SIGGRAPH2014

NATURALLY DIGITAL

vancouver

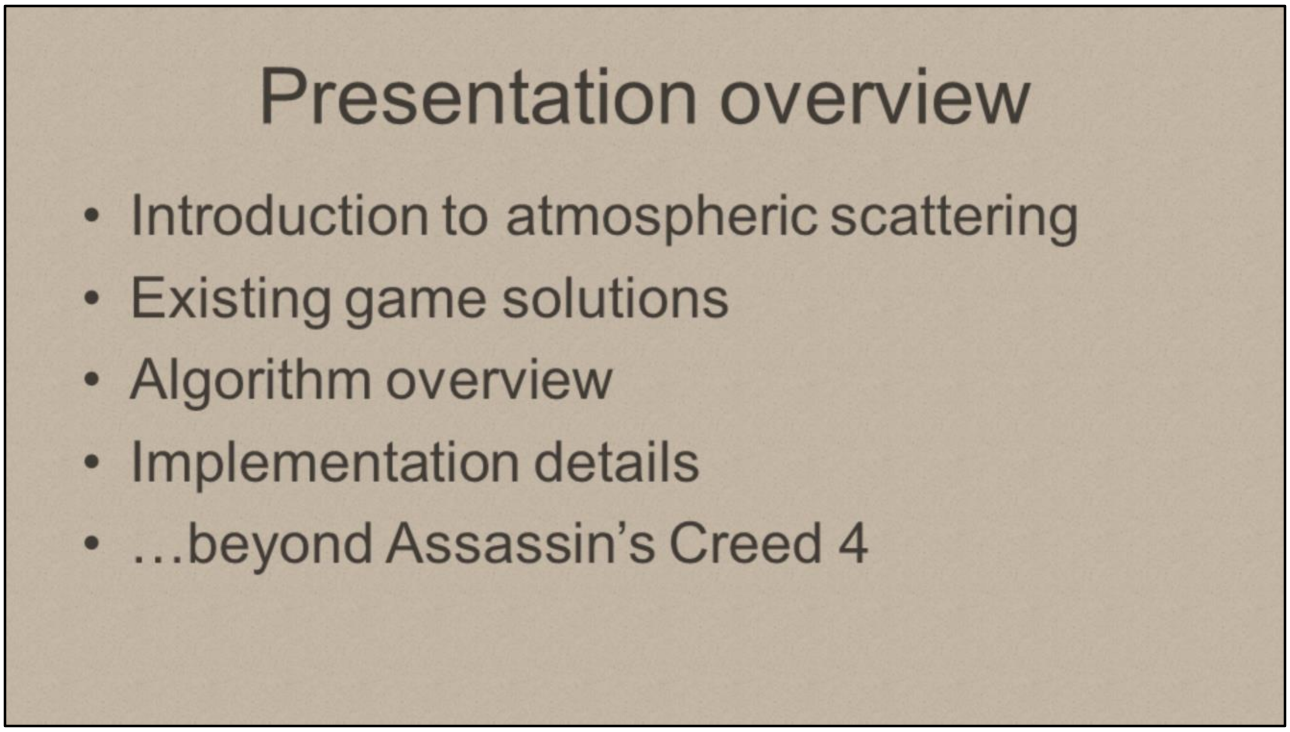
vancouver
SIGGRAPH2014
NATURALLY DIGITAL

The 41st International Conference and Exhibition on Computer Graphics and Interactive Techniques

UBISOFT™

Volumetric Fog: Unified compute shader based solution to atmospheric scattering

Bartlomiej Wronski
Ubisoft Montreal

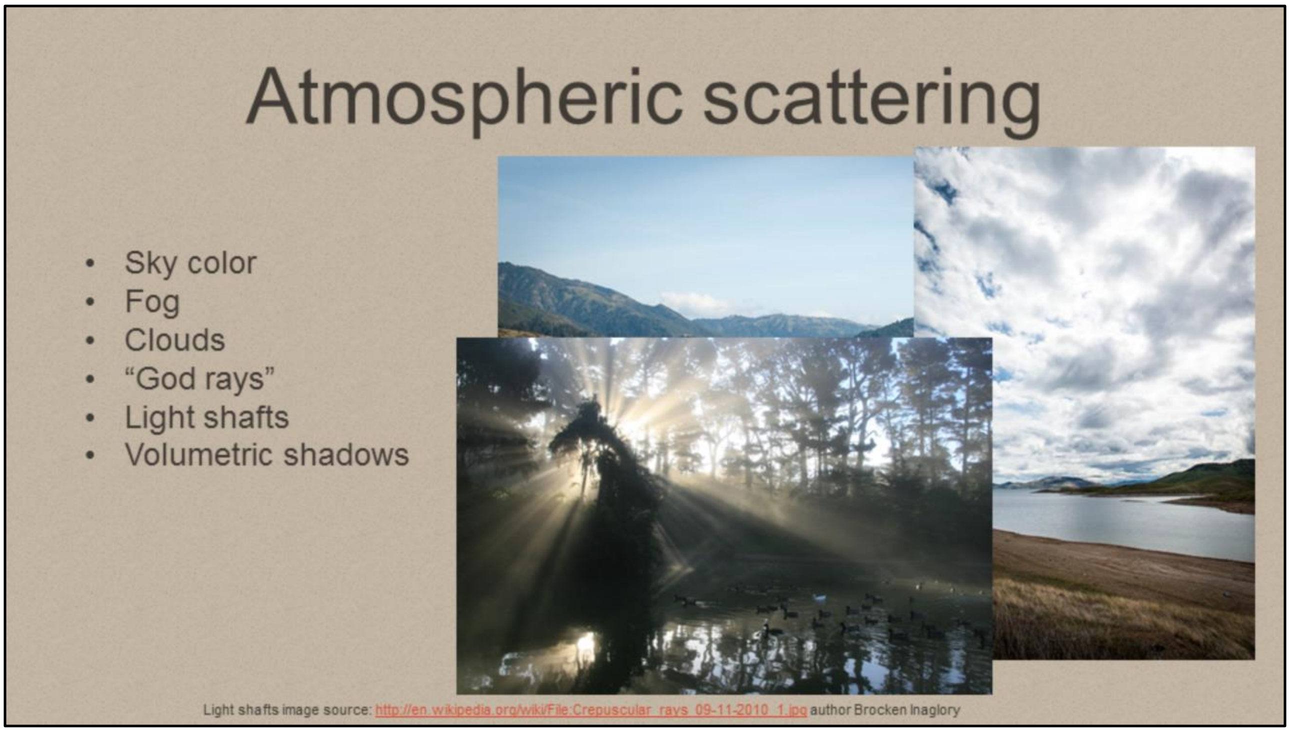


Presentation overview

- Introduction to atmospheric scattering
- Existing game solutions
- Algorithm overview
- Implementation details
- …beyond Assassin’s Creed 4

This presentation will be divided in 5 sections:

- Small introduction into atmospheric scattering phenomenon - its importance in realistic rendering and in games and physical laws that cover this topic.
- Brief description of how previous games handled the atmospheric effects and why none of existing solutions was enough for us.
- Quick overview of key ideas behind our algorithm, its steps and used data structures.
- …also small implementation details and keys to make algorithm fast and robust.
- And finally ideas we wanted to implement and are implementing for future projects, but didn’t make it into AC4 due to time or project constraints.

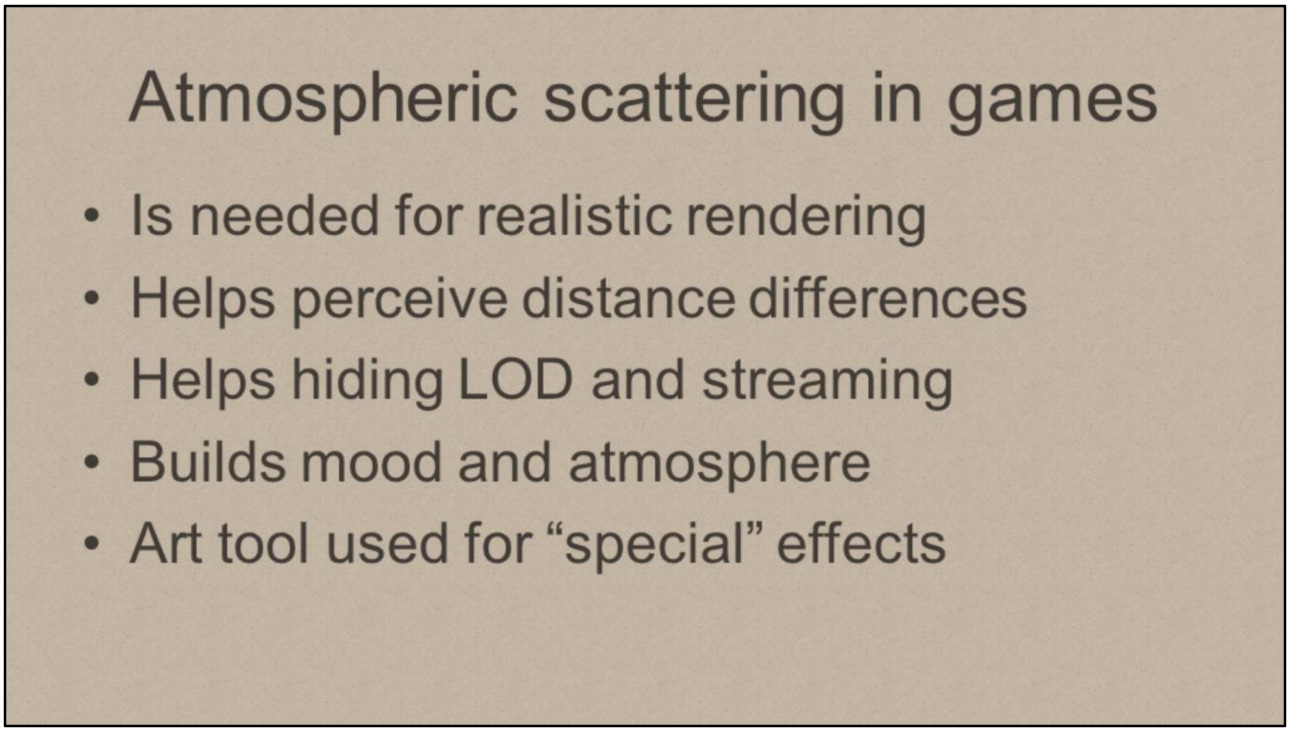


Atmospheric scattering

- Sky color
- Fog
- Clouds
- “God rays”
- Light shafts
- Volumetric shadows

Light shafts image source: http://en.wikipedia.org/wiki/File:Crepuscular_rays_09-11-2010_1.jpg author Brocken Inaglory

