Agenda

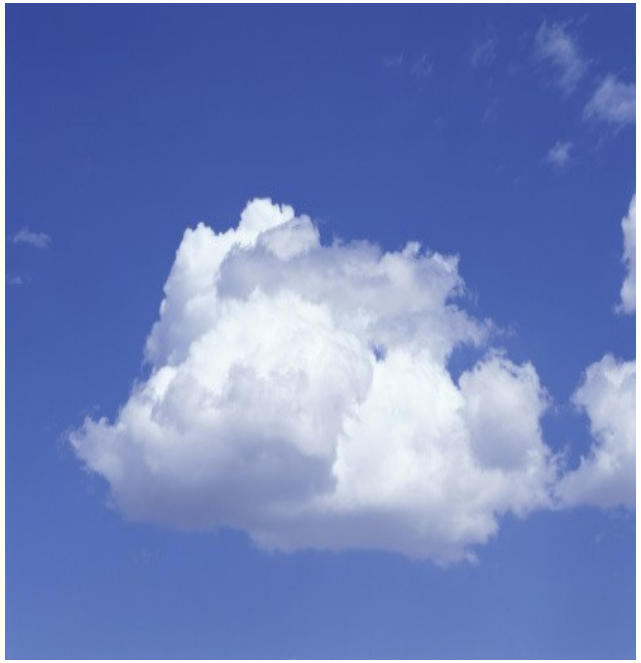
- Volume Rendering Introduction
- Motivation
- Data Model
- Render Pipeline
  - Fog
  - Clouds
  - Water
- Conclusion

Absorption, Scattering and Emission

Absorption

σ_a = absorption coefficient

Scattering

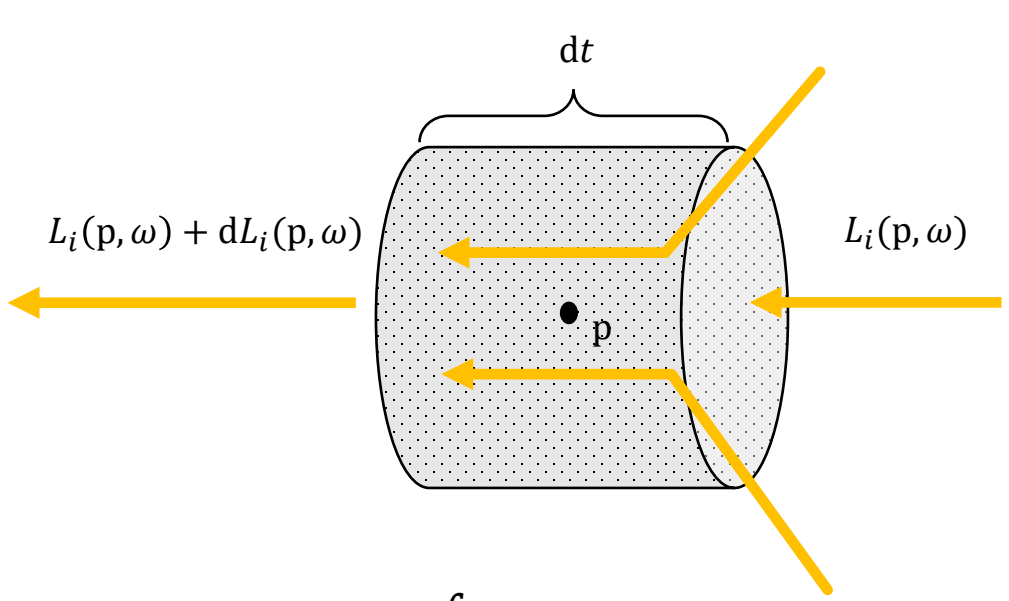


σ_s = scattering coefficient

Emission

Scattering

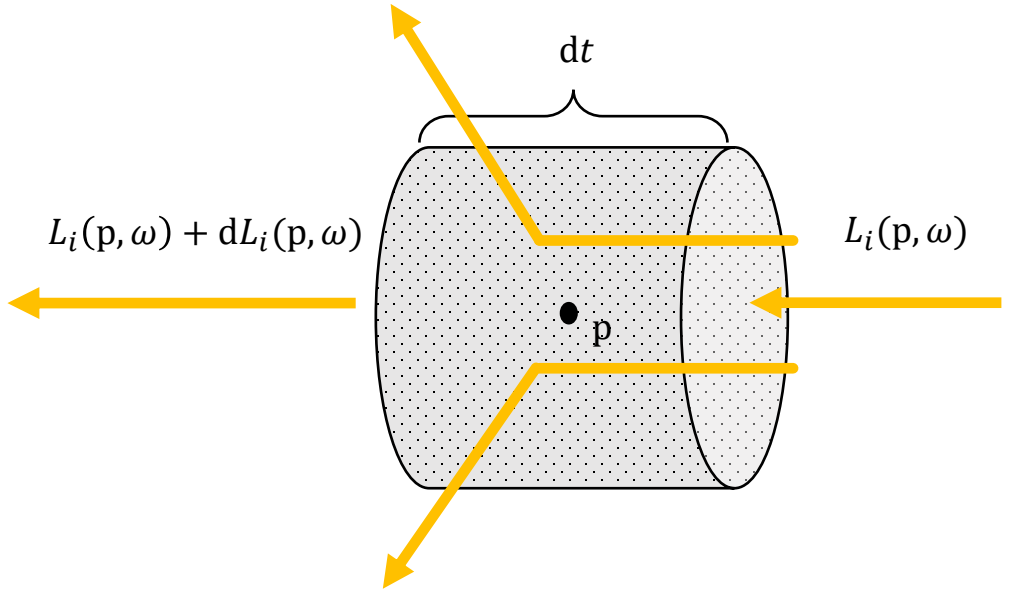
In-scattering



$$\mathrm{d}L_i(\mathrm{p},\omega) = \sigma_s(\mathrm{p},\omega)\int_{S^2} f_p(\mathrm{p},\omega,\omega')L_i(\mathrm{p},\omega')\mathrm{d}\omega'\,\mathrm{d}t$$

$f_p(\mathrm{p},\omega,\omega')$ = phase function

Out-scattering



$$\mathrm{d}L_i(\mathrm{p},\omega) = -\sigma_s(\mathrm{p},\omega)L_i(\mathrm{p},\omega)\mathrm{d}t$$

Phase Function

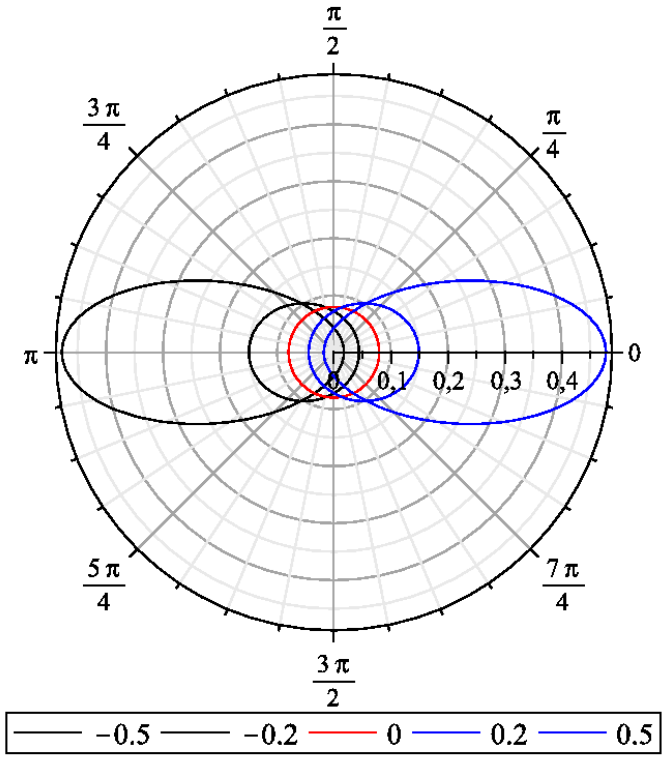
- Isotropic phase function
  Light is scattered equally in all directions

$$P(\theta) = \frac{1}{4\pi}$$

- Henyey-Greenstein (HG) phase function
  Approximation of the Mie phase function

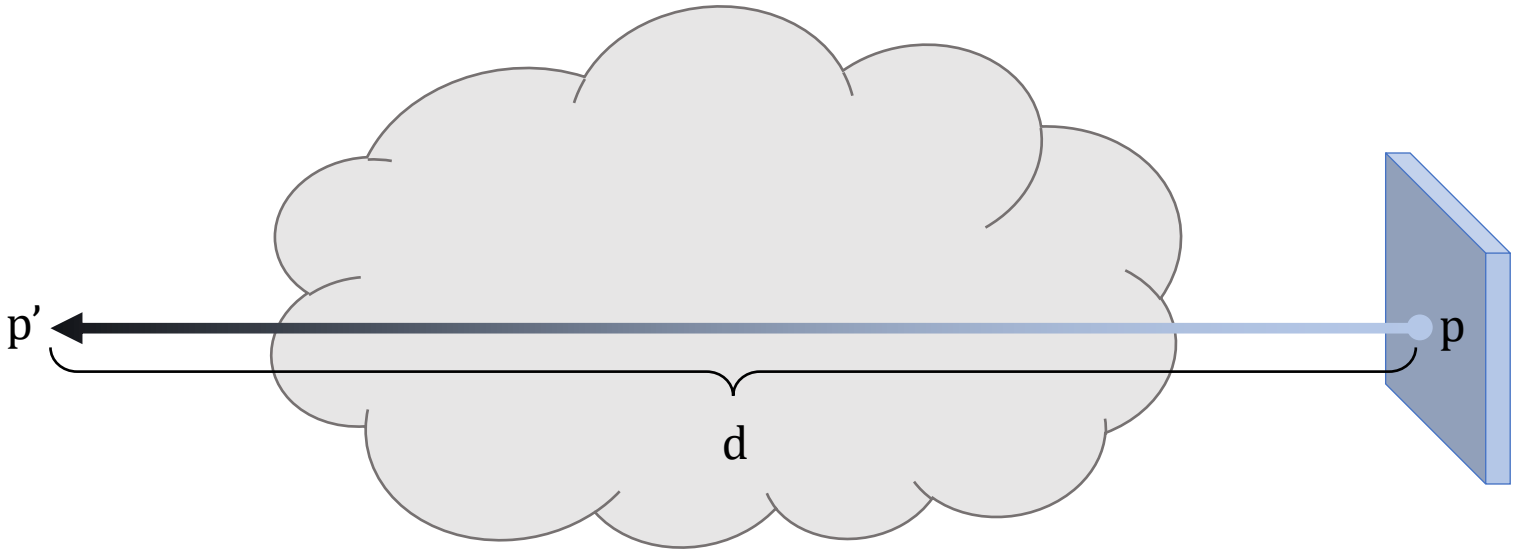
$$P(\theta) = \frac{1 - g^2}{4\pi(1 + g_2 - 2g\cos\theta)^{1.5}}$$

g = scattering angle anisotropy parameter in [-1,1]



Plot of Henyey-Greenstein phase function with different g values

Extinction and Transmittance



$$\sigma_t = \sigma_a + \sigma_s$$

$$\mathcal{T}(\mathrm{p} \rightarrow \mathrm{p}') \equiv \exp\left(-\int_0^d \sigma_\mathrm{t}(\mathrm{p} + \omega t, \omega)\, \mathrm{d}t\right)$$

Volume Rendering Equation

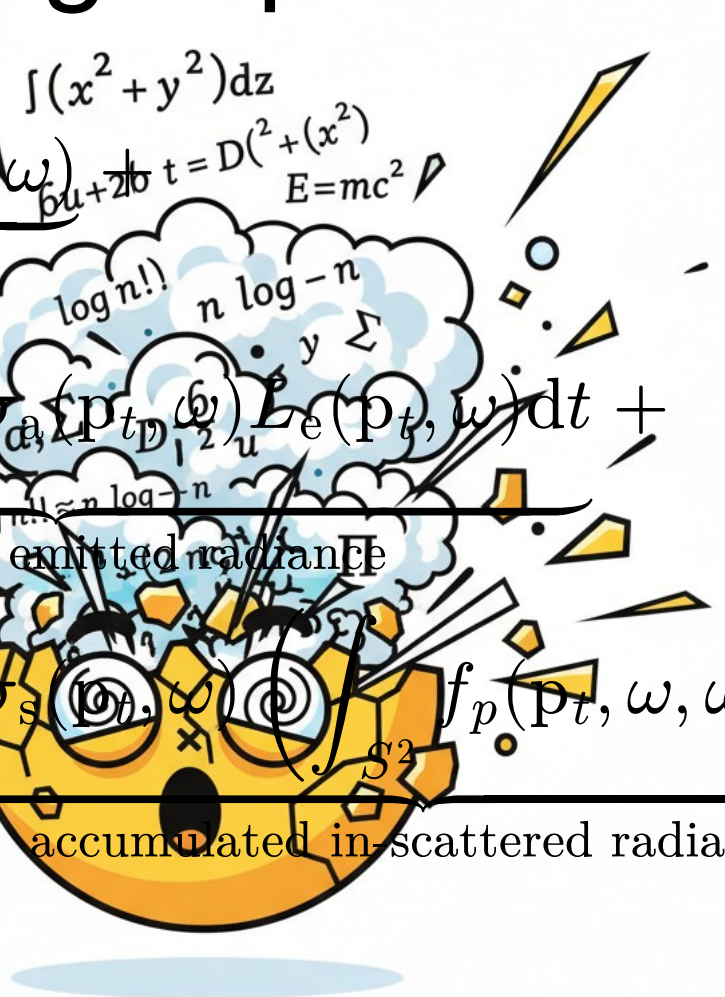
$$L(\mathrm{p}', \omega) = \underbrace{\mathcal{T}(\mathrm{p} \to \mathrm{p}')L(\mathrm{p}, \omega)}_{\text{reduced radiance}} +$$

$$\underbrace{\int_0^d \mathcal{T}(\mathrm{p} \to \mathrm{p}_t)\sigma_\mathrm{a}(\mathrm{p}_t, \omega)L_\mathrm{e}(\mathrm{p}_t, \omega)\mathrm{d}t}_{\text{accumulated emitted radiance}} +$$

$$\underbrace{\int_0^d \mathcal{T}(\mathrm{p} \to \mathrm{p}_t)\sigma_\mathrm{s}(\mathrm{p}_t, \omega)\left(\int_{S^2} f_p(\mathrm{p}_t, \omega, \omega')L(\mathrm{p}_t, \omega')\mathrm{d}\omega'\right)\mathrm{d}t}_{\text{accumulated in-scattered radiance}}$$

