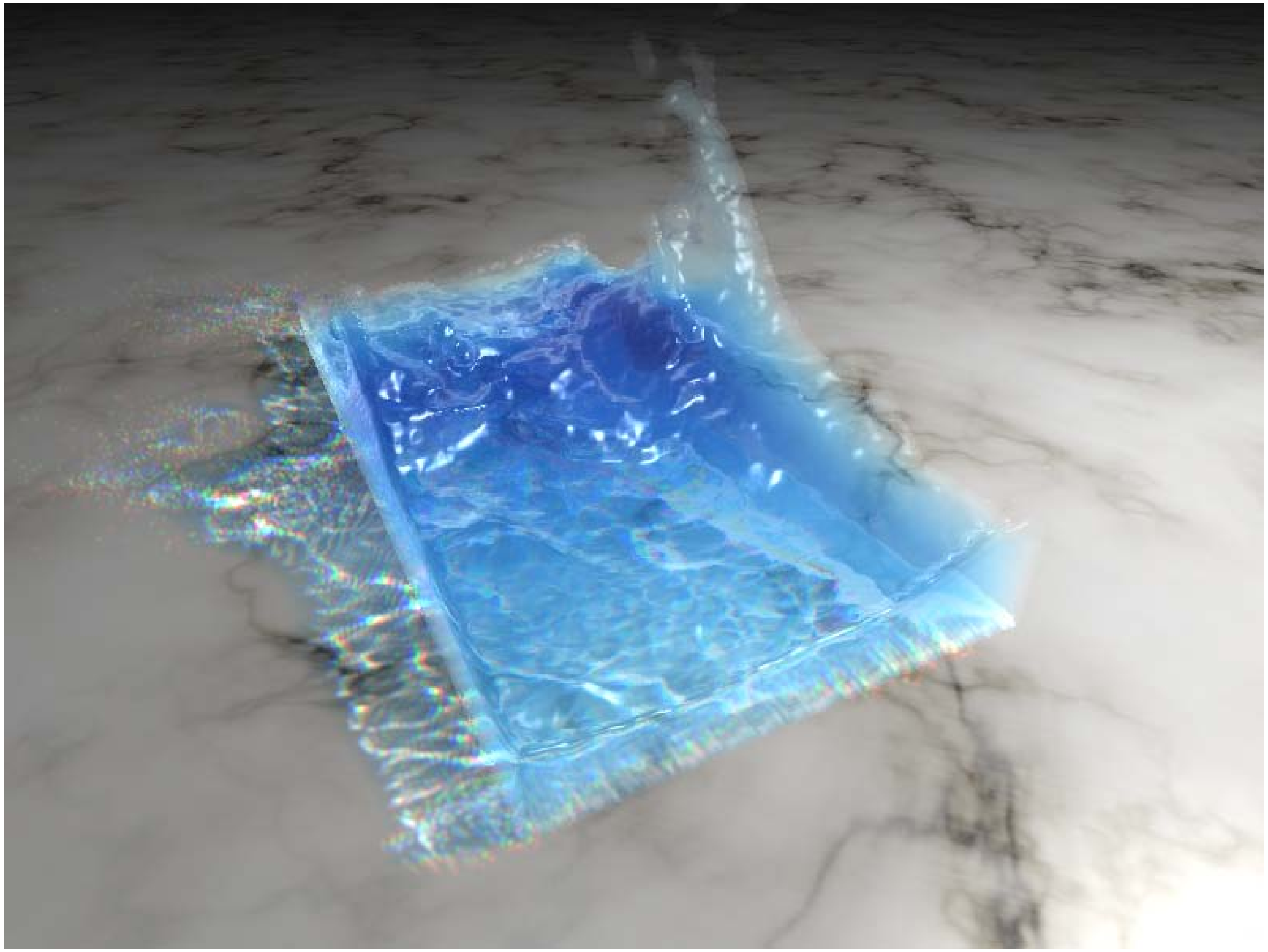
Caustics Diagram



Without Caustics



With Caustics



Caustics

- Note - caustics are only cast on ground plane, not on fluid surface!
- Can perform multiple times with different indices of refraction to simulate refractive dispersion (R, G, B)
- Quite expensive – requires rendering e.g. 512*512 = 256K points

Adding Surface Detail

- Surface can be too smooth
  - Doesn't show flow well
- Solution: add noise
- Render spheres again, using 3D noise texture in object-space
  - Moves with fluid
- Store in noise render target
  - Can be used during surface shading to perturb normal

DEMO

Summary

- Particle-based fluids are practical for use in games using today's hardware
- Rendering particle-based fluids can be simple and fast

Future Work

- Use Compute Shader for more efficient bilateral blur
  - Similar to diffusion DOF
- Polygon mesh collisions using BVH
- Add spray / foam
- Wet maps
- Direct3D 11 sample to be released in SDK soon

Questions?

Thanks

- Wladimir J. van der Laan, Rouslan Dimitrov, Miguel Sainz

References

- Robert Bridson, “Fluid Simulation for Computer Graphics”, A K Peters, 2008
- M. Müller, S. Schirm, S. Duthaler, ”**Screen Space Meshes”**, in *Proceedings of ACM SIGGRAPH / EUROGRAPHICS Symposium on Computer Animation* (SCA), 2007
- CORDS, H., AND STAADT, O. 2008. “**Instant Liquids”**. In Poster Proceedings of ACM Siggraph/Eurographics Symposium on Computer Animation
- Wladimir J. van der Laan, Simon Green, Miguel Sainz, “**Screen space fluid rendering with curvature flow”,** Proceedings of the 2009 symposium on Interactive 3D graphics and games
- Chris Wyman and Scott Davis. "**Interactive Image-Space Techniques for Approximating Caustics**." *ACM Symposium on Interactive 3D Graphics and Games*, 153-160. (March 2006)