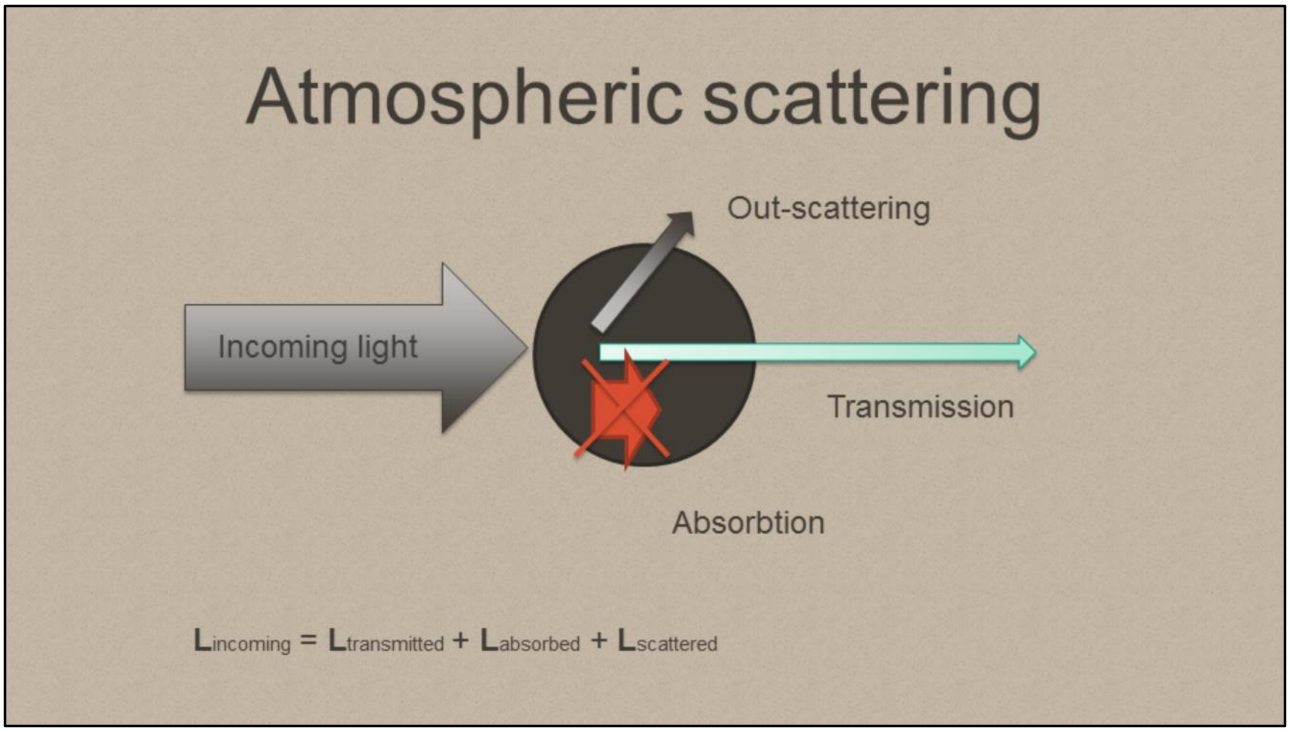
So, why do we need atmospheric scattering simulation?
It is physical effect responsible for many visual elements of visible world, such as sky color, fog, clouds, god rays, light shafts, volumetric shadows or even smoke.



Atmospheric scattering in games

- Is needed for realistic rendering
- Helps perceive distance differences
- Helps hiding LOD and streaming
- Builds mood and atmosphere
- Art tool used for “special” effects

So why do we bother with such effects in games?
They are needed not only to reproduce reality correctly, but also helps player or viewer to perceive distances between objects and creates uniform scene composition.
It helps to hide LOD and streaming (even very old games used very thick linear fog to hide very short streaming distance).
…and finally it is very important tool and part of art direction. It helps to build scene mood and atmosphere or do some special effects.



Atmospheric scattering

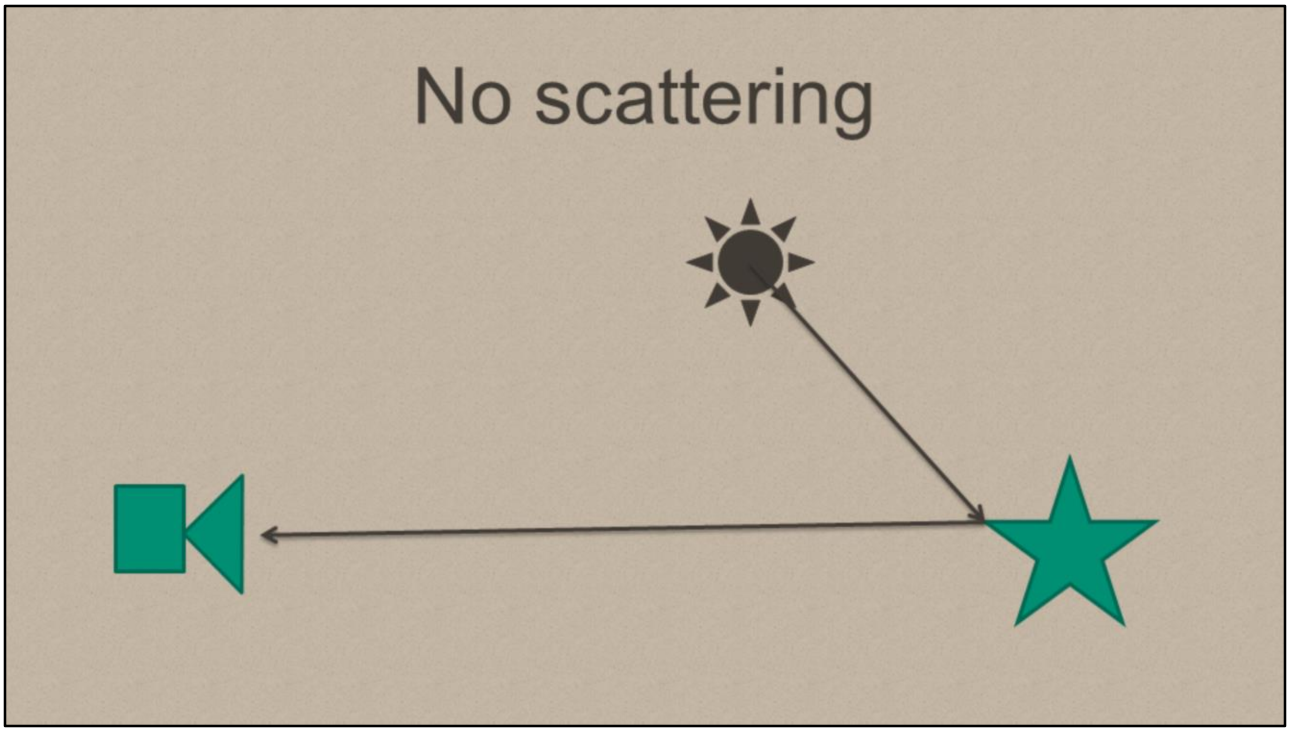
$L_{incoming} = L_{transmitted} + L_{absorbed} + L_{scattered}$

Phenomenon of atmospheric scattering is caused by interaction of photons with particles that form any transporting media. When light traverses any medium that isn’t void, photons/light rays may collide with particles that create such medium. On collision, they may collide and either be diffused or absorbed (and turned into thermal energy). In optics, such process is usually modelled statistically and we can define amount of energy that takes part in following processes:

- Transmittance
- Scattering
- Absorbtion

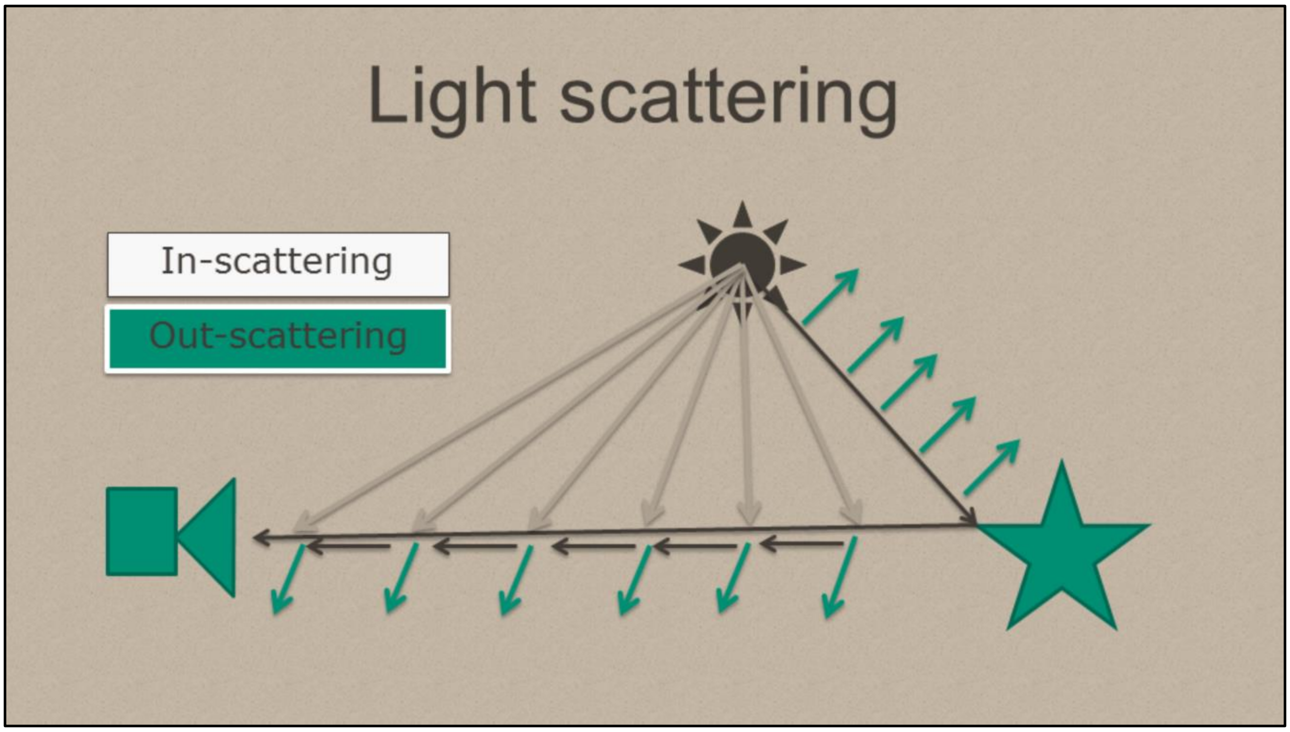
As energy is always conserved, we can write that:

Lincoming = Ltransmitted + Labsorbed + Lscattered



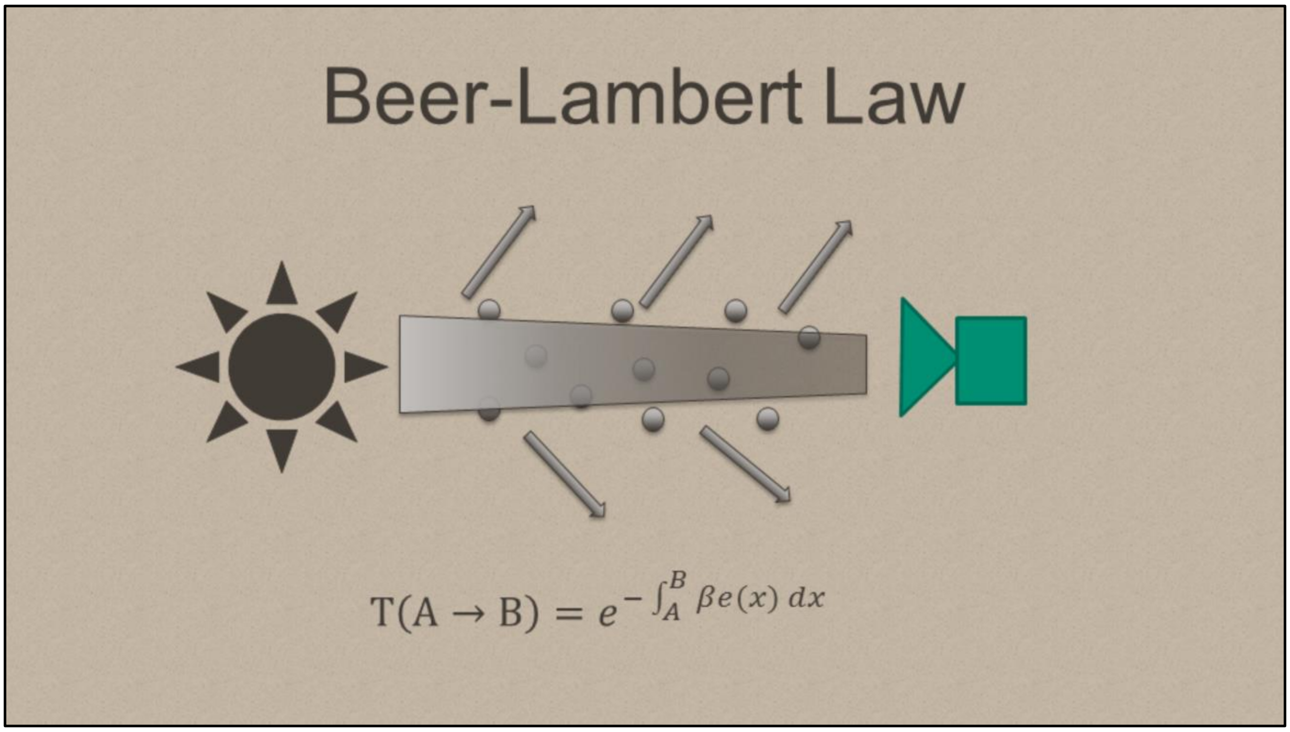
No scattering

No light scattering / non-participating transport media
We assume that the transport media behaves like vacuum – there is no radiance loss or gain on light paths between objects.
Typical rendering scenario – light from the light source bounces from one object to another according to the surface BRDF functions, finally reaching camera/eye.
Simplest possible case is no bounced light/GI, just direct lighting.



When media participates in light transport, every particle large enough to affect photons/light rays takes part in the light transport equation.
For example dust or water particles make some light rays / photons bounce in random directions, making some light enter the light path (in-scattering).
While on the other hand also some light gets bounced away, exiting the light path and becoming darker (out-scattering).

Obviously, in reality this is very complex as every particle both out-scatters and in-scatters some light according to phase functions, so multiple rays exit and enter the light path multiple times, but usually in real-time rendering we have to ignore multiple scattering.



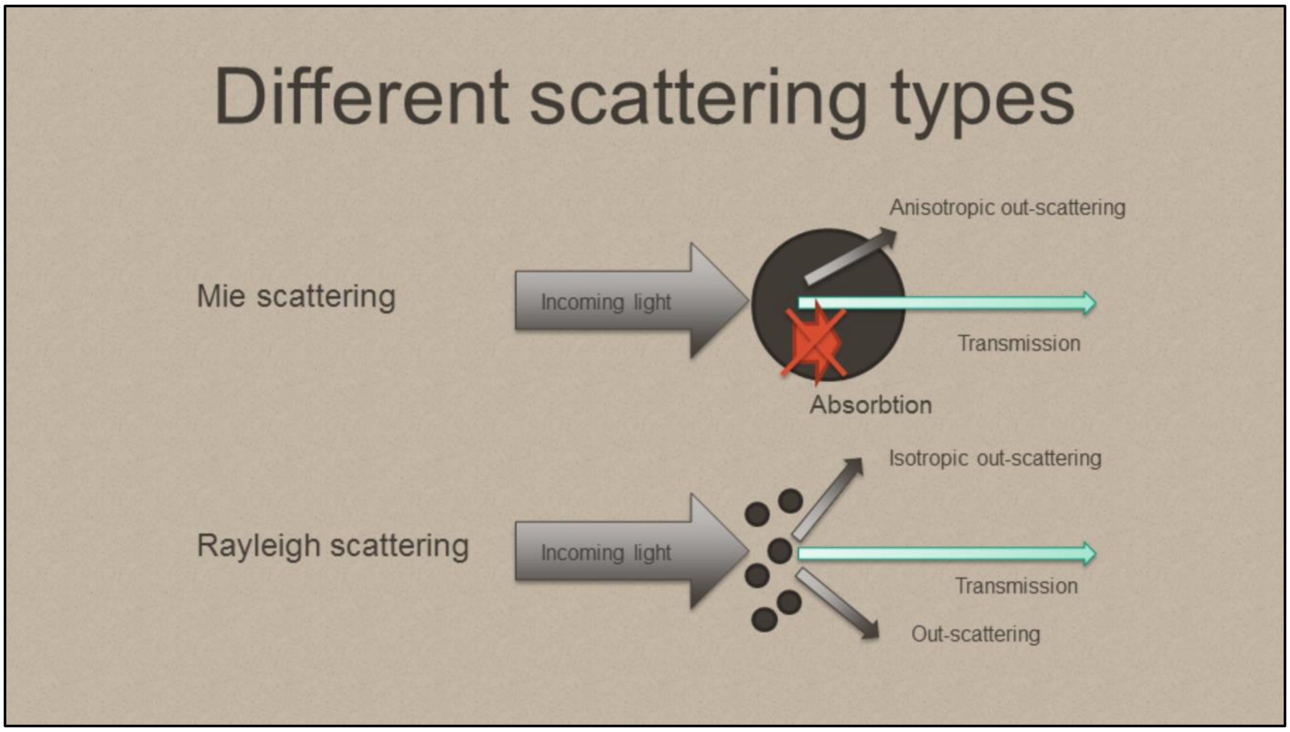
Beer-Lambert Law

$$T(A \to B) = e^{-\int_A^B \beta e(x)\, dx}$$

Physical law that is very useful for light scattering calculations is Beer-Lambert law that describes extinction of incoming lighting (light out-scattering). This law defines value of transmittance (proportion of light transported through medium to incoming light from given direction). It is defined usually as:

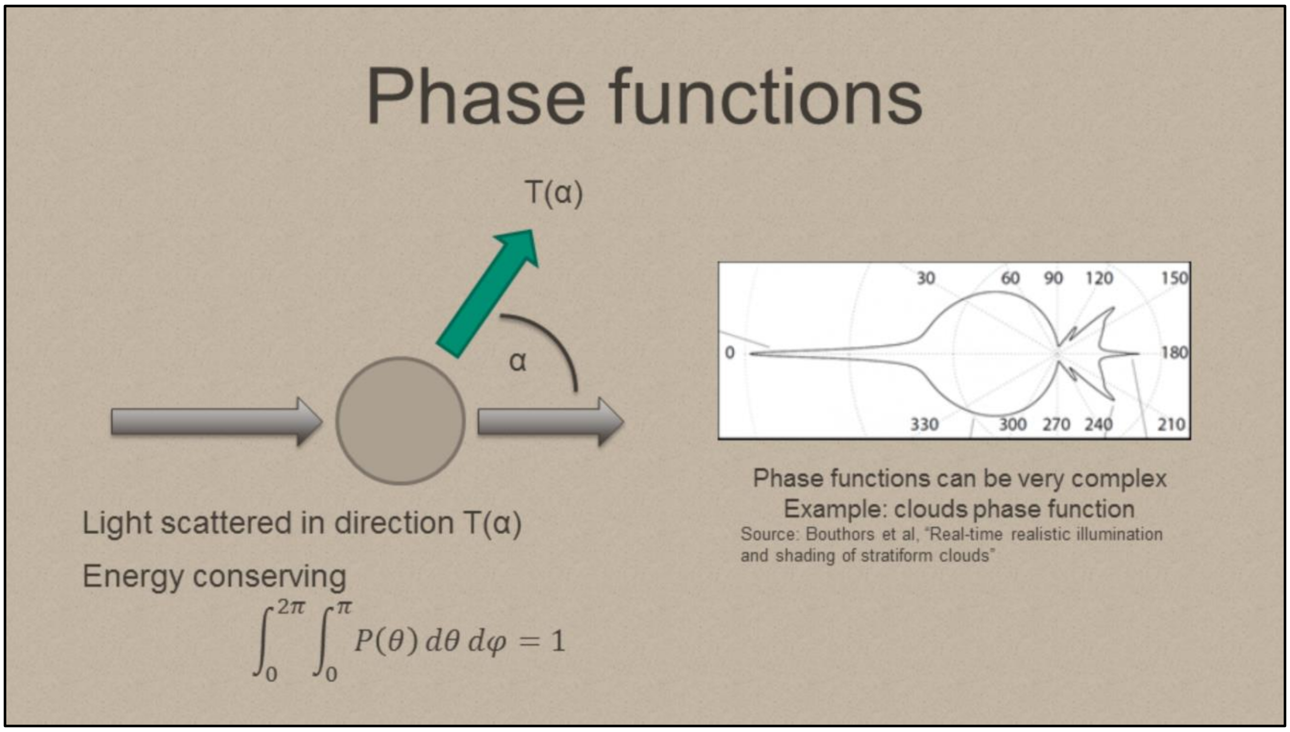
$$T(A \to B) = e^{-\int_A^B \beta e(x)\, dx}$$

Symbol βe is extinction coefficient, defined as sum of scattering and absorption coefficients. We can see from Beer-Lambert law that light extinction is exponential function of travelled distance by light in given medium.

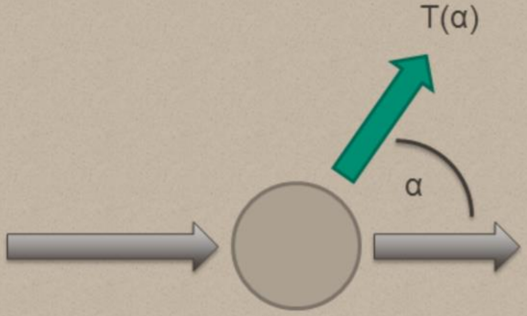

Different scattering types

Depending on the medium particles, amount of light that takes part in those processes can be different. One example of scattering models is Rayleigh scattering. It is scattering of very small particles (like air particles), responsible for blue color of the sky. It is very isotropic, uniform, but wavelength dependent – scattering is stronger for shorter wavelengths and absorption is negligible.

On the other hand, so called Mie scattering of bigger particles (like aerosols or dust) has very anisotropic shape with strong forward lobe and much higher absorption proportion.



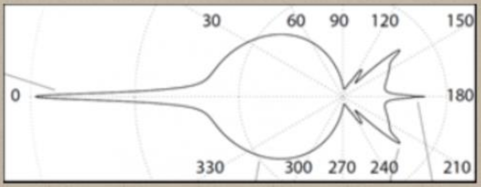
Phase functions



Light scattered in direction T(α)

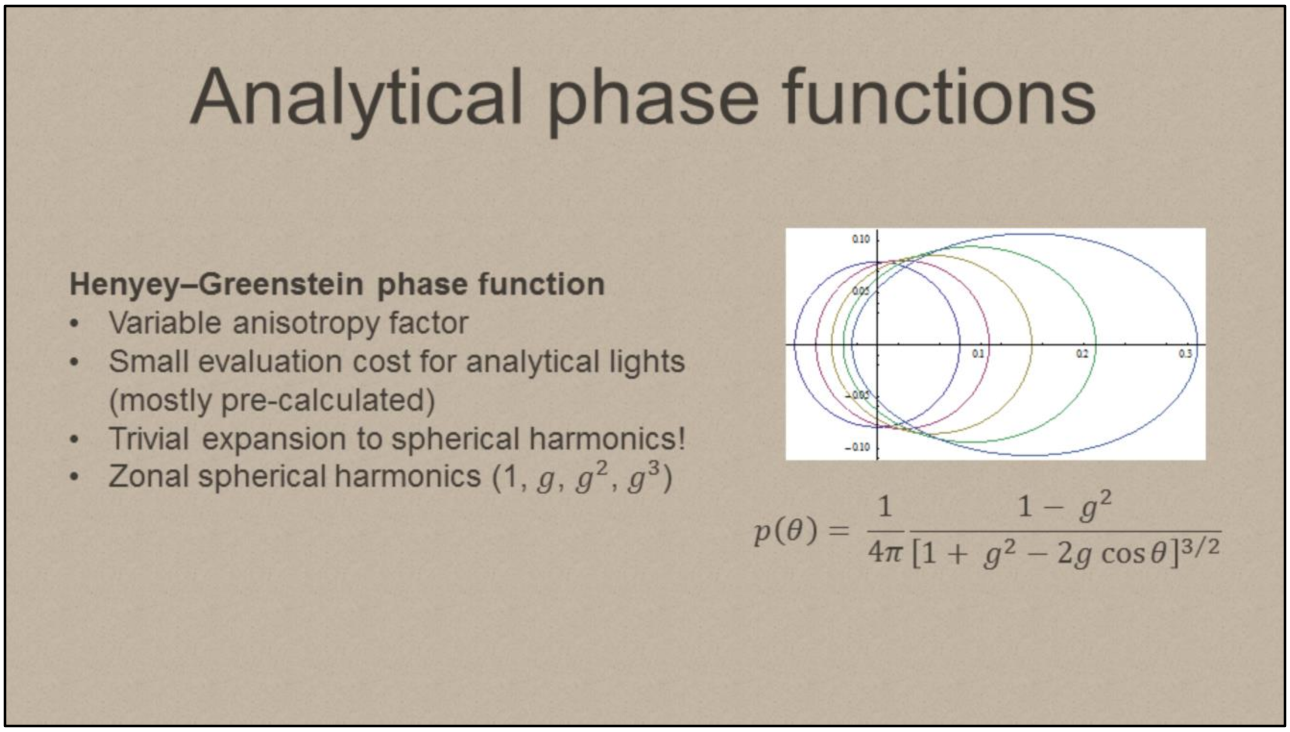
Energy conserving

$$\int_0^{2\pi}\int_0^{\pi} P(\theta)\,d\theta\,d\varphi = 1$$



Phase functions can be very complex
Example: clouds phase function
Source: Bouthors et al, "Real-time realistic illumination and shading of stratiform clouds"

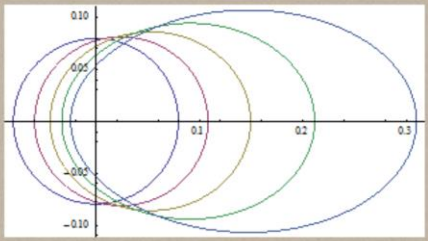
Phase function is a function that describes how much light is scattered in all direction.
It is a function of angle between the light vector and outgoing direction vectors.
It has property of energy conservation, so integral over all directions must be equal to 1 (or under 1 if it contains information about absorption baked in it).
Some phase functions can be very complex and described from various models or real captured data.