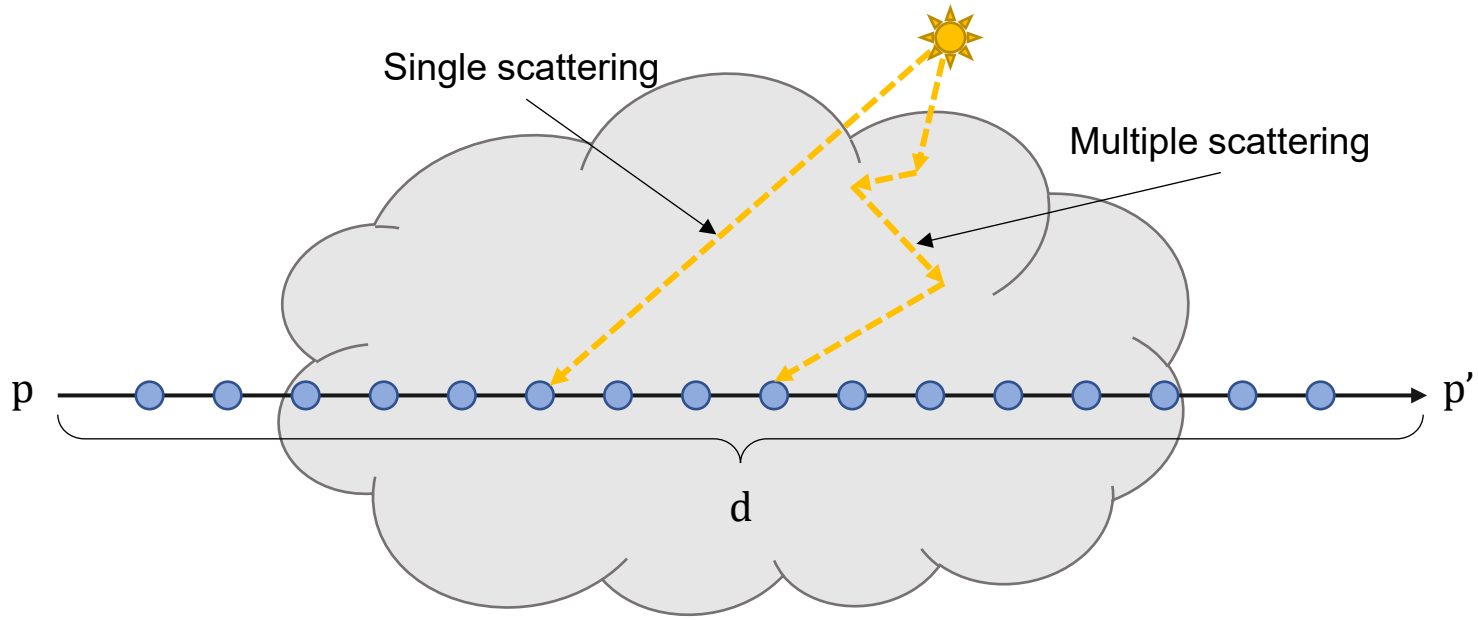
Volume Rendering Equation

$$L(\mathrm{p}', \omega) = \underbrace{\mathcal{T}(\mathrm{p} \to \mathrm{p}')L(\mathrm{p}, \omega)}_{\text{reduced radiance}} + \underbrace{\int_0^d \mathcal{T}(\mathrm{p} \to \mathrm{p}_t)\sigma_a(\mathrm{p}_t, \omega)L_e(\mathrm{p}_t, \omega)\mathrm{d}t}_{\text{accumulated emitted radiance}} + \underbrace{\int_0^d \mathcal{T}(\mathrm{p} \to \mathrm{p}_t)\sigma_s(\mathrm{p}_t, \omega)\left(\int_{S^2} f_p(\mathrm{p}_t, \omega, \omega')L(\mathrm{p}_t, \omega')\mathrm{d}\omega'\right)\mathrm{d}t}_{\text{accumulated in-scattered radiance}}$$



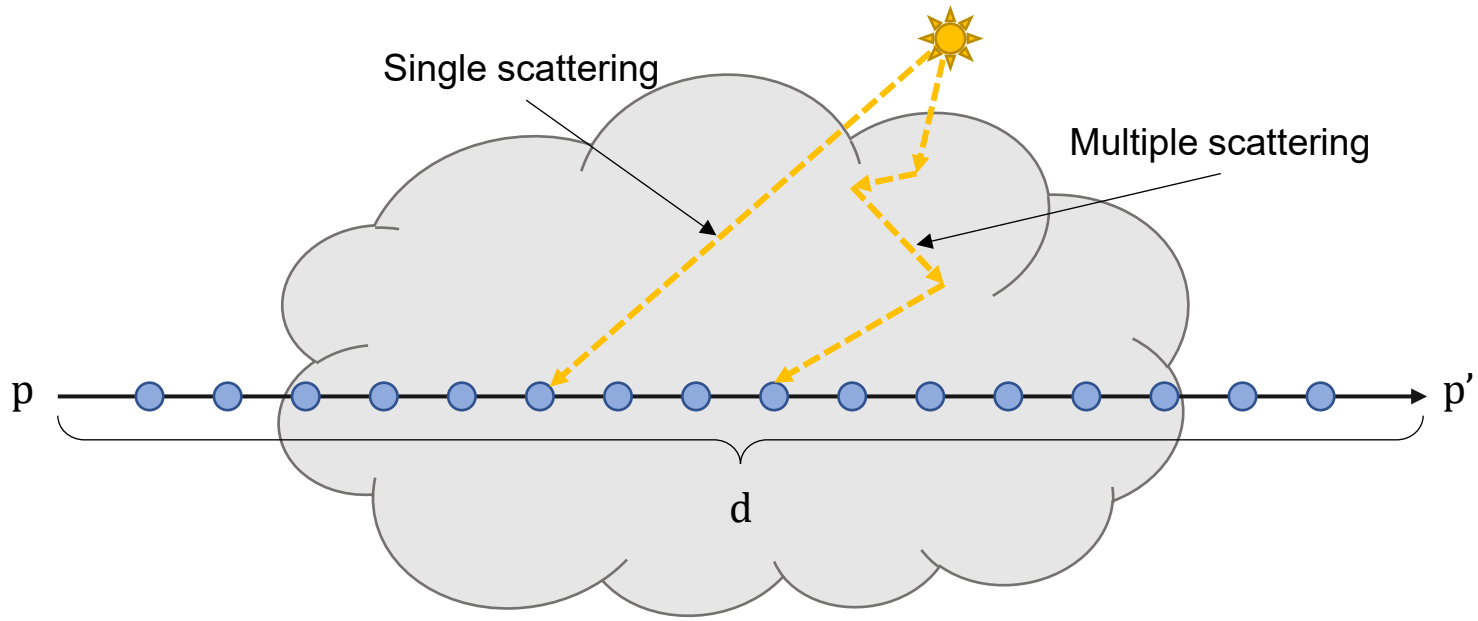
Volume Rendering

- A lot of integrals
- No analytical solution for inhomogeneous media
- Approximation by discrete sampling over distance d from p to p’



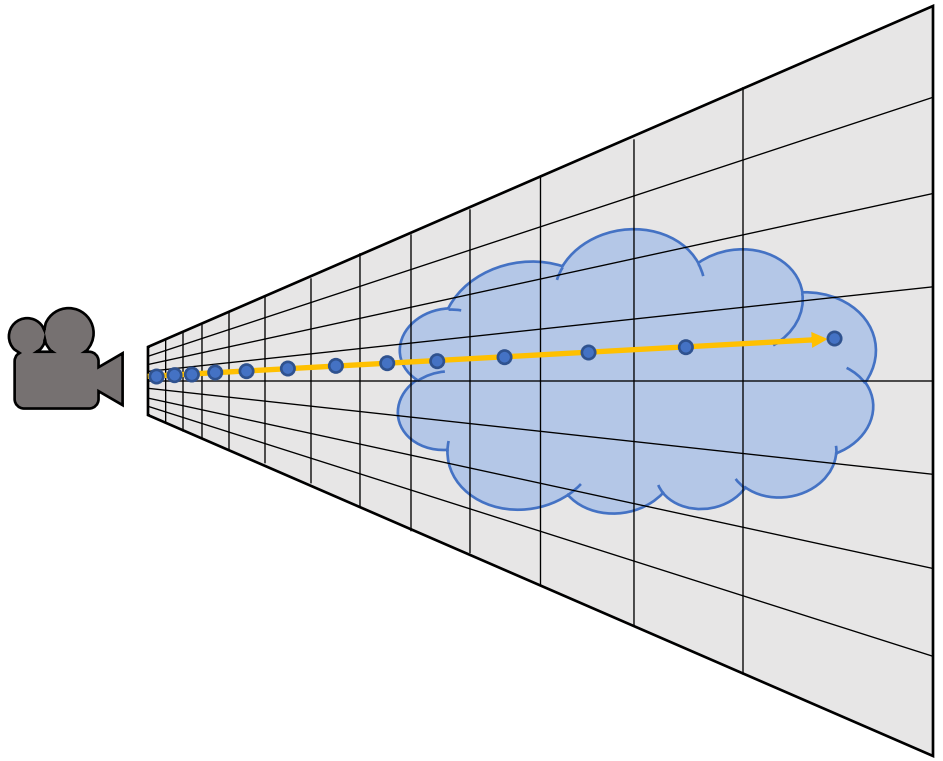
Ray marching

- Single loop per ray
- Accumulate transmittance and in-scatter
- Use energy conservative integration from [Hillaire15]



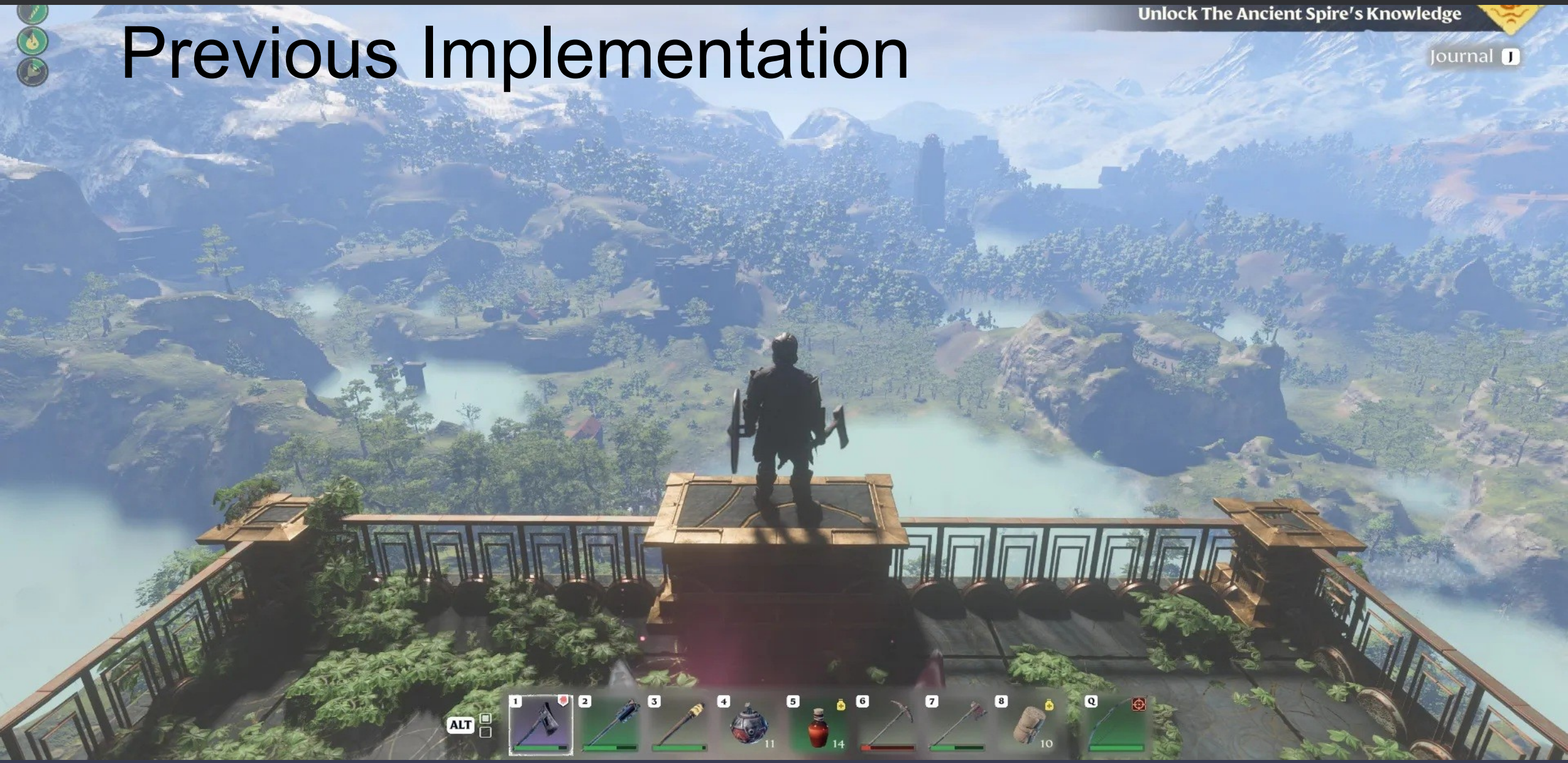
Froxel Volumes

- Froxel -> View frustum voxel [Wronski14] [Hillaire15]
- Clip space 3D textures as cache for media and lighting
- Low resolution e.g. 160x90x64
- Compute media properties and lighting for each froxel in parallel
- Integrate froxels along depth and store result per froxel



Motivation

Previous Implementation



Previous Implementation

- Previous implementation presented at GPC 2024 [Feller24]
- Supported different volumetric medias
- Froxels for near-field
- Ray-marching for far-field
- Composed clouds and precomputed atmosphere on top

Problems with Previous Implementation

- Missing flexibility
- Quality did not match our expectations

Design Goals

- Increase flexibility
- Unified solution for different media including clouds
  - Proved difficult due to different visual requirements and scales
  - Dropped after first iteration
- Realistic and detailed lighting
- Stable under motion
- Physically based but with some artistic freedom
- Water

Voxel Fog - Models and Instances

Voxel Fog - Models and Instances

- Fog models
  - Artist driven high resolution voxel models
  - Down sampled to 16 meter voxels
  - Stored as sparse 16^3 tiles with half voxel border on each size
  - Store signed distance and density
  - Block compressed (BC4 density, BC1 distance scalar encoding [Schneider23])
  - Tiles with uniform density stored as single value

Voxel Fog - Models and Instances

- Fog model tile atlas
  - Runtime cache of required tiles
  - Signed distance texture (BC1)
  - Density texture (BC4)

- Instances could be placed freely within world boundaries
  - Reference to single fog model
  - Media material

Voxel Fog - World Volume

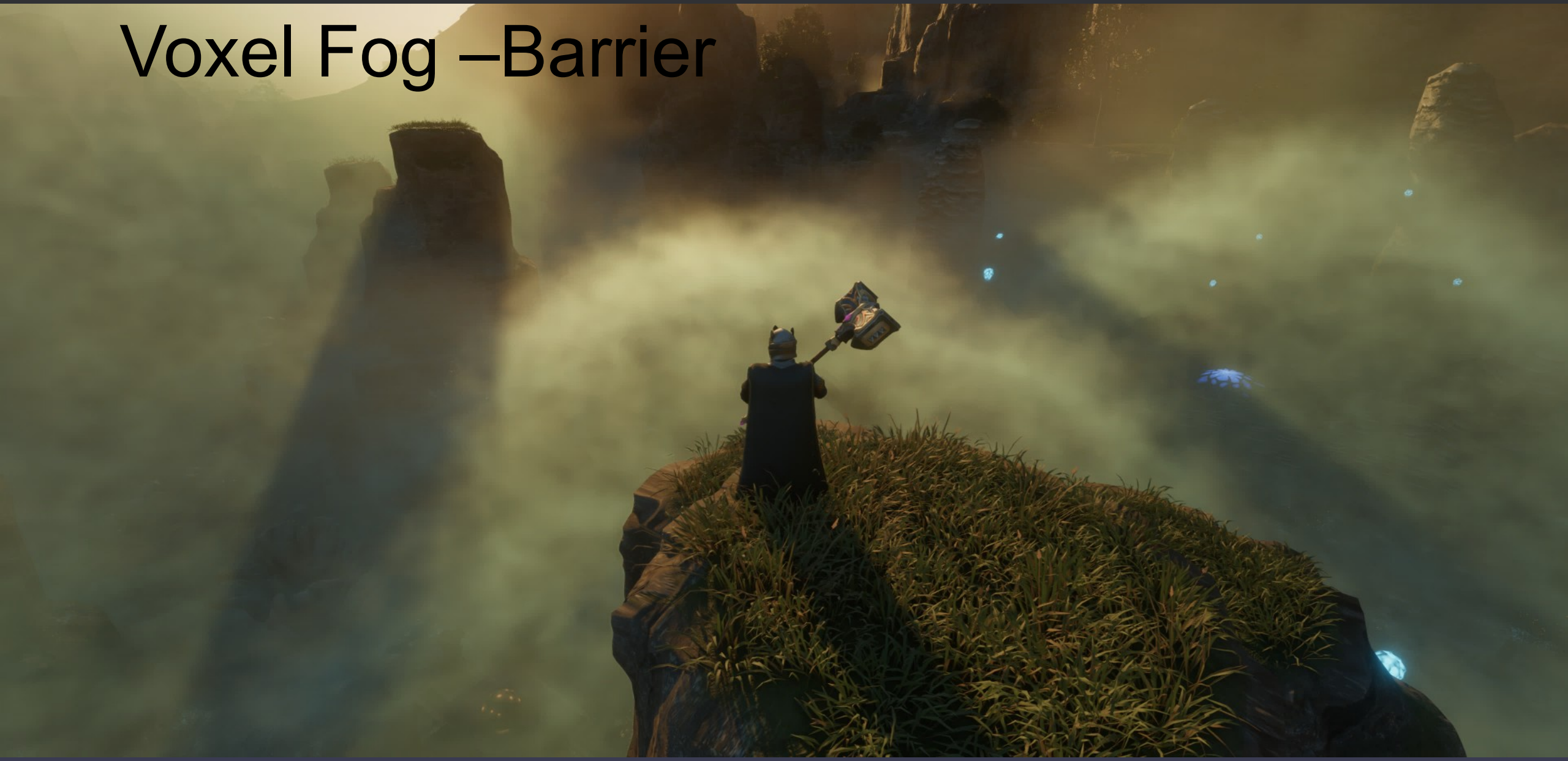
Voxel Fog - World Volume

- Covers the entire playable area
- Tile atlas with 16^3 voxels per tile
  - Half voxel border on each size for seamless trilinear filtering
  - Effective size of 15^3 voxels
- Multiple block-compressed layers
  - BC1 distance field (Bc1 scalar encoding from [Schneider23])
  - BC5 density and extinction
  - BC3 scatter albedo and detail noise type
- Content updated on the GPU
  - Compute shader with runtime block compression

Voxel Fog - World Volume

- Resolution is 43 x 13 x 43 cells
  - Each cell covers 240^3 meter in the game world
- R16_uint texture format
  - 15 bit tile address
  - 1 bit empty flag
- Managed on the CPU
  - Indirection texture is updated once the frame
  - Allocate and free tiles during indirection texture update
  - Schedule GPU update for dirty tiles
- Reuse identical uniform tiles

Voxel Fog –Barrier



Voxel Fog – Barrier

- Limited to 64 meters around camera
- 1 meter resolution
- 128x128x128 signed distance field texture
- R16G16_snorm
  - R: Distance to dangerous fog interface
  - G: Distance to deadly fog interface
- Runtime generated on the GPU from fog and scene voxel data
  - Fast Hierarchical 3D Distance Transforms on the GPU [Cuntz07]
  - Followed by single jump flood pass
- As a bonus we get ground fog in the shroud for free

Injected Fog Volumes

- Spawned by VFX
- Box or sphere shape
- Optional density texture
- The only emissive media in pipeline
- See [Feller24] for more details

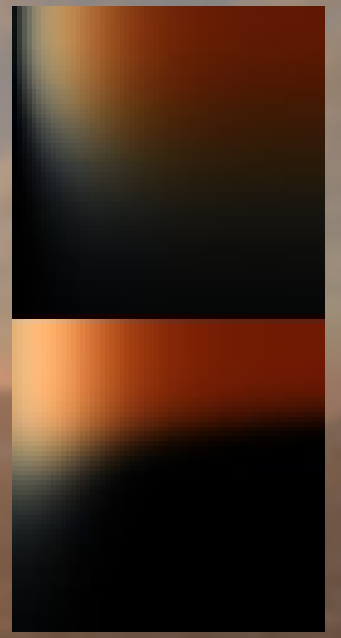
```
struct VolumetricFogInjectInstanceData
{
    uint shape;
    float3 position;
    float4 rotation;
    float3 size;
    uint densityTexture;
    float3 uvOffset;
    float3 uvScale;
    float density;
    float falloff;
    float3 emission;
    float extinction;
    float3 scattering;
};
```

Atmosphere and Weather

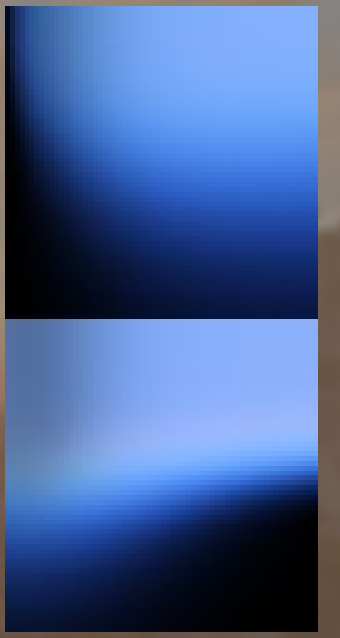
Atmosphere

- Precomputed sky transmittance
  - 2D LUT parametrized by height and zenith angle
  - Needed to determine sun and moon color through atmosphere
  - Used in all sun and moon lighting calculations

- Precomputed sky atmosphere [Bruneton08]
  - 3D LUTs ignore earth shadow [Elek09]
  - Multiple scattering and Ozone
  - Improved LUT parameterization [Elek09]
  - Additional LUT for cloud ambient lighting



Single slice from Mie LUT



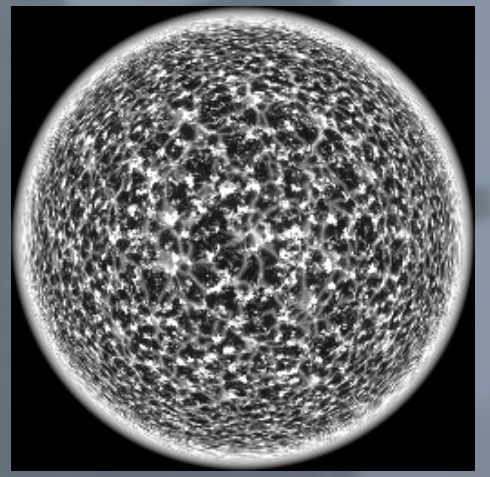
Single slice from Rayleigh LUT

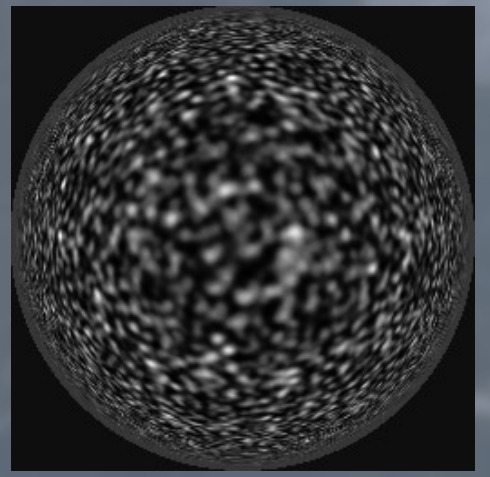
Aerial Perspective and Height Fog

- Aerial perspective
  - Same math and coefficients as used for atmosphere LUT computation
  - No multiple scattering
- Analytic height fog
  - Exponential height falloff
  - Density and albedo

Clouds

- Rendered at runtime by weather system
  - Multiple cloud instances rendered into cloud map
  - Artist generated density and height textures
  - Tiled and non tiled clouds
  - Spherical distortion to simulate curvature of earth
- R16G16 cloud density and height
- Top down projected 1024x1024 texture
- Covers 80x80 km around world center





Rain and Snow Fog

- Fog below clouds where it rains or snows
- 512x512 texture with R16G16
  - R: Density
  - G: Max height
- Top down projected
- Covers the entire playable area
- Runtime generated
- Derived from weather systems cloud and rain map
- Top down variance shadow map to avoid fog in buildings or caves

Water

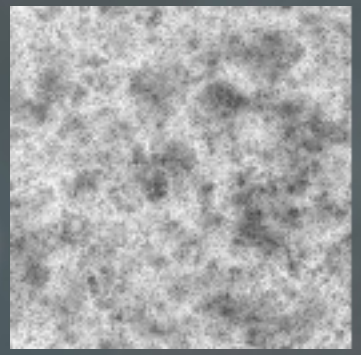
Water

- Sparse signed distance field
  - Fully dynamic
  - Generated from water simulation
- First view ray intersection with water surface rendered into screen space texture
- More in GPC 2025 talk “Water Simulation & Rendering in Enshrouded” from Simon Stempfle and Andreas Mantler

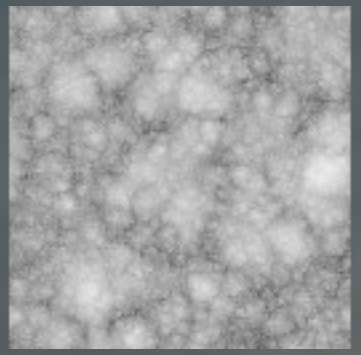
Detail Noise

- Adds detail to voxel fog and cloud ray marching
- Based on Nubis Cubed [Schneider23]
- Two noise types
  - Curly-Alligator noise
  - Alligator noise
- Configurable in editor and generated by asset pipeline
- Tileable
- 128x128x128 four channels uncompressed