Analytical phase functions

Henyey–Greenstein phase function

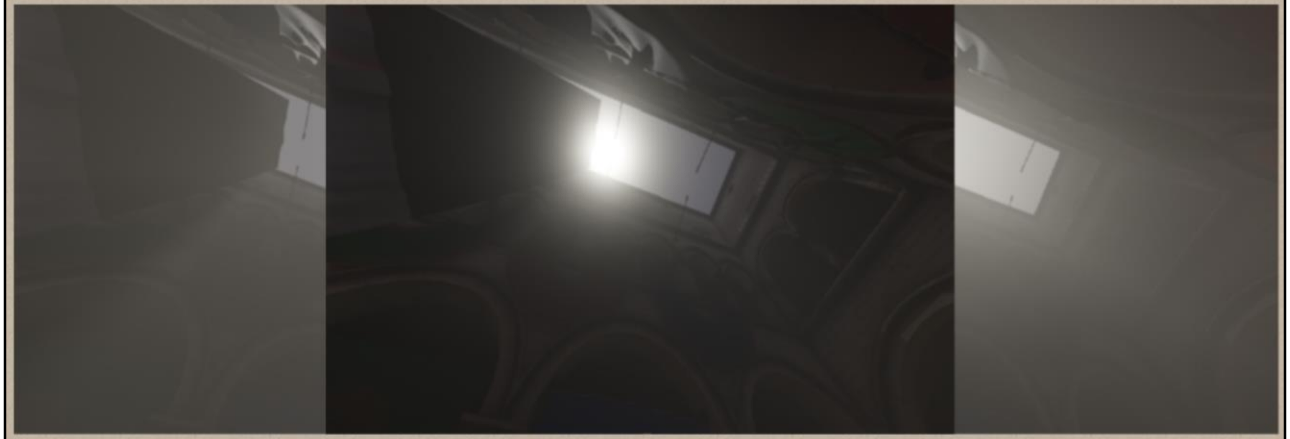
- Variable anisotropy factor
- Small evaluation cost for analytical lights (mostly pre-calculated)
- Trivial expansion to spherical harmonics!
- Zonal spherical harmonics ($1, g, g^2, g^3$)



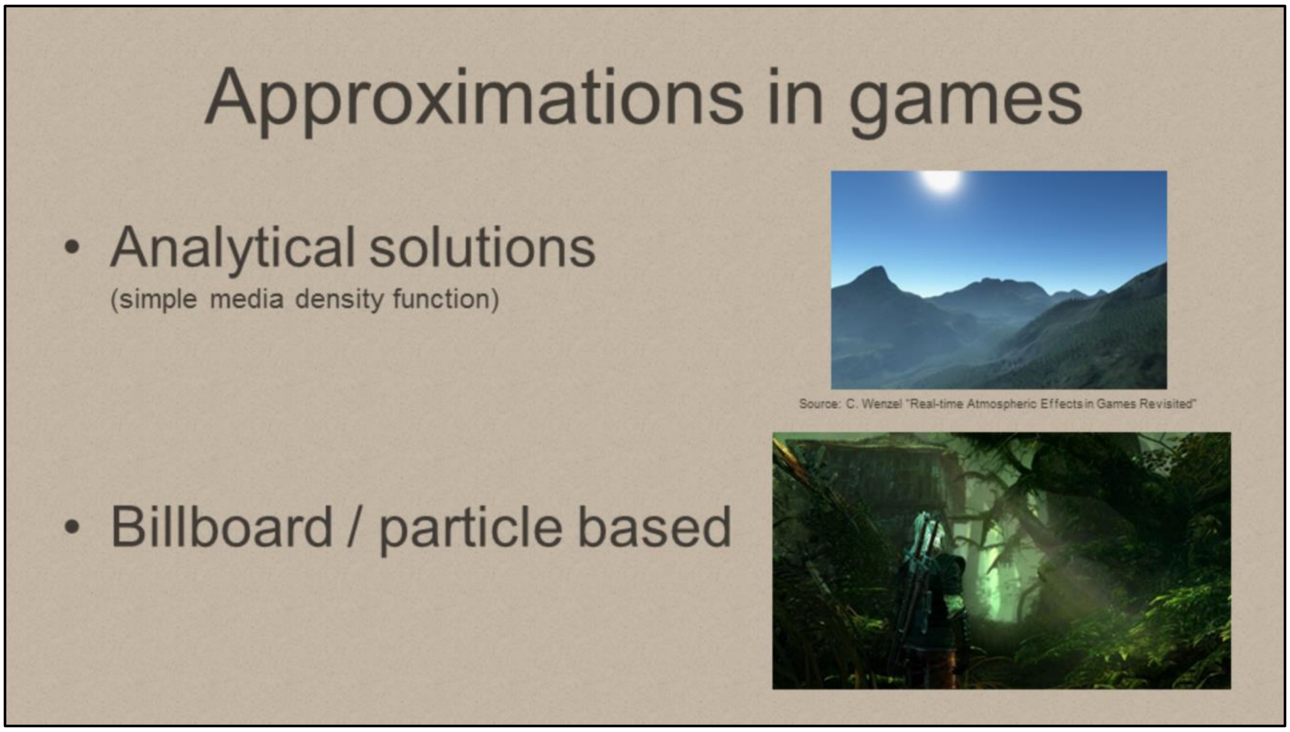
$$p(\theta) = \frac{1}{4\pi}\frac{1-g^2}{[1+g^2-2g\cos\theta]^{3/2}}$$

Most common phase function to simulate Mie-like anisotropic scattering is Henyey-Greenstein phase function. It has numerous advantages – it is efficient to evaluate in the runtime, expands trivially to spherical harmonics and supports variable anisotropy factor.
Some more complex phase functions can be constructed from weighted sum of multiple Henyey-Greenstein phase functions.

Scattering anisotropy



g = 0	g = 0.9	g = 0.3



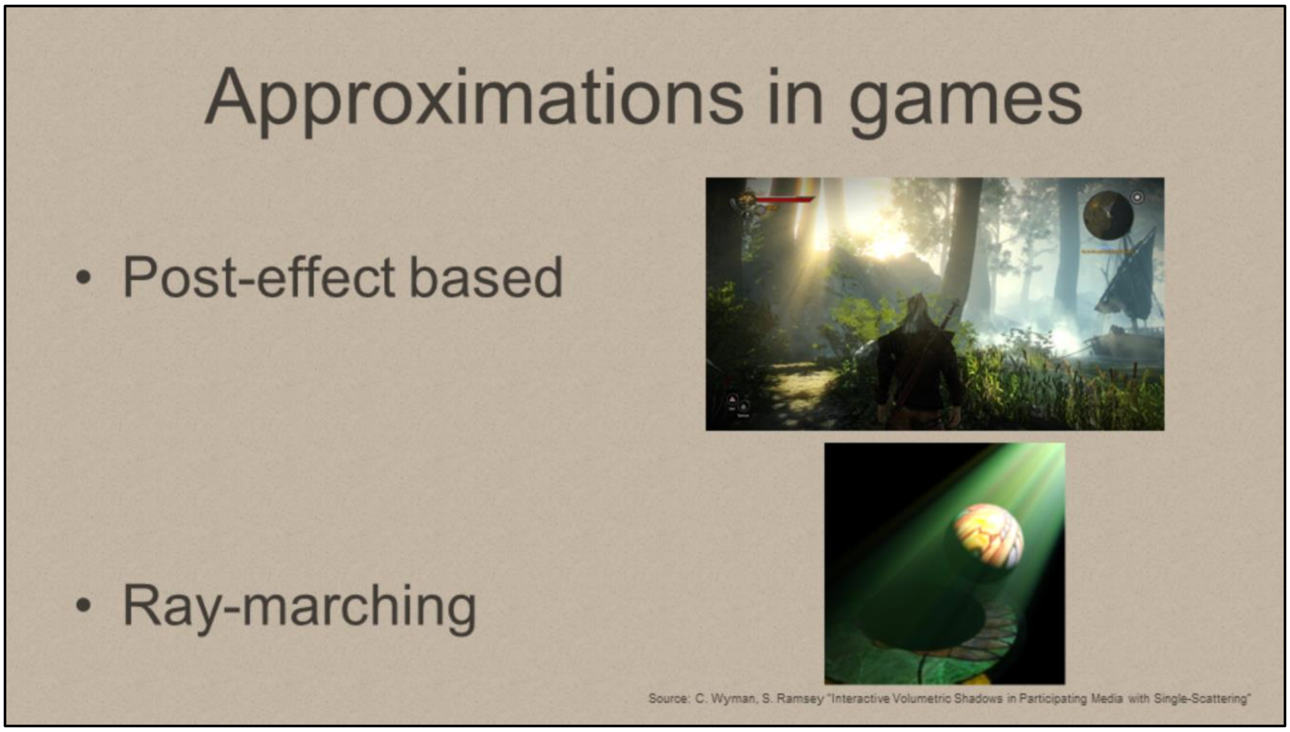
Approximations in games

- Analytical solutions
  (simple media density function)
- Billboard / particle based

Source: C. Wenzel "Real-time Atmospheric Effects in Games Revisited"

In games, different atmospheric and scattering phenomena were approximated using many approaches.
First one used since 90s and early OpenGL specifications are analytical solutions. First approximations used very simple linear depth based fog.
State of the art of fog rendering on previous console generation was analytical, exponential distance based fog. It was very well described by Wenzel in “Real-time atmospheric effects in games revisited” presentation. It is based on real scattering phenomenon and provides analytical solution to it – but unfortunately doesn’t handle the varying medium density and shadowing.

The second and probably easiest to use (not much programmer support needed) is using artist authored camera facing billboards or particles (with a fadeout on camera intersection). This approach has obvious disadvantage of being artist dependent, tedious to set up, not robust (eg with changing time of day and on some angles rotation can look “wrong”) and not dynamic enough (not reacting properly to changes in lighting and shadowing).