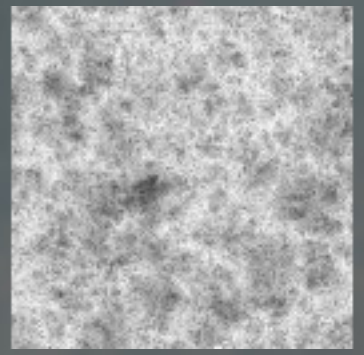
Curly-Alligator

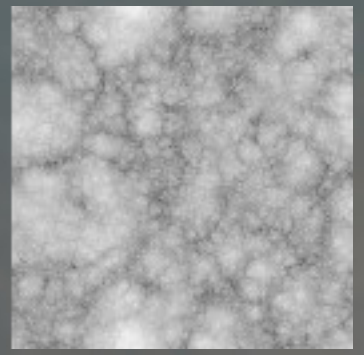


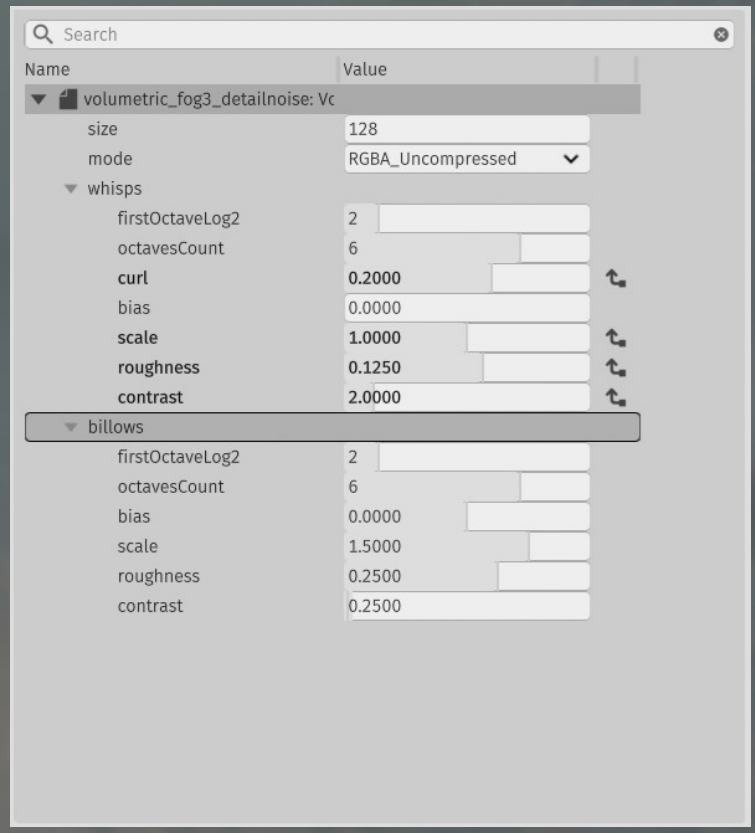
Alligator



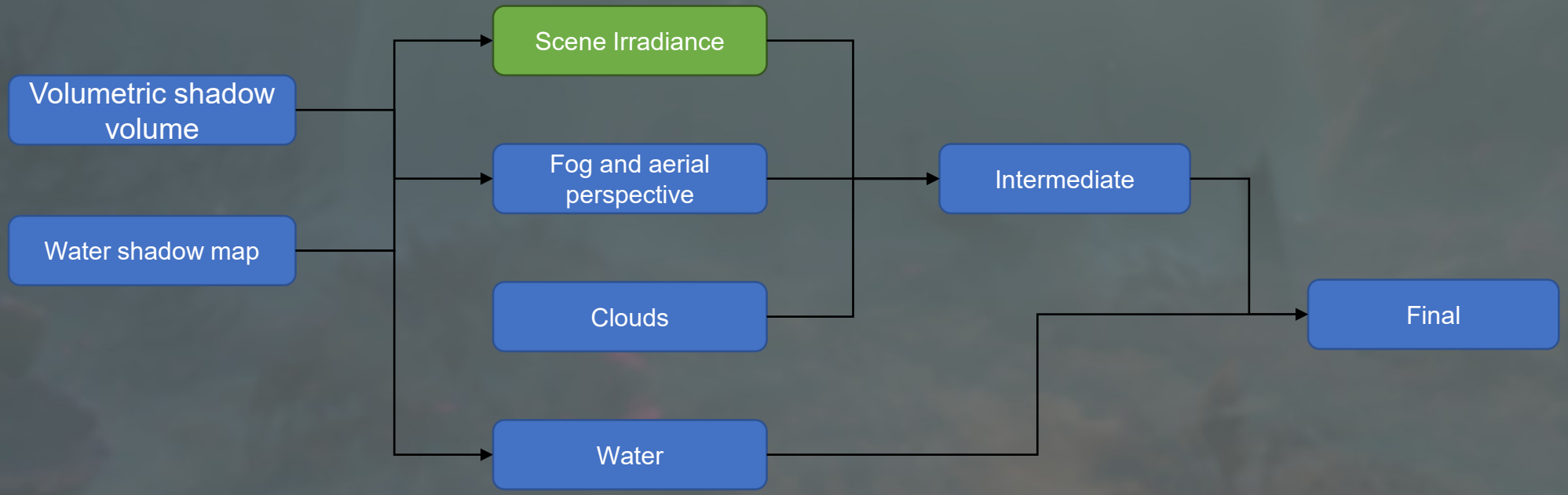
Variations



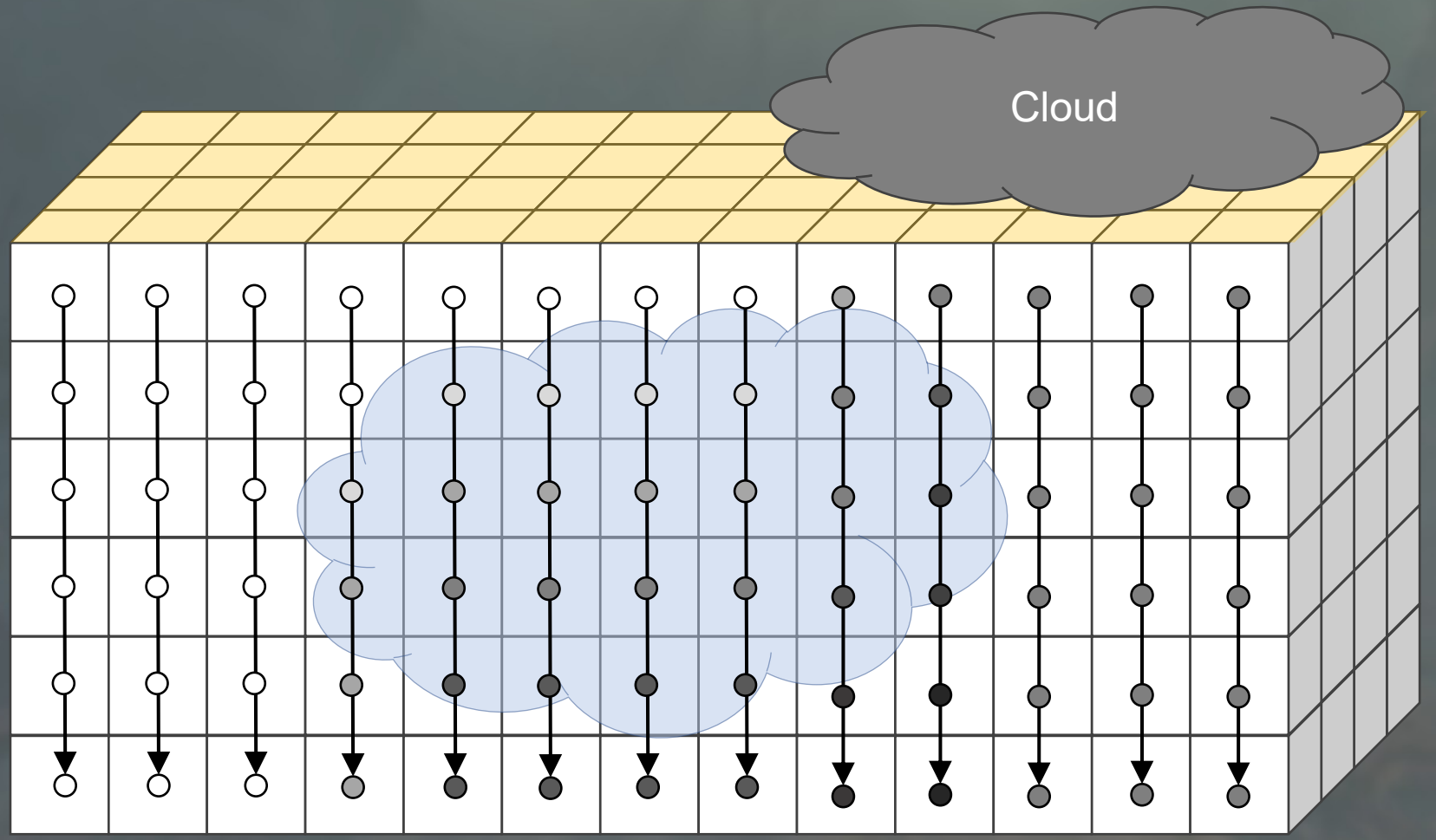




Render Pipeline



Volumetric shadow Volume

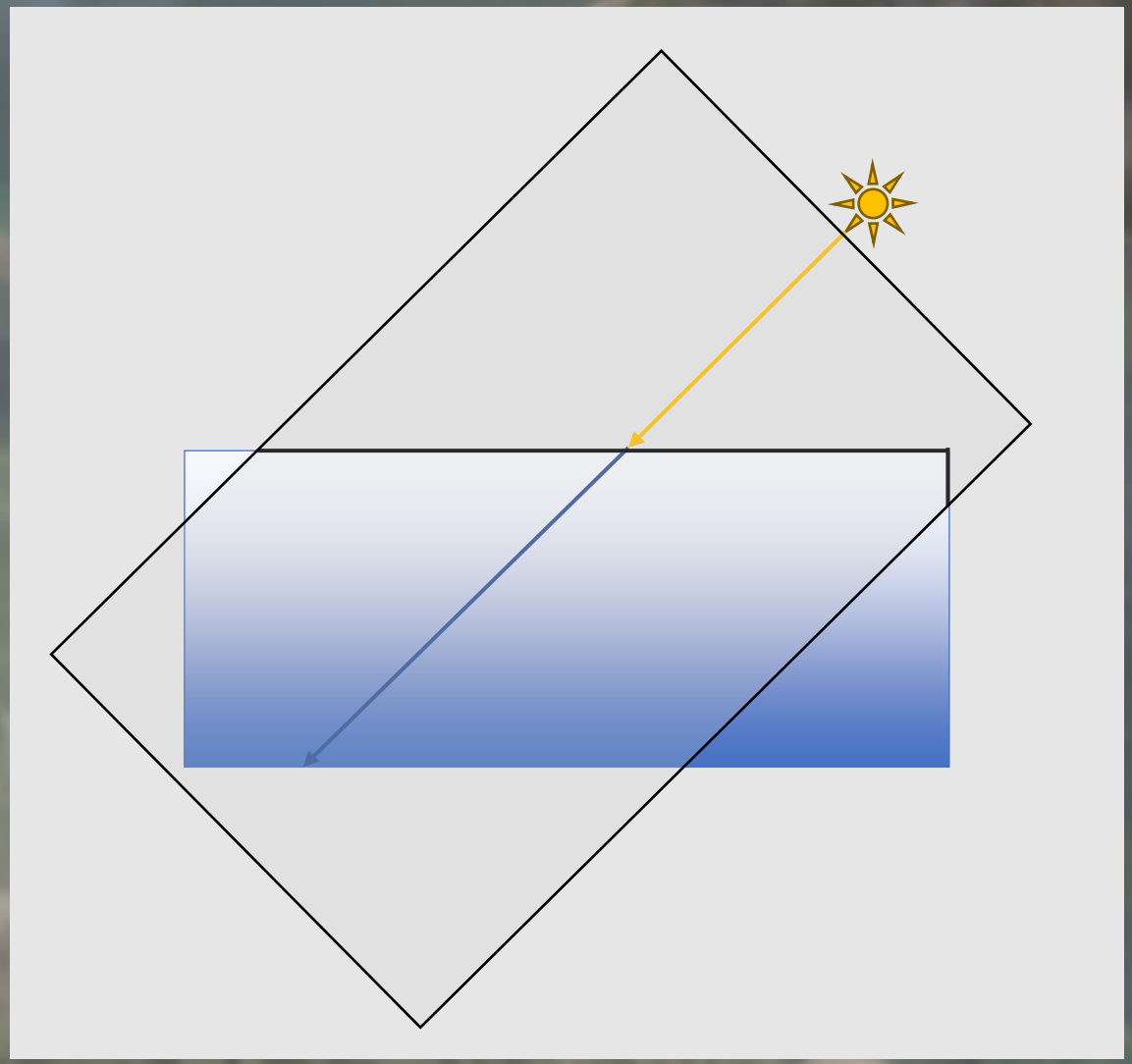


Volumetric shadow Volume – Sweep

```
float accumulatedTransimttance = sampleCloudShadows( texCoord );
for( uint i = 0u; i < sweepCount; ++i )
{
   if( any( saturate( texCoord ) != texCoord ) )
   {
      // restart ray
      texCoord = frac( texCoord );
      accumulatedTransimttance = sampleCloudShadows( texCoord );
   }
   // sample extinction and accumulate transmittance
   float extinction = sampleExtinction( texCoord );
   accumulatedTransimttance *= exp( dt * -extinction );
   // temporal accumulate result into voxel
   const uint3 pos = floor( texCoord * g_constants.outputSize );
   g_output[ pos ] = lerp( g_output[ pos ], accumulatedTransimttance, 1.0f / 64.0f );
   // next step
   texCoord += texCoordRayDir;
}
```

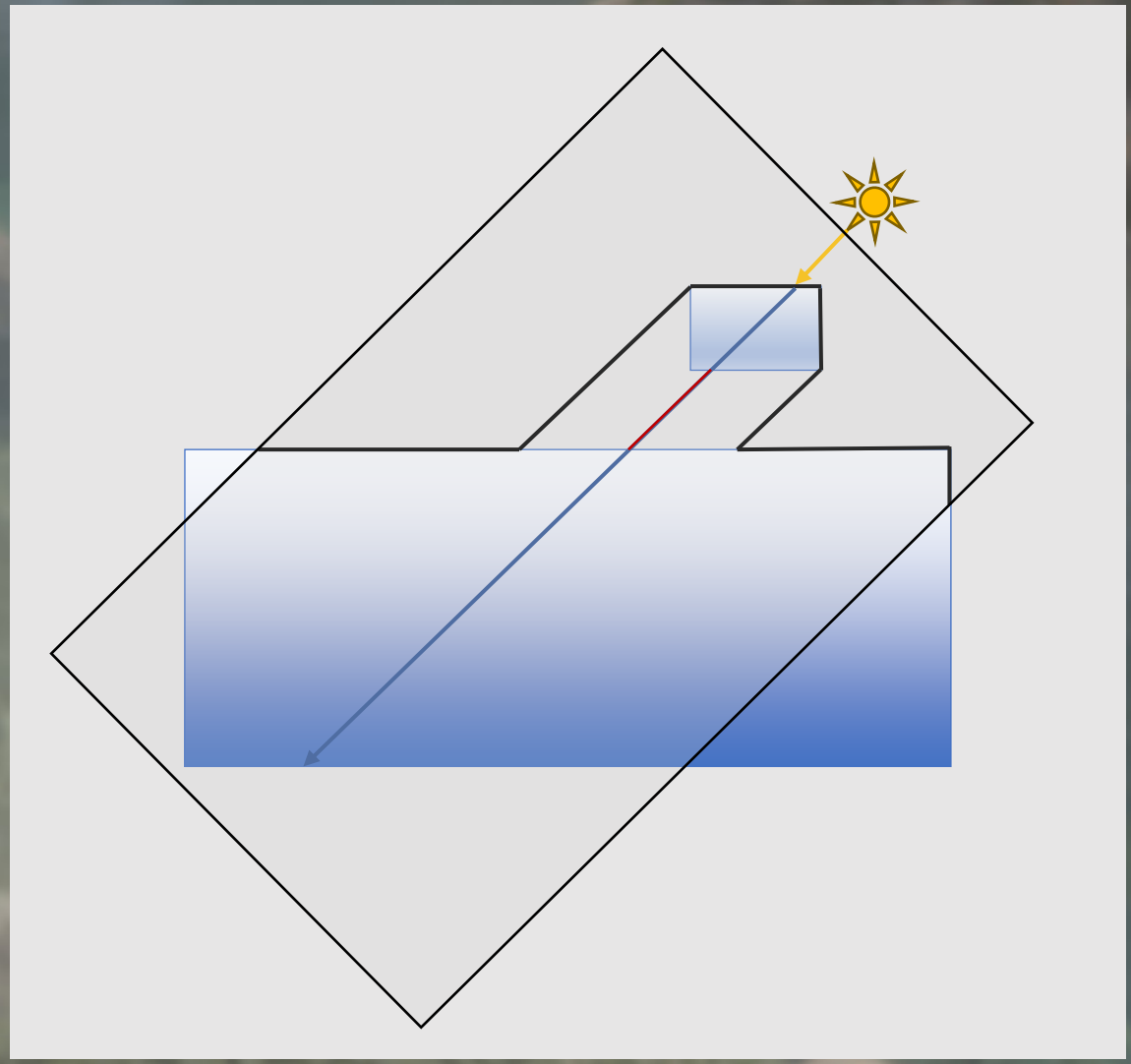
Water shadow map

- Approximate water depth and transmittance for directional light
- Render water depth map in light space
- Camera centered clip map
- Snap to pixel position to avoid flickering under movement
- Used for
  - Direct lighting
  - Underwater volume
  - Global illumination



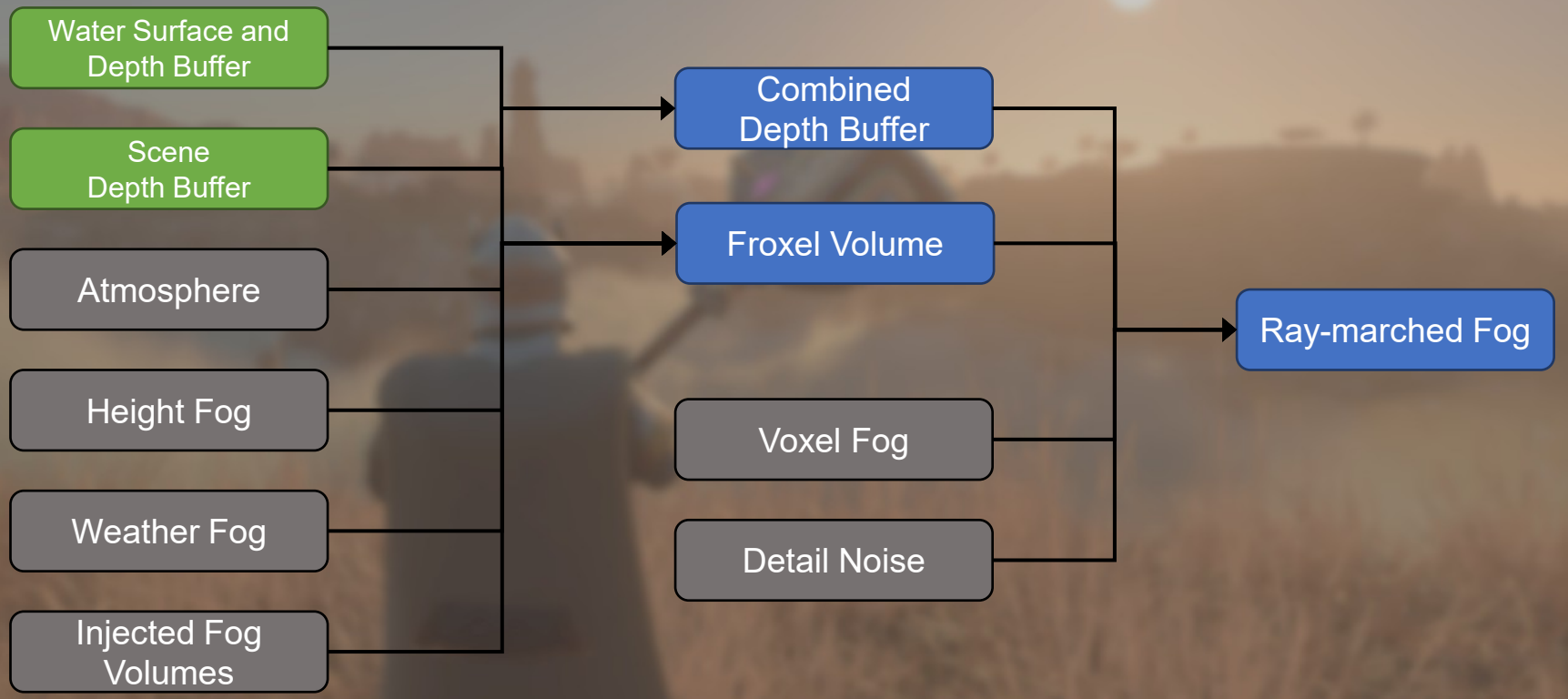
Water shadow map

- Overestimates transmittance in case of overlapping water surfaces
- Not a big problem in practice because water is mostly flat and surrounded by geometry
- Only apply water shadows to underwater geometry



Fog

Fog Pipeline

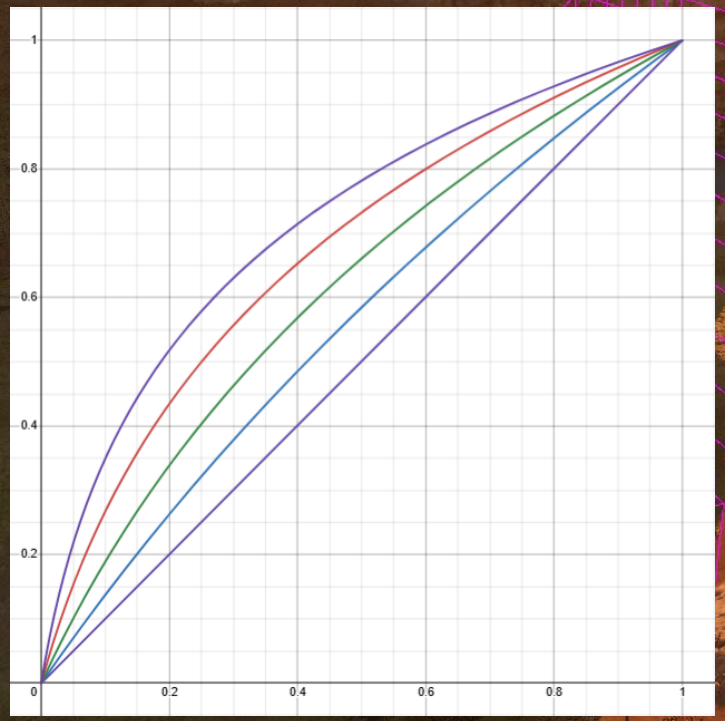


Froxel Volume

- Contains aerial perspective and all fog types other than voxel fog
- Resolution depends on screen size and quality setting
- Covers full view depth range up to 10km
- Upper depth limit for all computations based on scene depth buffer

Froxel Volume – Depth Distribution

$$\frac{\log\left(\frac{x-n}{b}+1\right)}{\log(a+1)}, a = kf, b = \frac{f-n}{a}, n = near, f = far, \mathrm{x} = \{\, \mathrm{n} \leq \mathrm{x} \leq \mathrm{f} \,\}, k = \{\, 0 < k \,\}$$



Froxel Volume - Froxel Depth

- Down-sample scene depth buffer
  - Max depth
- Dilate
  - 3x3 max depth kernel
  - Linear filtering artifacts without dilation

Froxel Volume - Populate

- Output scatter and monochrome extinction coefficients texture
  - No atmosphere
- Output radiance texture
  - Point lights, global illumination and emissive from Injected fog volumes
- Output directional light visibility texture
  - Primary directional light (sun or moon)
- Temporal accumulate with previous frame

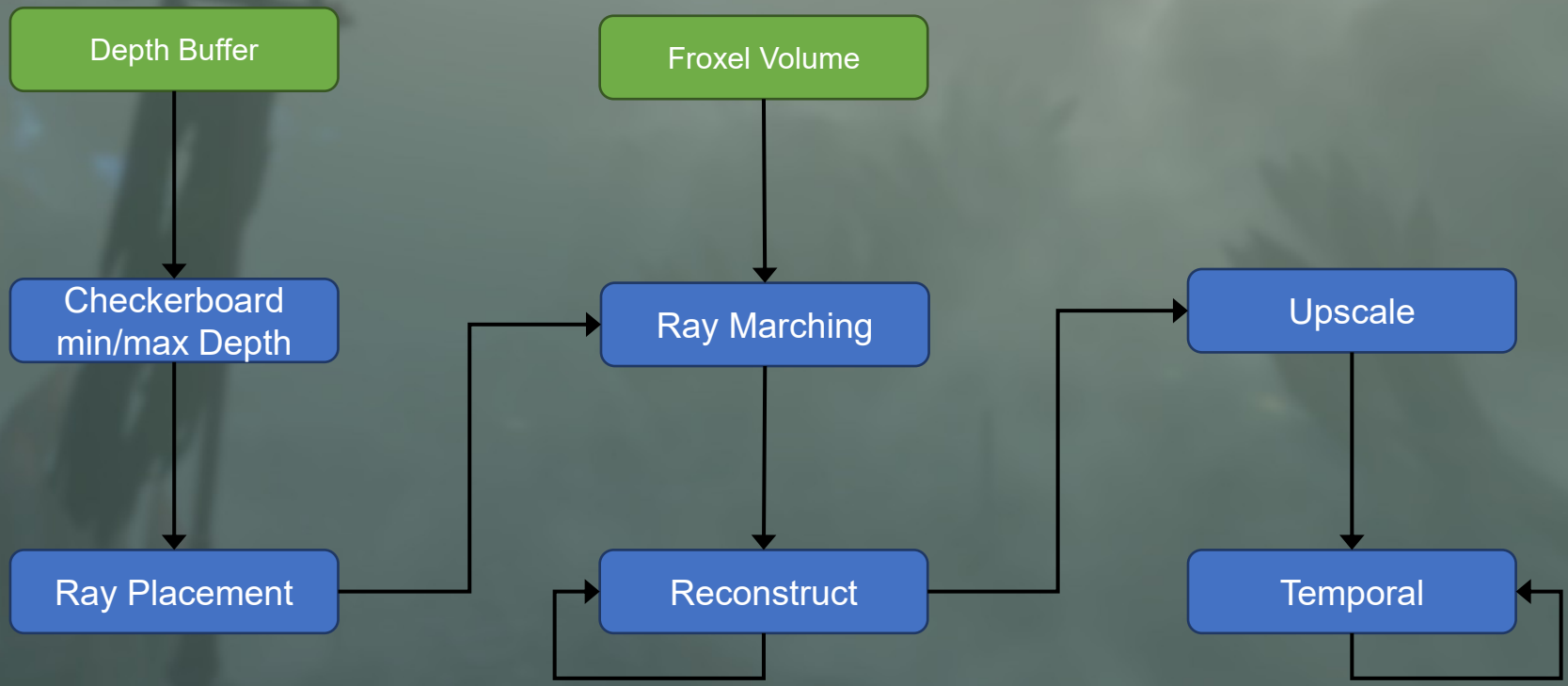
Froxel Volume - In-scatter & Extinction

- Output in-scatter texture
  - Primary directional light
  - Point light, GI and emissive radiance from populate pass
- Output extinction texture
  - Sum extinction from populate pass with atmosphere Mie and Rayleigh extinction
  - No longer monochrome
- **No** temporal accumulation,
  - Phase function is view angle dependent and leads to smearing

Froxel Volume - Integrate

```
float3 accumulatedTransmittance = 1.0f;
float3 accumulatedRadiance = 0.0f;
// compute integration slice count and linear start linear depth
const uint integrationSliceCount = (uint)ceil( froxelDepthTexture[ pixelPos ] * sliceCount );
float prevLinearDepth = computeLinearDepthFromFroxelDepth( 0.0f, froxelDepthToLinearDepth );
// loop over all depth slices
for ( int i = 0; i < integrationSliceCount; ++i )
{
    const uint3 froxelPos = uint3( pixelPos, i );
    // compute integration step length
    const float linearDepth = computeLinearDepthFromFroxelDepth( ( i + 1.0f ) * invSliceCount,
    froxelDepthToLinearDepth );
    const float stepLength = ( linearDepth - prevLinearDepth ) * linearDepthToDistance;
    prevLinearDepth = linearDepth;
    // load in-scatter and extinction
    const float3 inScatter = outputScatterTexture[ froxelPos ];
    const float3 extinction = outputTransmittanceTexture[ froxelPos ];
    // integrate transmittance and radiance
    const float3 transmittance = exp( -extinction * stepLength );
    accumulatedRadiance += inScatter * ( 1.0f - transmittance ) * accumulatedTransmittance;
    accumulatedTransmittance *= transmittance;
    // store transmittance and radiance
    outputScatterTexture[ froxelPos ] = accumulatedRadiance;
    outputTransmittanceTexture[ froxelPos ] = accumulatedTransmittance;
}
```

Ray marching - Pipeline



Ray marching – Ray Reconstruction

- Mostly follows [Bauer19]
- Ray march 1 of 4 rays in half resolution
- Reconstruct half resolution
  - Reproject missing rays from previous frame
  - Clamp with neighborhood AABB
  - Weight neighborhood pixels by depth difference
  - Depth weighted bilateral filtering of history
  - Use weighted average of neighborhood if reprojection fails

Ray marching – Ray placement

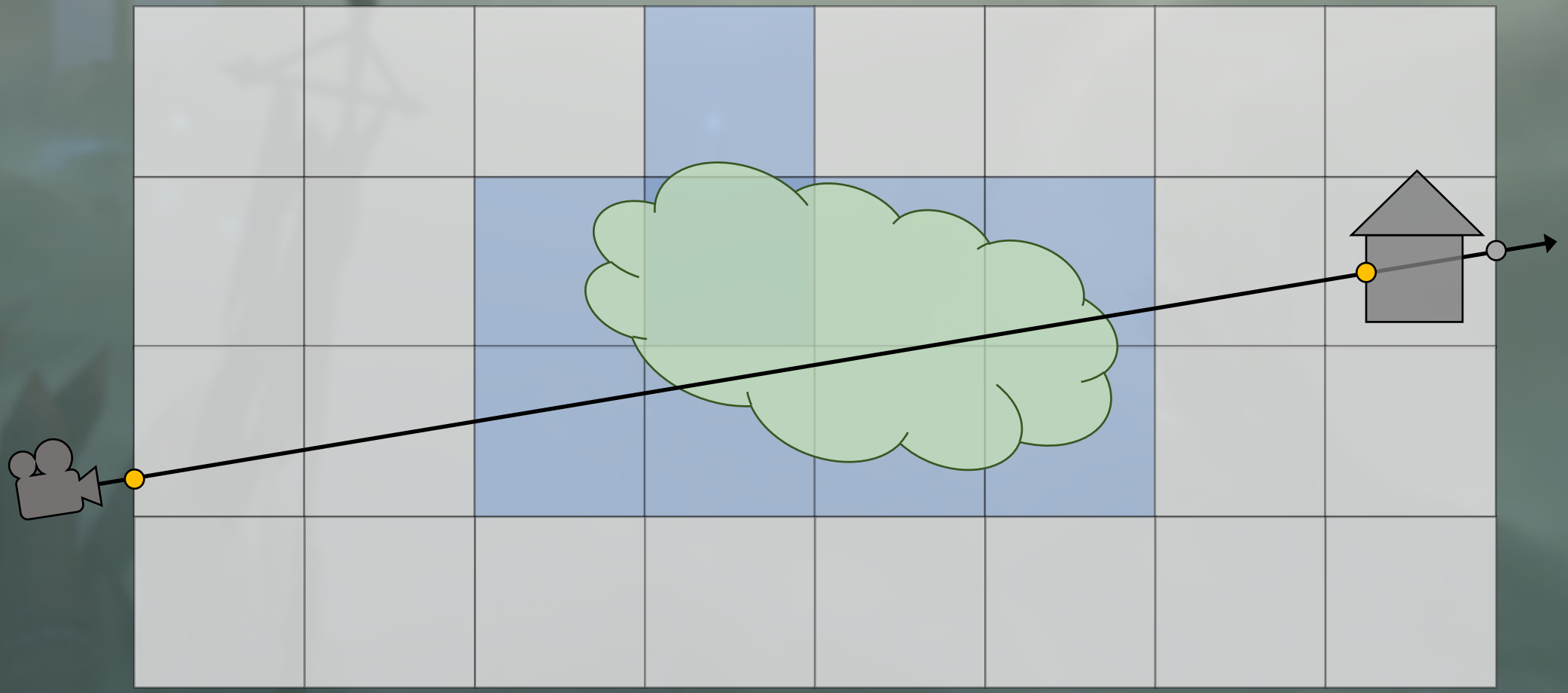
- Half res min/max checkerboard depth
- Ray placement in 2x2 tile follows [Bauer19]
- Fixup ray placement in isolated depth cases
- Outputs R8 sample index in 2x2 tile
- Outputs R16 linear depth of sample in 2x2 tile

Ray marching - Upscale & Temporal

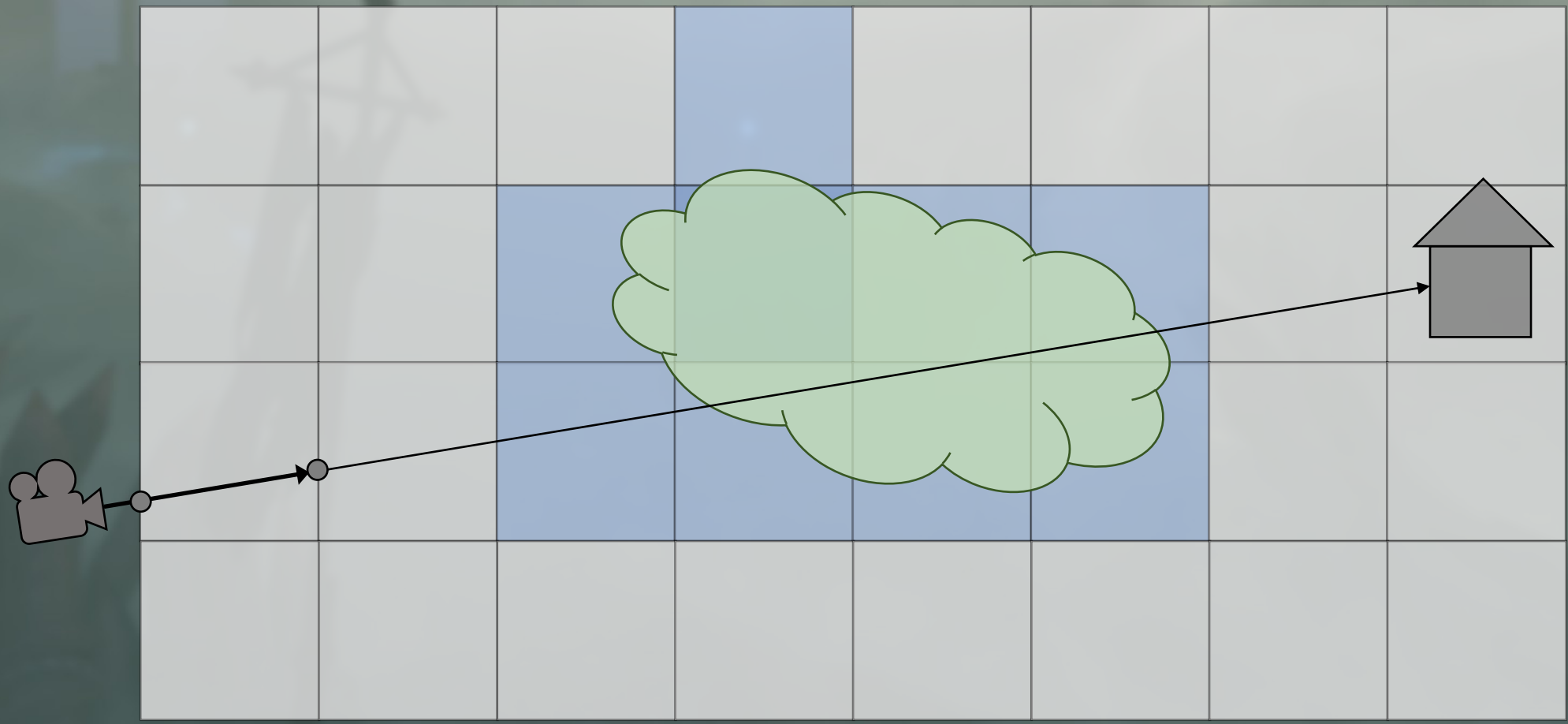
- Dithered upscale to full res
  - Four half res samples
  - Spatio-temporal blue noise offset
  - Scale kernel size by distance
  - Weighted by depth difference

- Temporal filter
  - Use transmittance weighted depth for reprojection
  - Depth weighted bilateral filtering of history
  - Clamp with neighborhoods AABB