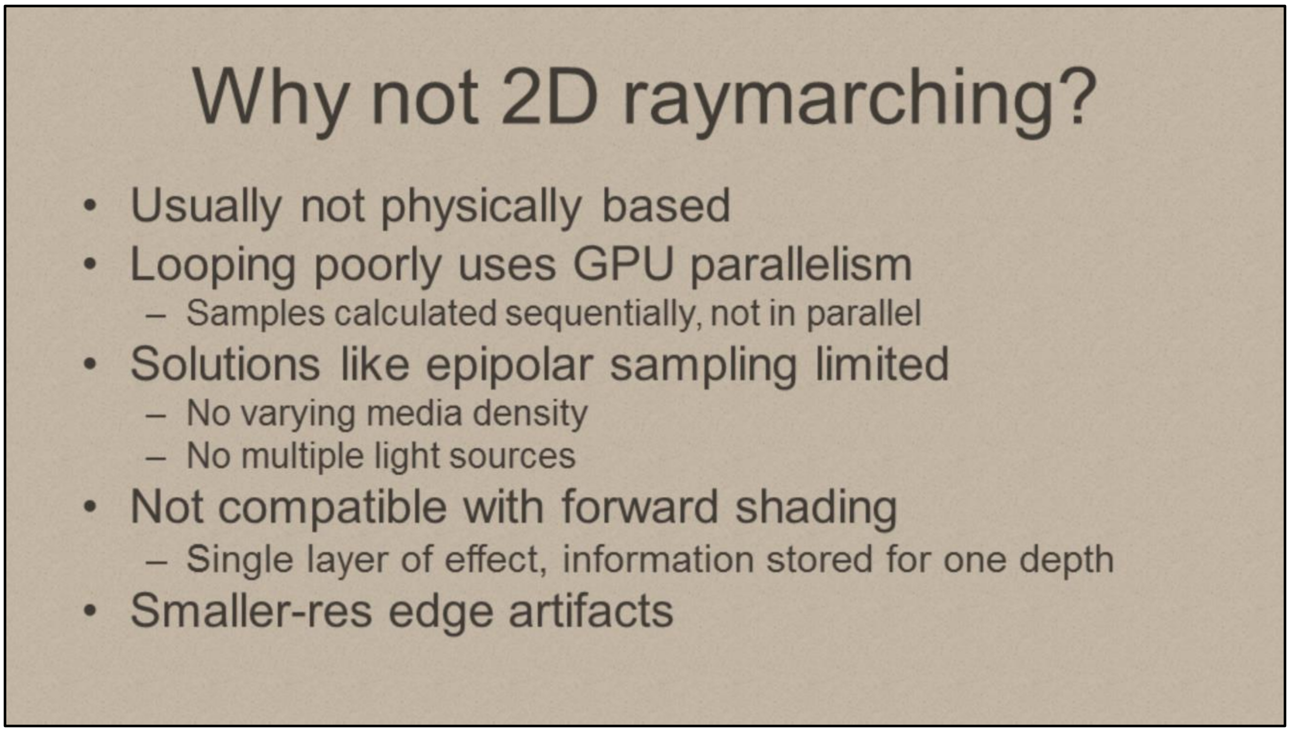
Approximations in games

- Post-effect based
- Ray-marching

Source: C. Wyman, S. Ramsey "Interactive Volumetric Shadows in Participating Media with Single-Scattering"

Third approach that gained popularity because of UE3 and CryEngine is post-effect and radial blur based screenspace effect. It can look very effective, but unfortunately disappears completely when light sources are not visible on screen.

Finally, there are ray-marching solutions. They work very well (especially with epipolar sampling extension), but usually are very short range and have their limitations – next slide.

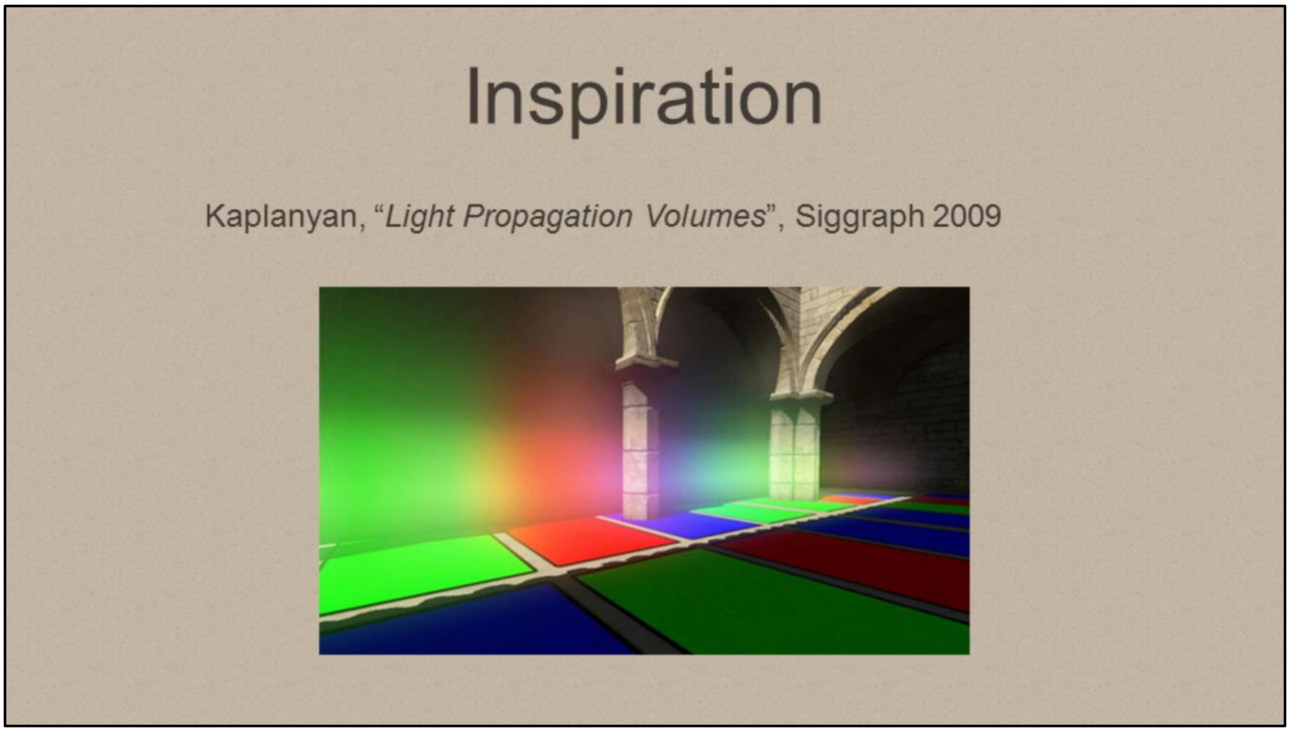


Why not 2D raymarching?

- Usually not physically based
- Looping poorly uses GPU parallelism
  - Samples calculated sequentially, not in parallel
- Solutions like epipolar sampling limited
  - No varying media density
  - No multiple light sources
- Not compatible with forward shading
  - Single layer of effect, information stored for one depth
- Smaller-res edge artifacts

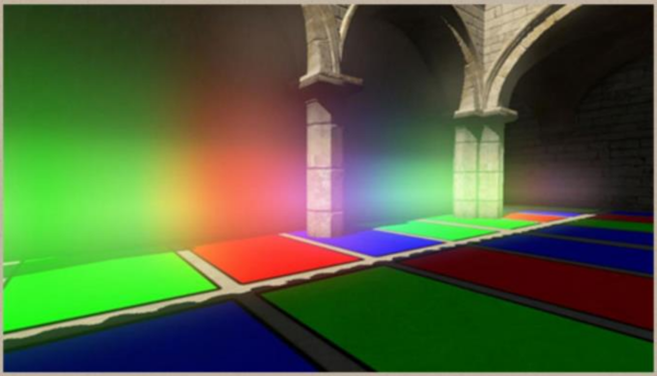
Raymarching volumetric shadows seemed like most modern solution, used by multiple other titles and potentially giving biggest visual improvement.
However, we noticed some limitations of regular 2D raymarching.

- Most of those implementations are for close range “volumetric shadows” / “light shafts”, not really physically based and suited for supporting long range fog. They often use simply artist-specified blending mode.
- They operate in loops, which is counter-productive on modern GPUs. Especially with modern architectures like AMD GCN which can launch thousands of thread waves it is a big waste of GPU power. It usually is about fetching information from big resolution shadowmaps, so latency hiding can be beneficial here.
- Some smart optimizations gaining popularity recently like epipolar sampling work under many constraints / limitations like lack of varying media density and no handling of multiple light sources. Every different light source requires different epipolar sampling scheme.
- Finally, all up to date implementations were designed as a post-effect and didn’t handle properly forward-shaded objects like transparencies or particles.
- All modern implementations of raymarching work in smaller resolutions. Usually signal is low-frequency so it isn’t a big problem on flat surfaces, but produces undersampling, aliasing and edge artifacts that are difficult to avoid.