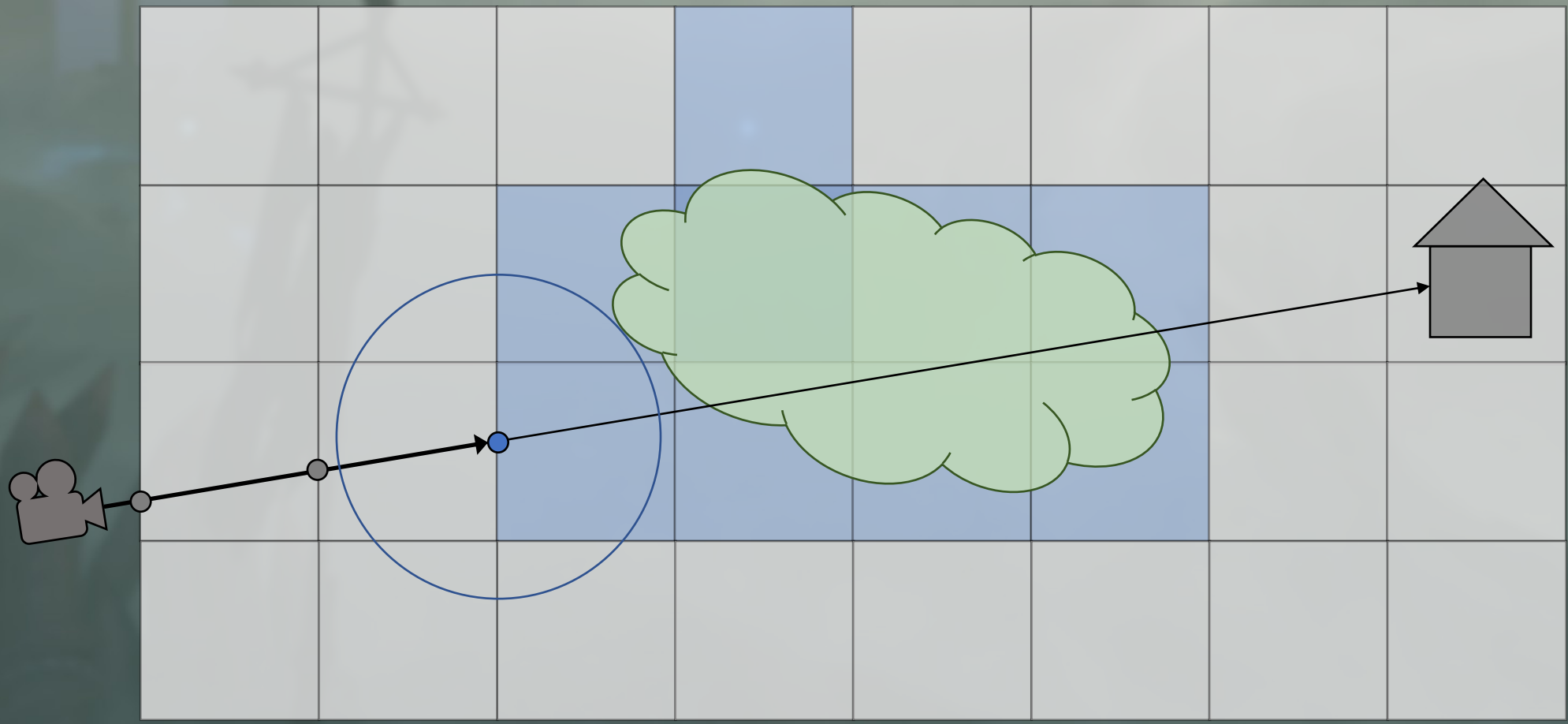
Ray marching



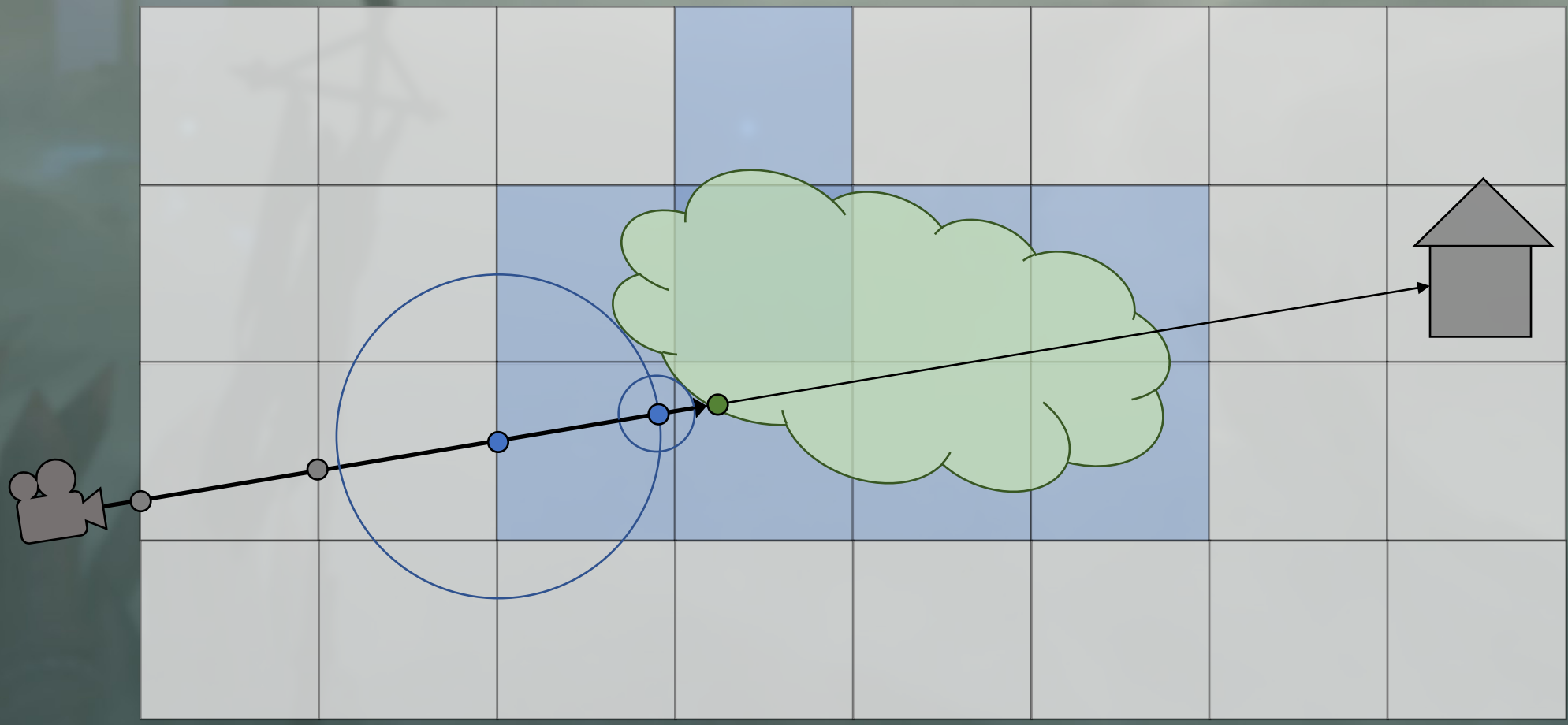
Ray marching



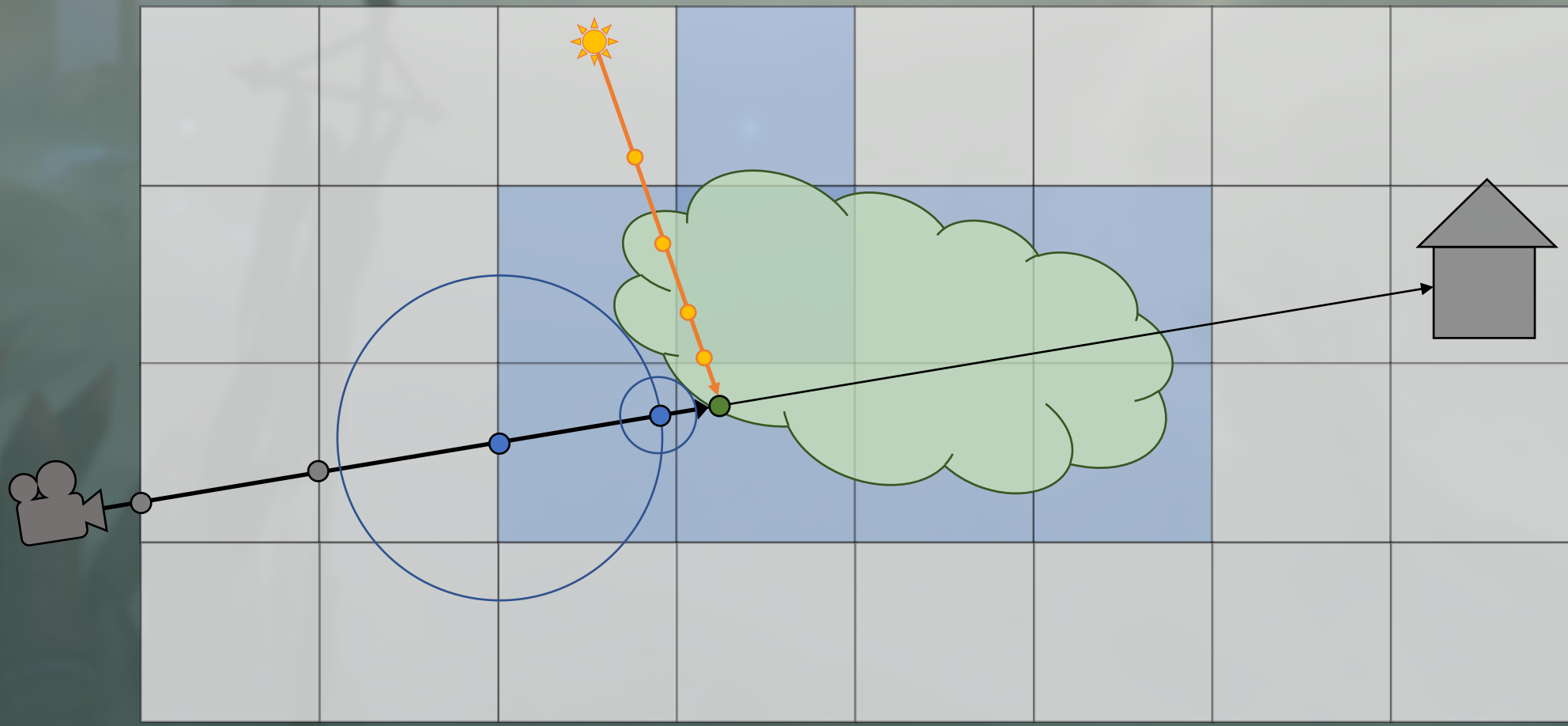
Ray marching



Ray marching



Ray marching



Ray marching - Lighting

- Single scattering from directional light
  - Sample cascade shadow map
  - Sample volumetric shadow volume
  - Four addition fog voxel samples in light direction for detail volume shadows
  - Mix of two HG phase functions with different anisotropy [Hillaire16]

Ray marching - Lighting

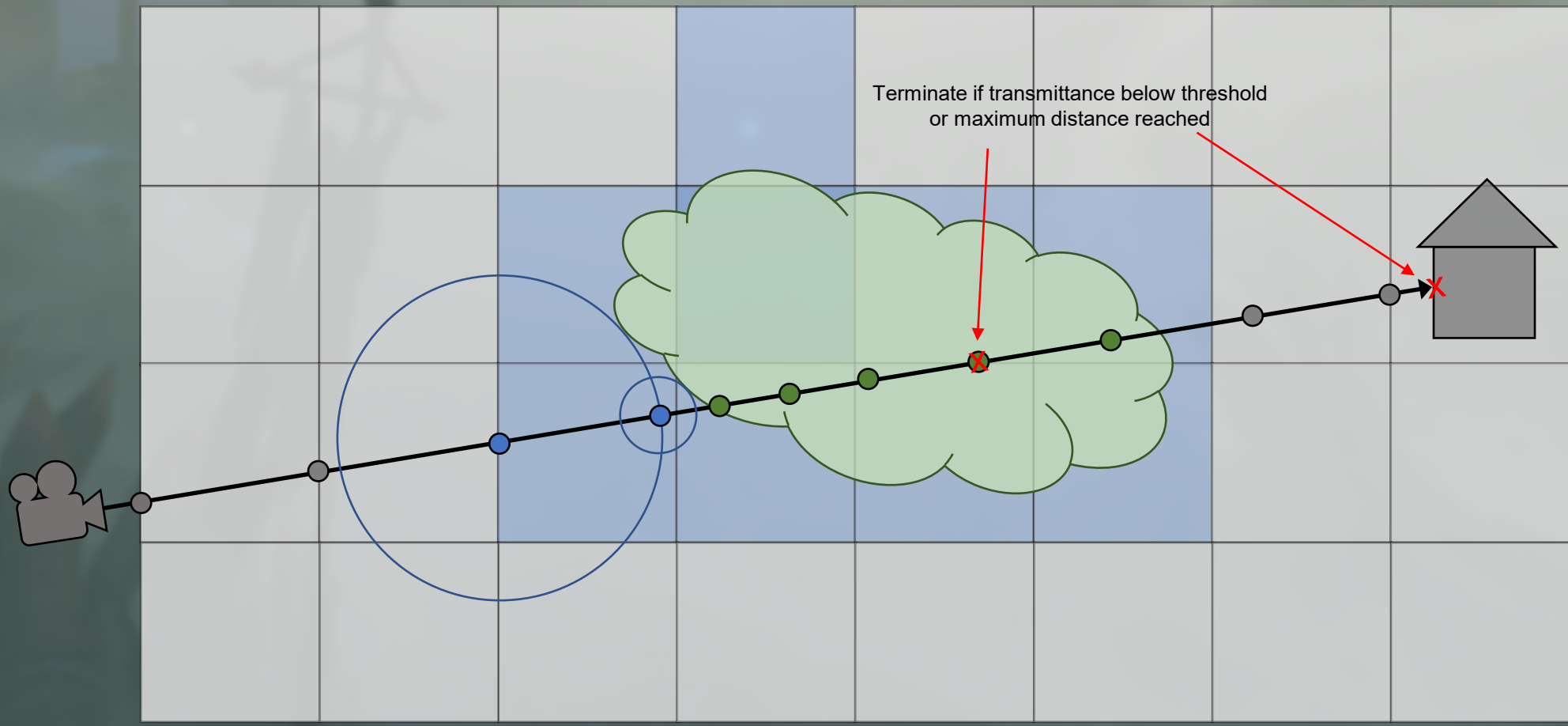
- Lookup radiance froxel volume for point lights, emissive and global illumination
  - Very flat because of missing volumetric shadows
  - Change intensity and scatter albedo based on voxel fog density

Ray marching – Combine with Froxel Volume

- Sample radiance and transmittance from froxel volume
- Compute froxel radiance of ray segment by subtracting previous sample radiance
- Sum froxel and ray march radiance weighted by the transmittance of each other

```
// sample froxel volume
const float3 froxelAccumulatedRadiance = sampleFroxelRadiance( ... );
const float3 froxelAccumulatedTransmittance = sampleFroxelTransmittance( ... );
// compute radiance for given ray segment
const float3 froxelSampleRadiance = froxelAccumulatedRadiance - prevFroxelAccumulatedRadiance;
prevFroxelAccumulatedRadiance = froxelAccumulatedRadiance;
// combine with ray march sample
const float3 sampleRadiance = rayMarchSampleRadiance * froxelAccumulatedTransmittance
                            + froxelSampleRadiance * accumulatedTransimttance;
accumulatedRadiance += sampleRadiance;
```

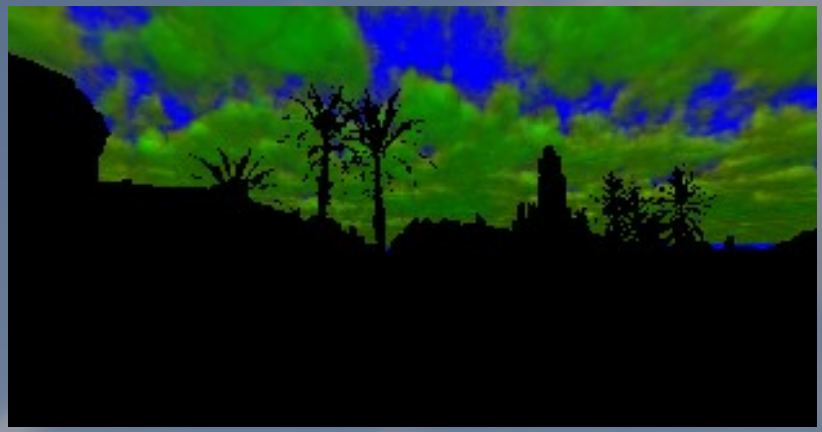
Ray marching



Clouds

Clouds

- Ray march 1 of 16 rays in full resolution
- Store in ¼ resolution array texture with 16 slices
- Store view projection matrix for each slice
- Reconstruct full res from all slices
- R10G10B10
  - R: Directional light scattering intensity
  - G: Ambient scattering intensity
  - B: Transmittance
- Apply light color and phase function during reconstruction

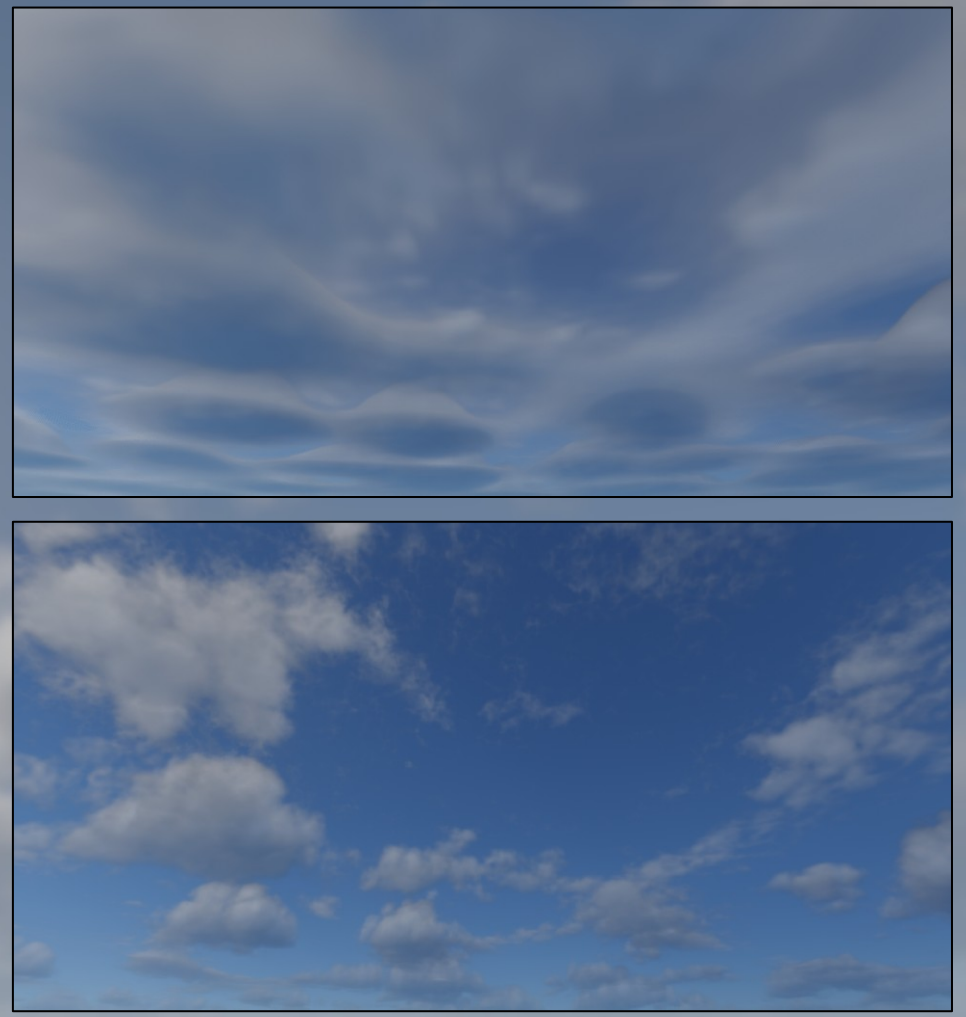


Clouds – Ray marching

- Generate sky ¼ resolution mask texture from depth buffer
- Early out rays that didn't hit the sky
- Find ray start and end distance by intersecting two spheres with cloud layer start and end radius
- Ray march sample count depends on quality settings
- Early out ray march loop if transmittance below threshold

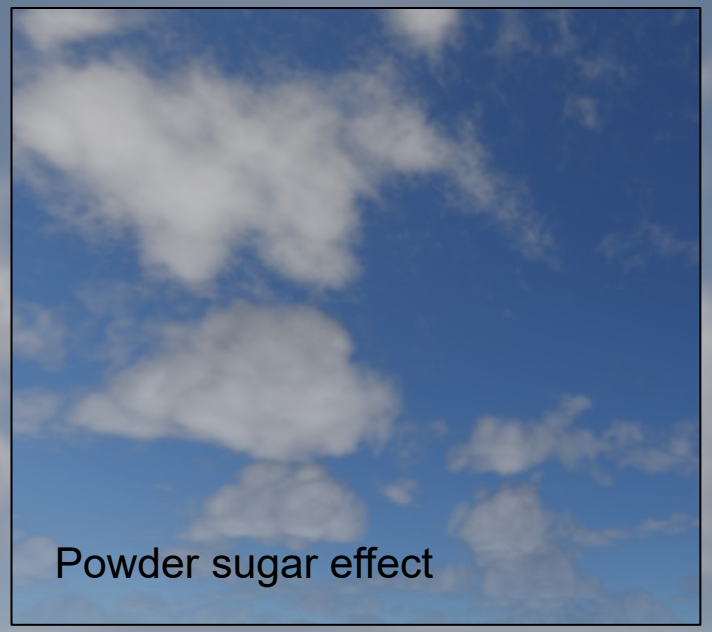
Clouds – Ray marching

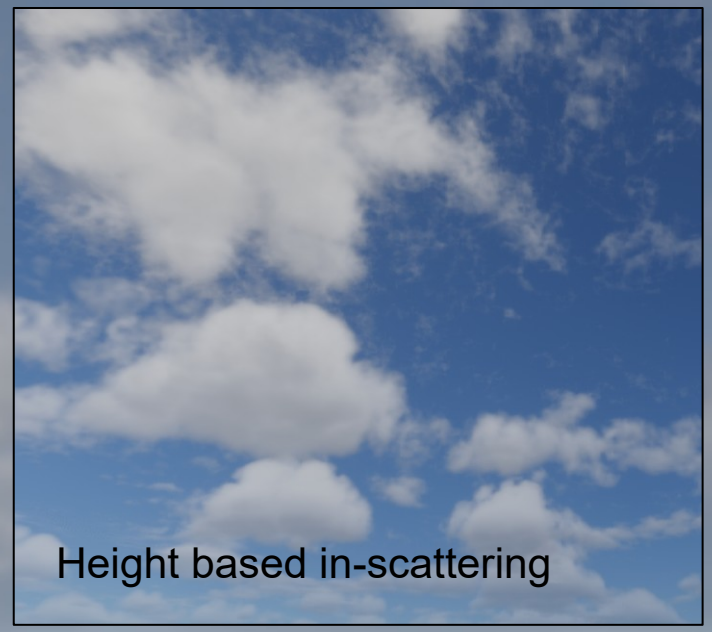
- Mostly follows [Schneider17] [Schneider22]
- Construct density from cloud map and samples relative height
- Erode density by detail noise



Clouds – Ray marching

- Approximated multi scattering from primary directional light
- Darken edges “powder sugar effect”
- Decrease in in-scattering at the bottom of the cloud





Composite

Composite

Composite

Water

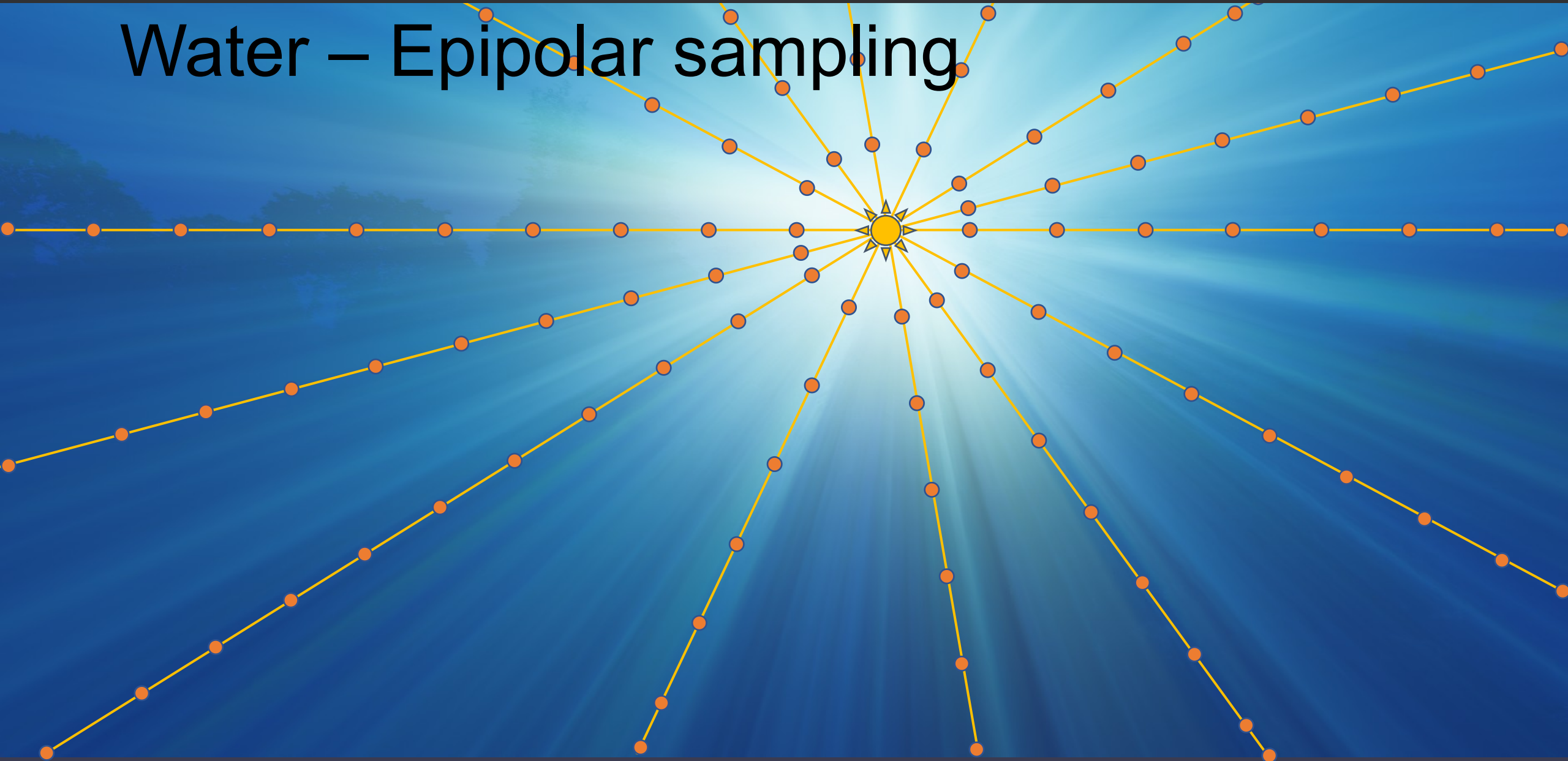
Water – Observations

- Interface between air and water prohibits us from rendering water together with other media
- Can be treated as homogenous media
- Water is “exclusive” and won’t be mixed with other media like fog or atmosphere

Water – God rays

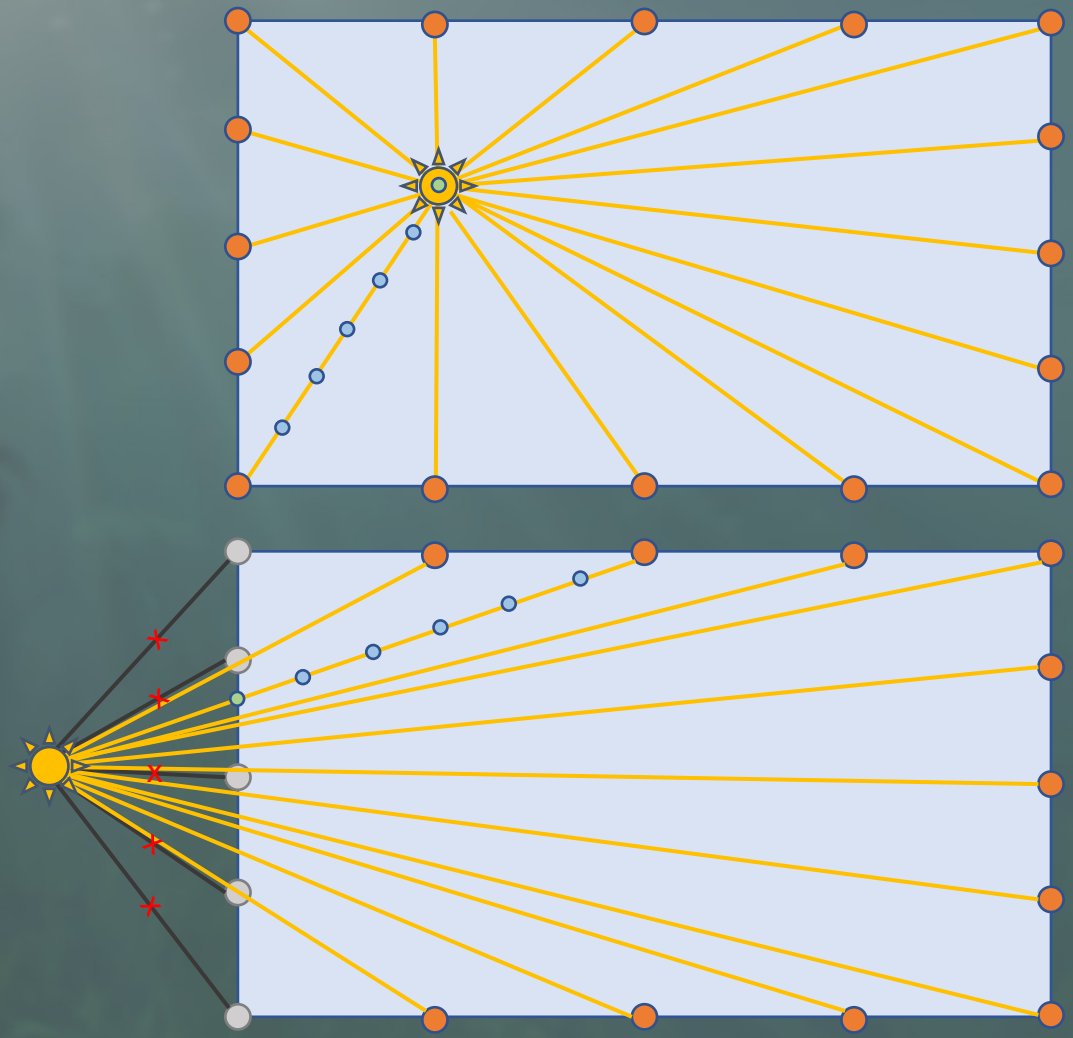
- Good rays occur where
  - Light abruptly change -> shadows
  - Light bundled by reflections or refractions
- Want sharp, high frequent god rays from directional light
- Want god rays animated and fast responding

Water – Epipolar sampling



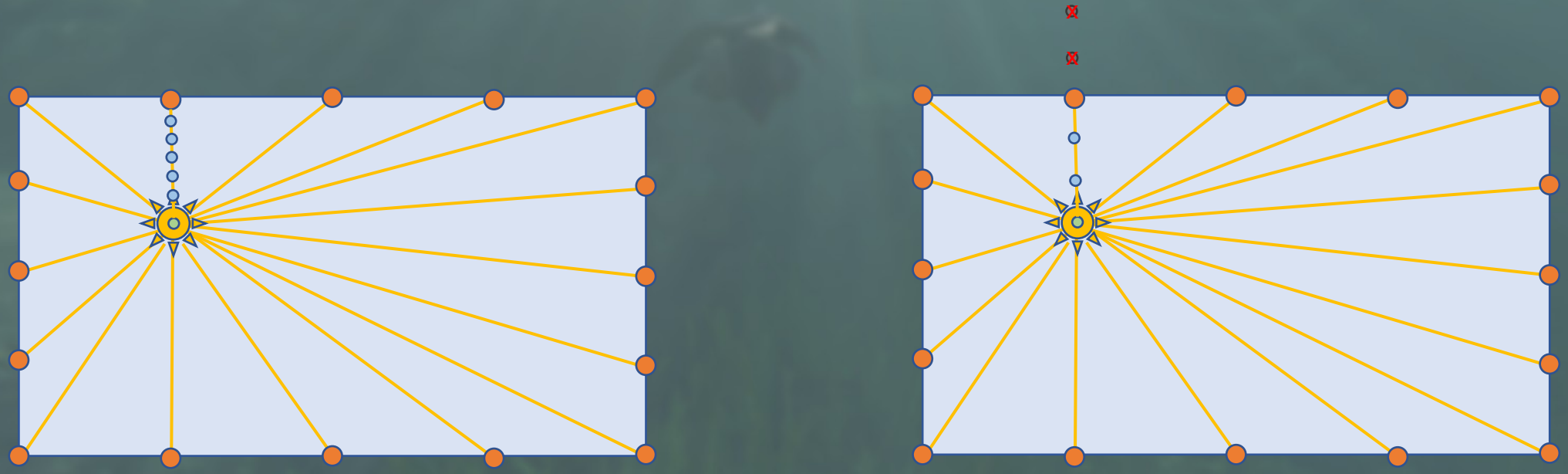
Water – Sample generation

- Find start and end points for all epipolar lines
  - Project light on screen
  - If light is on screen
    - start point is the lights screen space position
  - If light is outside of screen
    - some lines are complete outside and became invalid
    - The remaining lines need to be intersected and truncated against the screen borders
  - Equidistantly placing end points along the border of the screen
  - Store start and endpoints in buffer



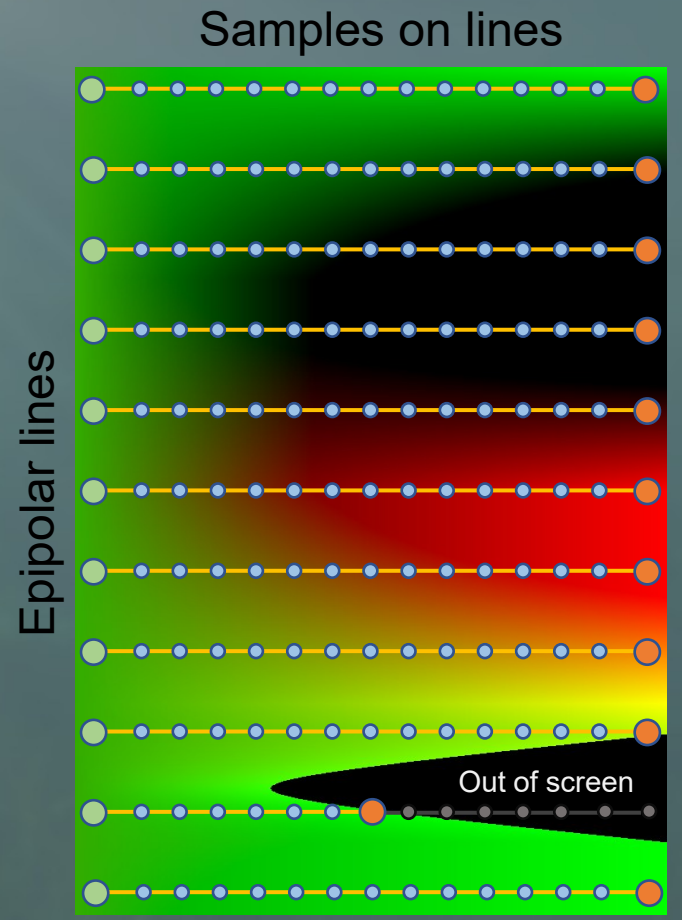
Water – Sample generation

- Equidistantly place samples between start and end points
  - If light source is close to screen border samples became to dense
  - Scale sample distance to be equal in screen space
  - Samples at end of epipolar line could be out of screen and will became invalid