Inspiration

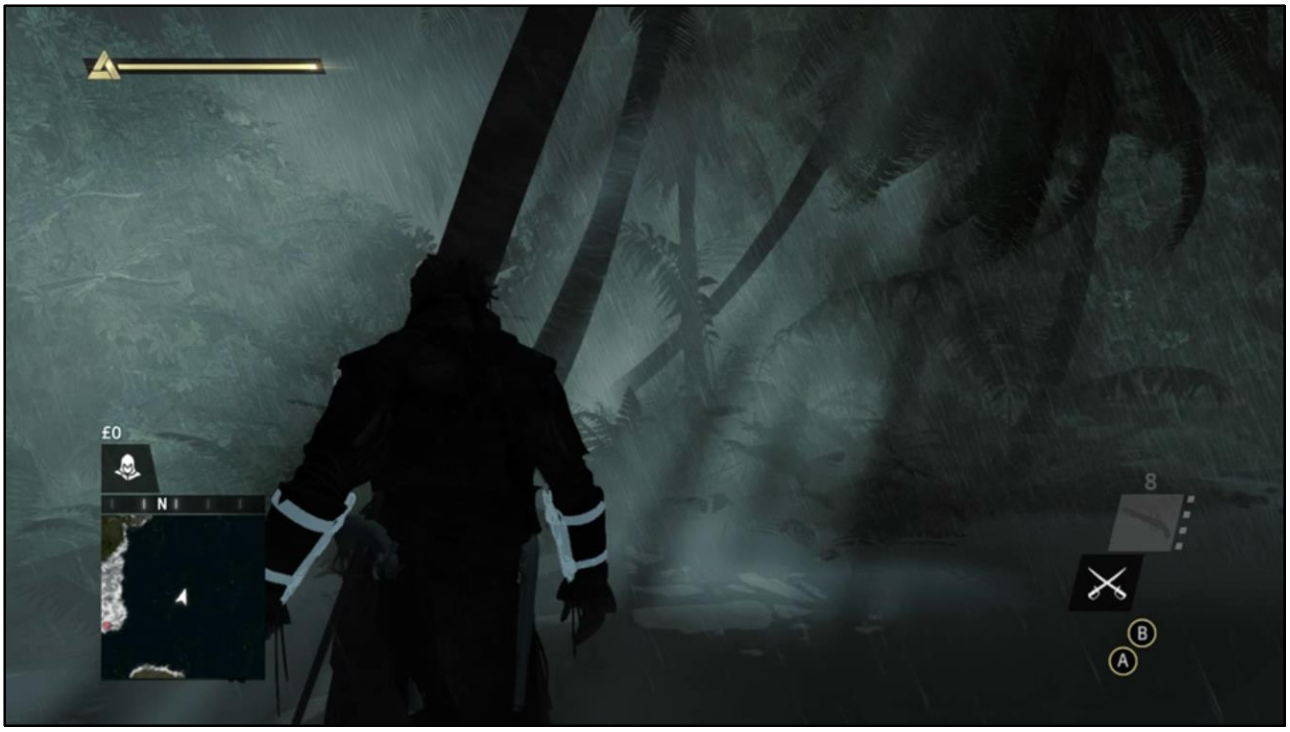
Kaplanyan, "*Light Propagation Volumes*", Siggraph 2009



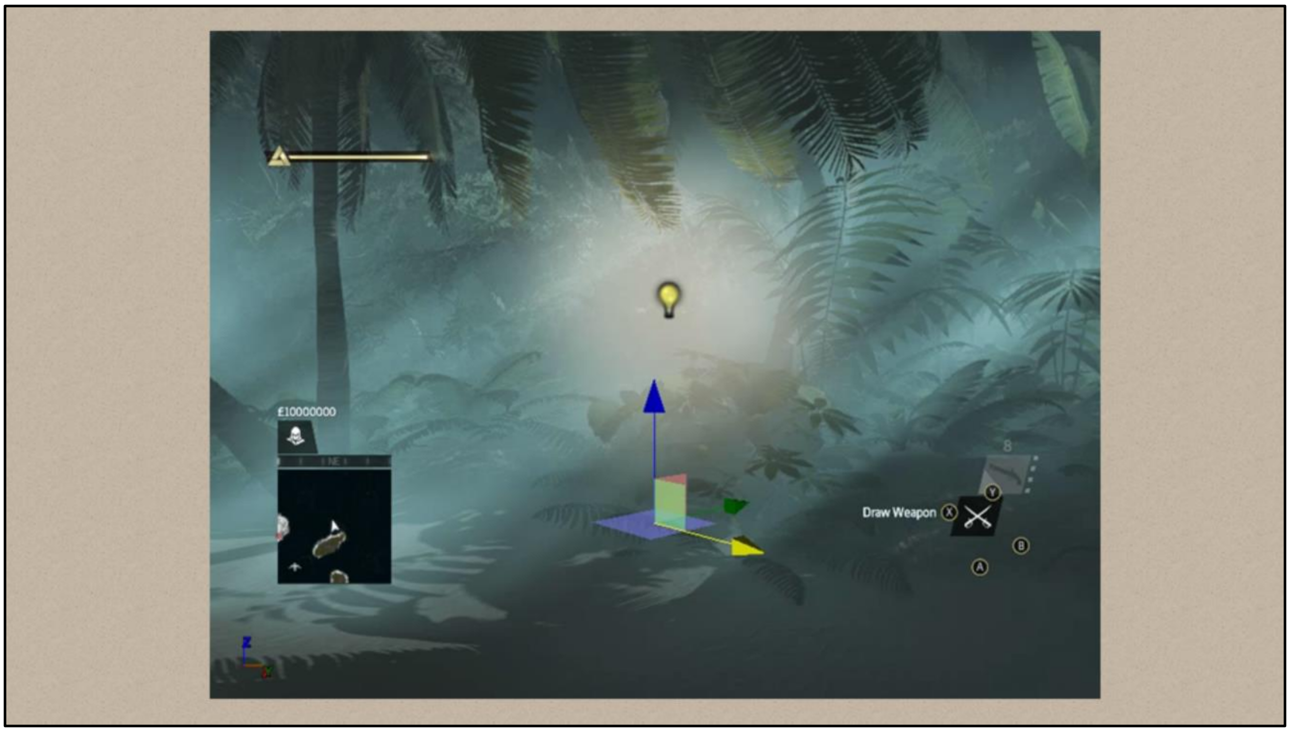
After prototyping numerous "classic" techniques, our biggest inspiration came from "Light Propagation Volumes" GI technique by Anton Kaplanyan presented at Siggraph 2009.
In technique summary and future work section, author mentions using lit volumetric texture – the result of injecting and propagating light to simulate GI – to compute participating media light transport.
This has advantage of performing raymarching just once, independent of number of light sources. Whole light transport in participating media can be unified. We believed that if we added simple shadowing term, we would gain light shafts/god rays just for the cost of calculating the shadowing.
We decided to follow that path.

Volumetric Fog

This inspiration together with raymarching solutions guided us towards developing a new solution that we called “Volumetric Fog”.