Water – Sample generation

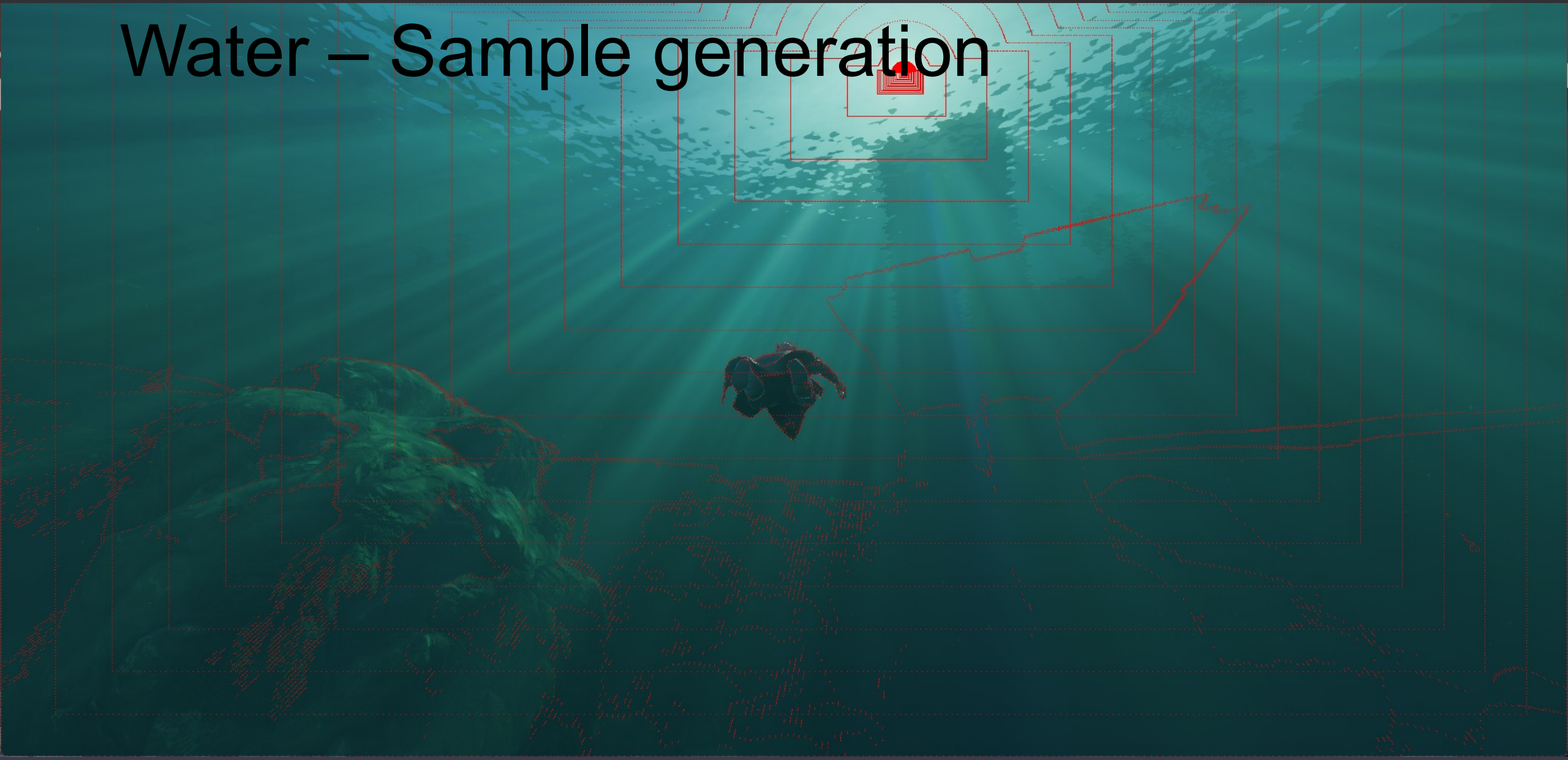
- Compute shader stores screen space sample coordinates in lookup texture
  - Each row corresponds to one epipolar line
  - Each column corresponds to one sample on the line
  - Set invalid samples to negative off-screen coordinate
  - Line and sample count can be freely chosen
  - We use 1024 lines with 512 samples



Water – Sample generation

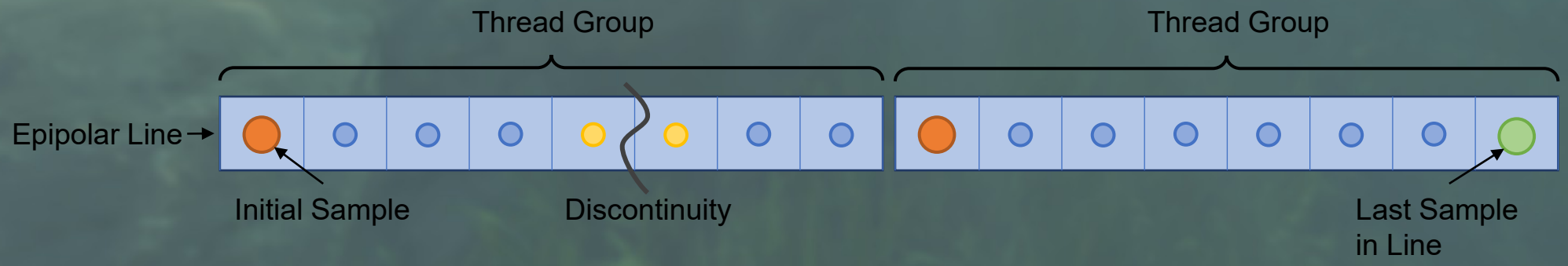
- We don’t want to ray march all the samples
- Place initial ray marching samples every N samples.
- Place more ray marching samples where they are really needed
- Remaining samples will be linear interpolated from the closest ray marching samples

Water – Sample generation



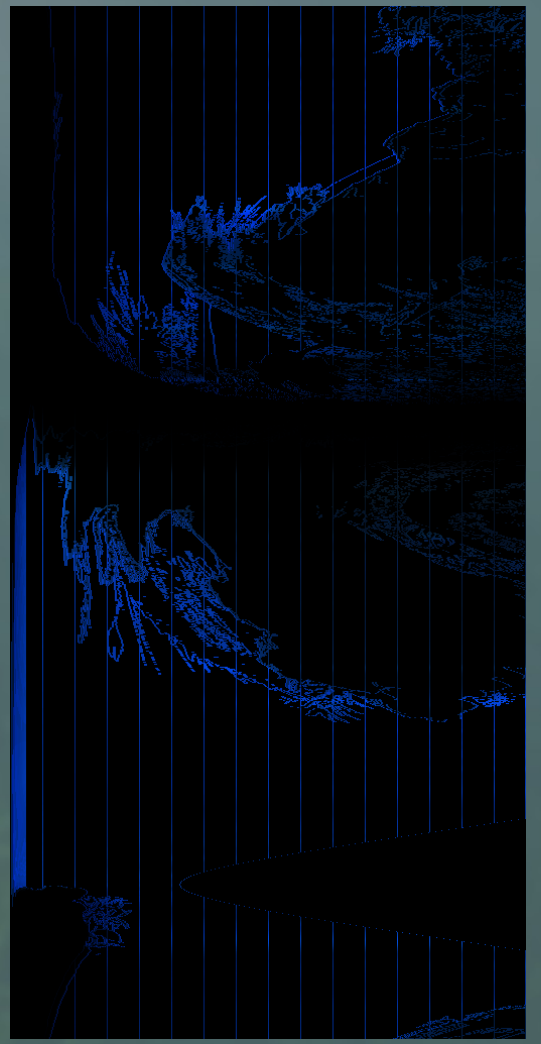
Water – Sample generation

- Run compute shader with one thread per epipolar coordinate
- Thread group size equal to initial ray marching sample distance
- First sample in thread group and last sample in line will always ray marched
- InterlockedOr() depth and lighting discontinuities in group shared bitmask
- Find discontinuity using shared bitmask and add ray-marching sample on left and right side
- Append all ray marching samples to a buffer
- InterlockedOr() all ray marching samples in R32 uint bitmask texture for later interpolation



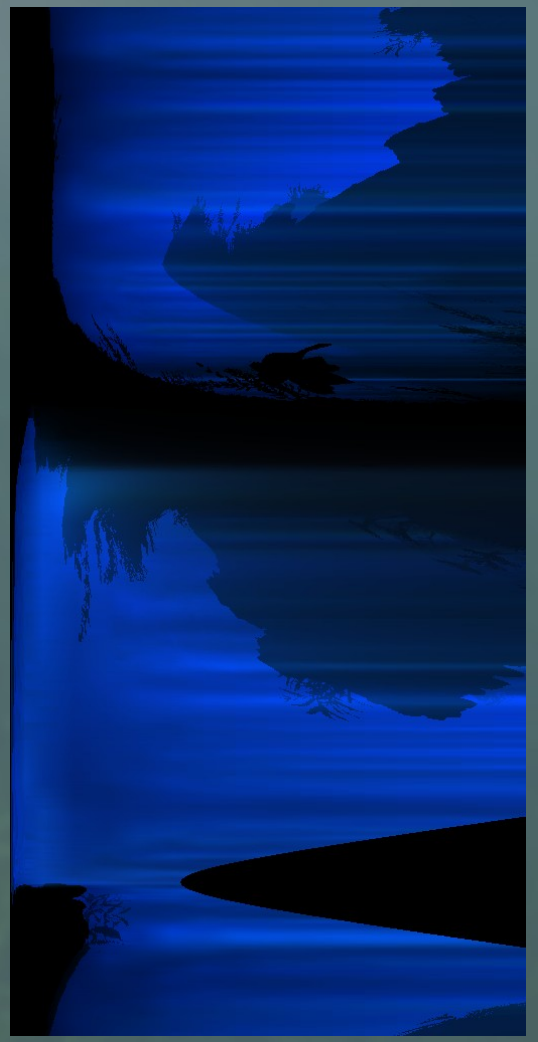
Water – Ray marching

- Ray-march in light space
- Sample cascade shadow map and shadow volume
- Compute transmittance from water shadow map
- Sample animated caustics texture
- Phase function is applied later in compositing
  - Phase function could break due to linear interpolation on epipolar line
- Output sparsely into epipolar texture



Water – Interpolation

- Linearly interpolate missing samples
- Linear search two nearest ray-marching samples
  - Use R32 uint bitmask texture from sample generation
  - We have an upper limit for the linear search because of initial sample location
- Store the result in same texture where we load ray marched samples from
  - No contention because ray marched samples will not be interpolated



Water – Interpolation

- Still need to “unwrap” the epipolar texture to screen space
  - Compute screen space ray going from the light through the pixel
  - Find two closest epipolar lines from which we will interpolate
  - Project current pixel onto the epipolar lines using the precomputed start and end points
  - Compute UV coordinates and bilinear weights of the four epipolar interpolation samples
  - Weight by depth difference of epipolar sample and pixels depth
- Apply HG phase function

Water – Other in-scattering

- Ray march froxel volume to calculate in-scatter of other light sources
  - Low sample count
  - Fetch precomputed point light, global illumination and emissive lighting from froxel volume texture
  - Add to directional light in-scattering
- Calculate max ray length from water extinction coefficients

Water – Compositing

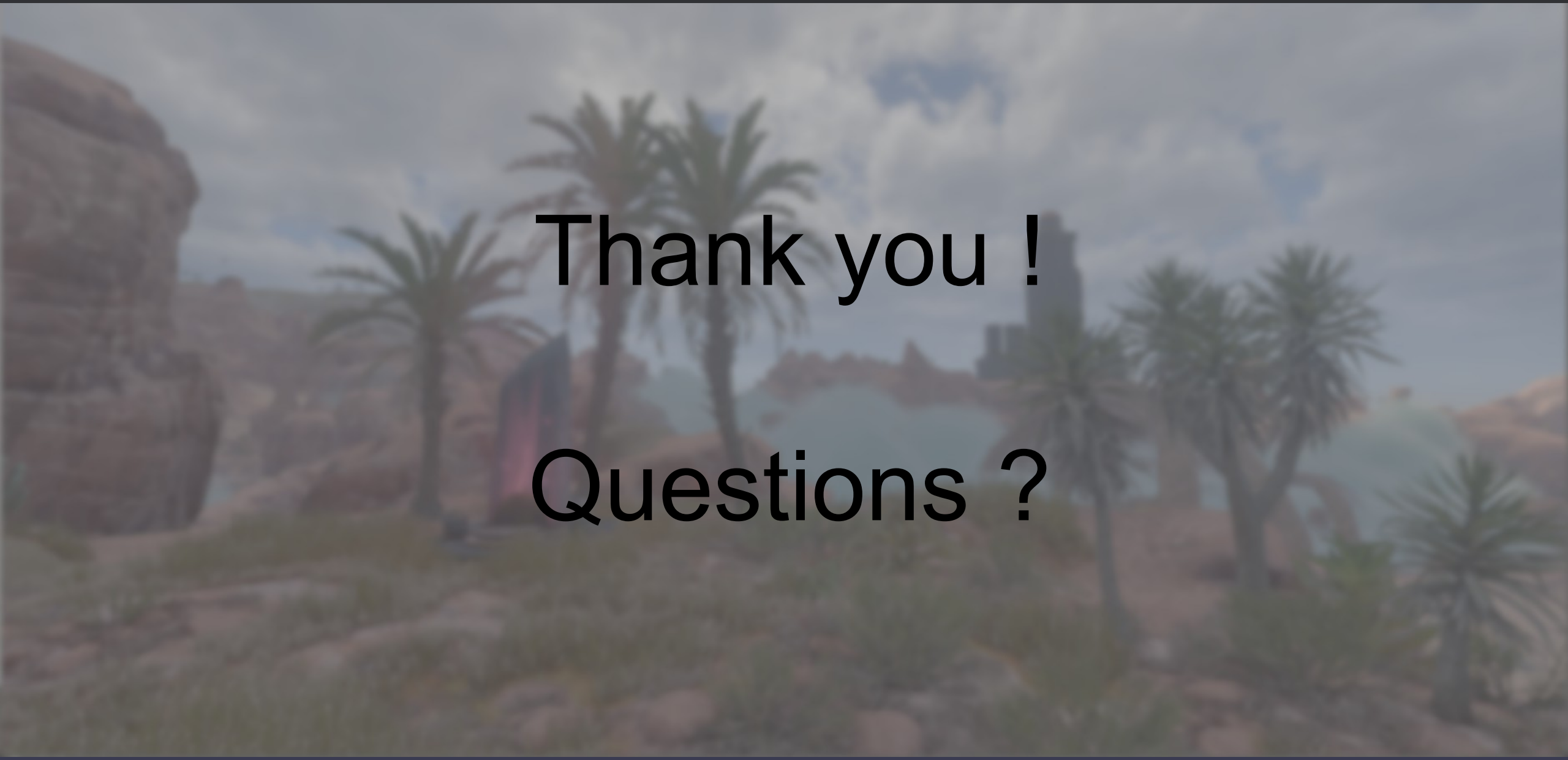
- Two full-screen passes
  - Needed because of refraction through air/water interface
- Also water surface is rendered during composite
- For more information's check out Andreas Mantler’s talk about water rendering in Enshrouded

Transparent Draws

- Transparent draws are separated into underwater and above water draws
- For underwater draws we need to apply water only
  - Calculate transmittance based on pixels depth
  - Multiply color by transmittance
  - Multiply alpha by transmittance luminance
- For over water draws we need to combine with fog only
  - Inspired by Variance Based Depth [Tatarchuk13]
  - Calculate fraction of fog by using mean and variance from transmittance weighted depth moments
  - Fails in some situations -> not happy with this solution

Conclusion

- Volume rendering is hard
- Unified solution for different media did not work for us
- Temporal accumulation not ideal for volumetric rendering but unfortunately still needed
- Sometimes difficult to remain physically based and still meet the art direction
- Transparent draw compositing is subject of ongoing research
- We would like to integrate volume rendering in our GI but haven’t found a suitable solution yet


Thank you !
Questions ?