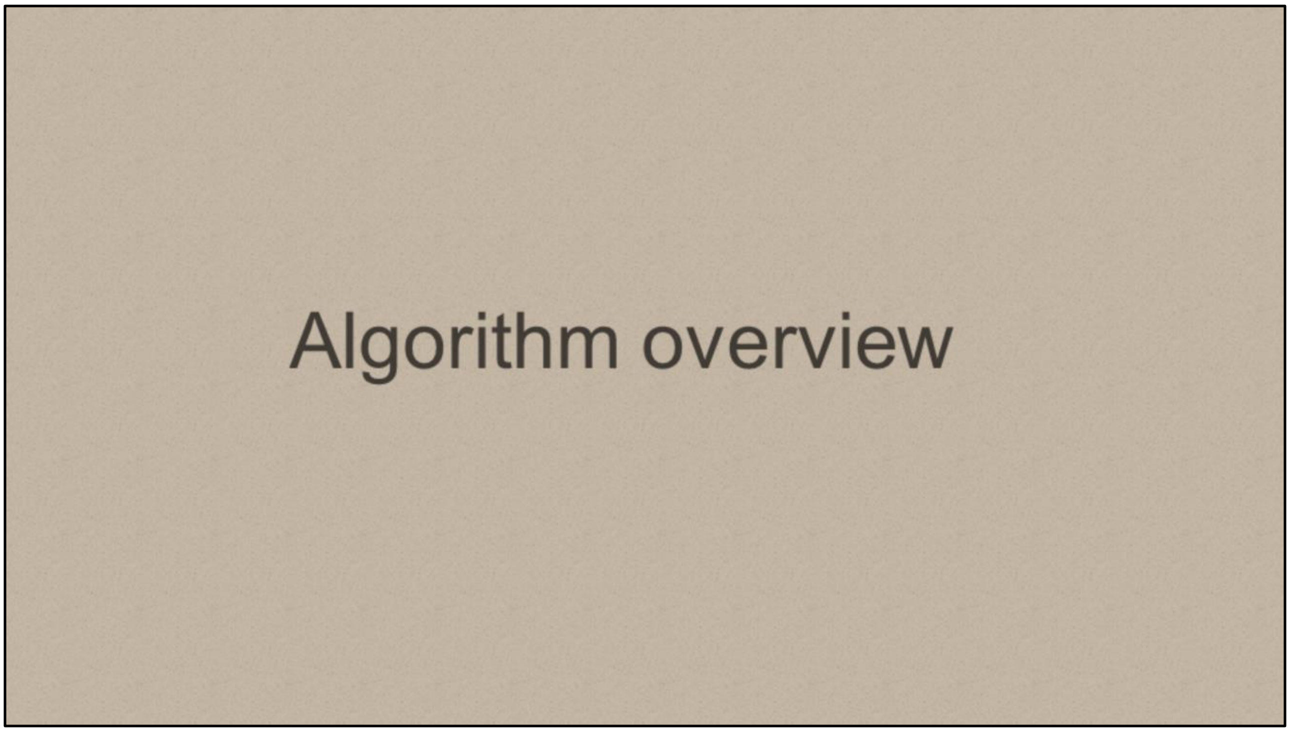


VIDEO

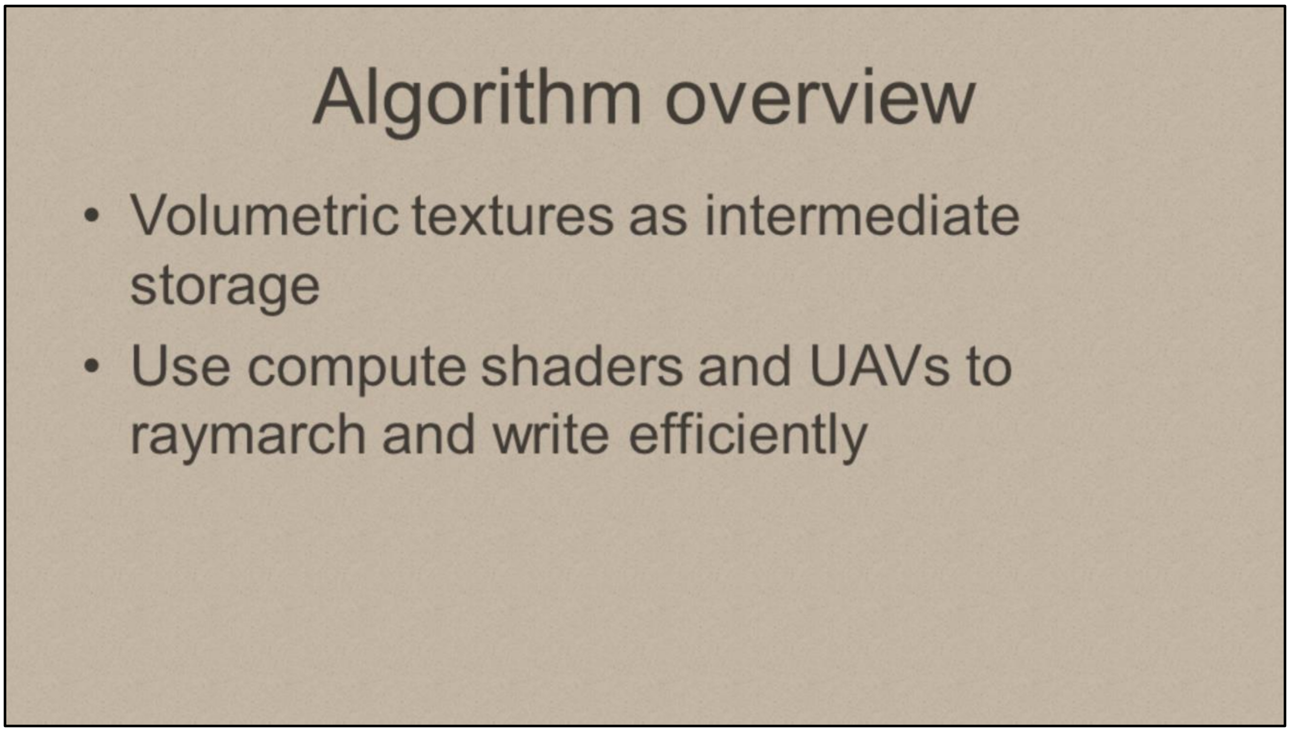


VIDEO

It worked coherently, correctly and perfectly with multiple light sources



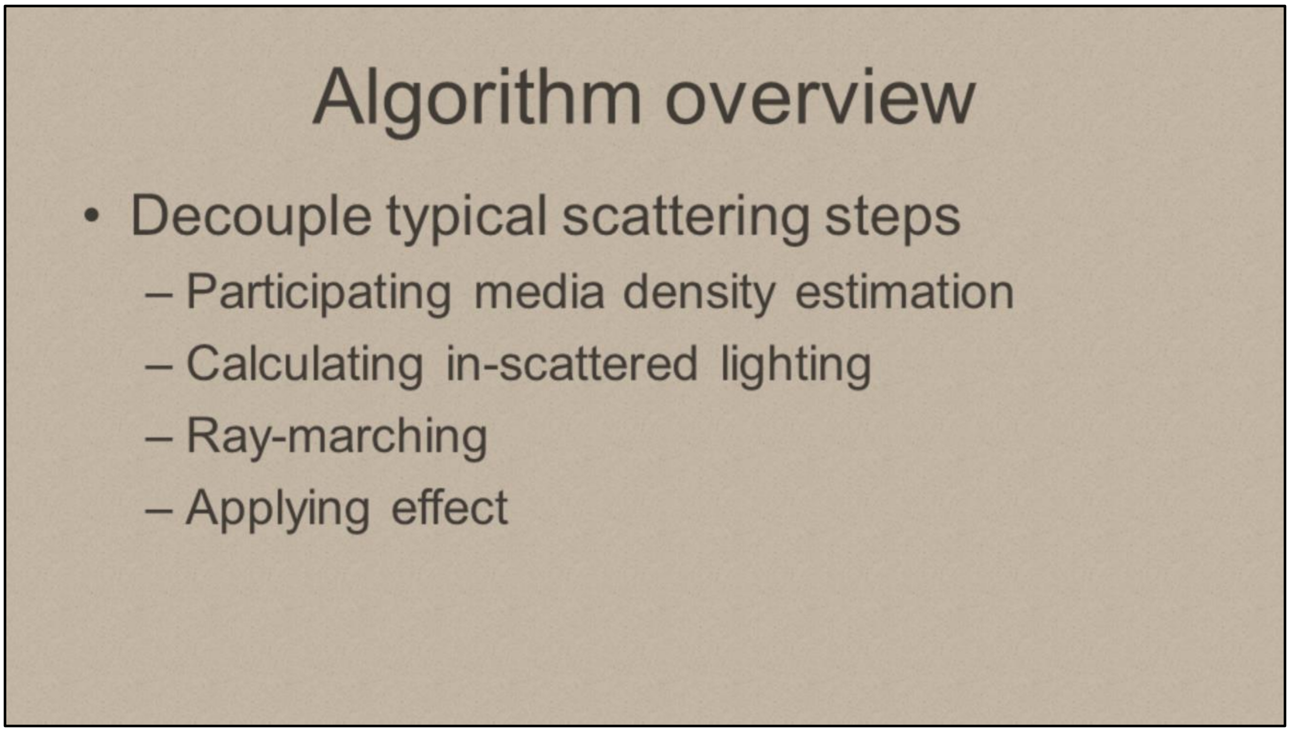
Algorithm overview



Algorithm overview

- Volumetric textures as intermediate storage
- Use compute shaders and UAVs to raymarch and write efficiently

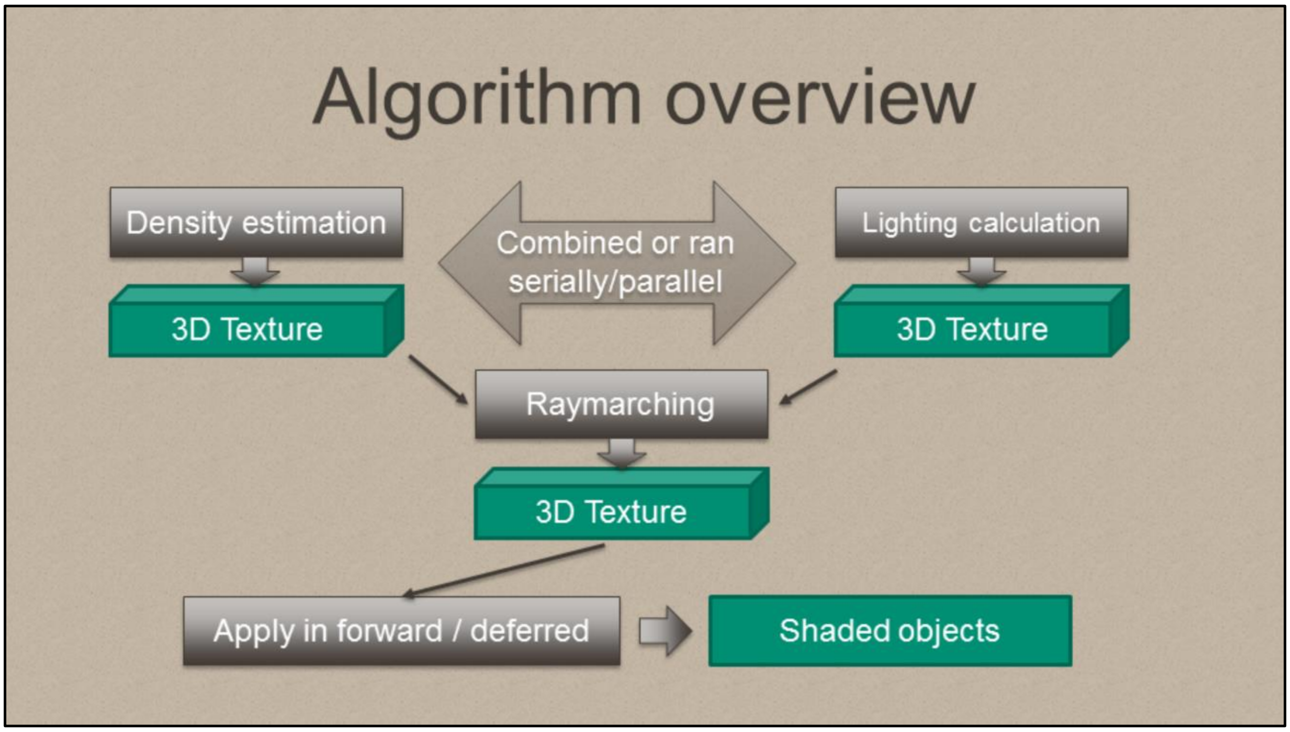
To allow efficient splitting of passes and launching them separately, we used volumetric textures as storage for intermediate and partial results.
We used compute shaders and 3D texture UAVs to write volumetric data in both efficient and very convenient manner.



Algorithm overview

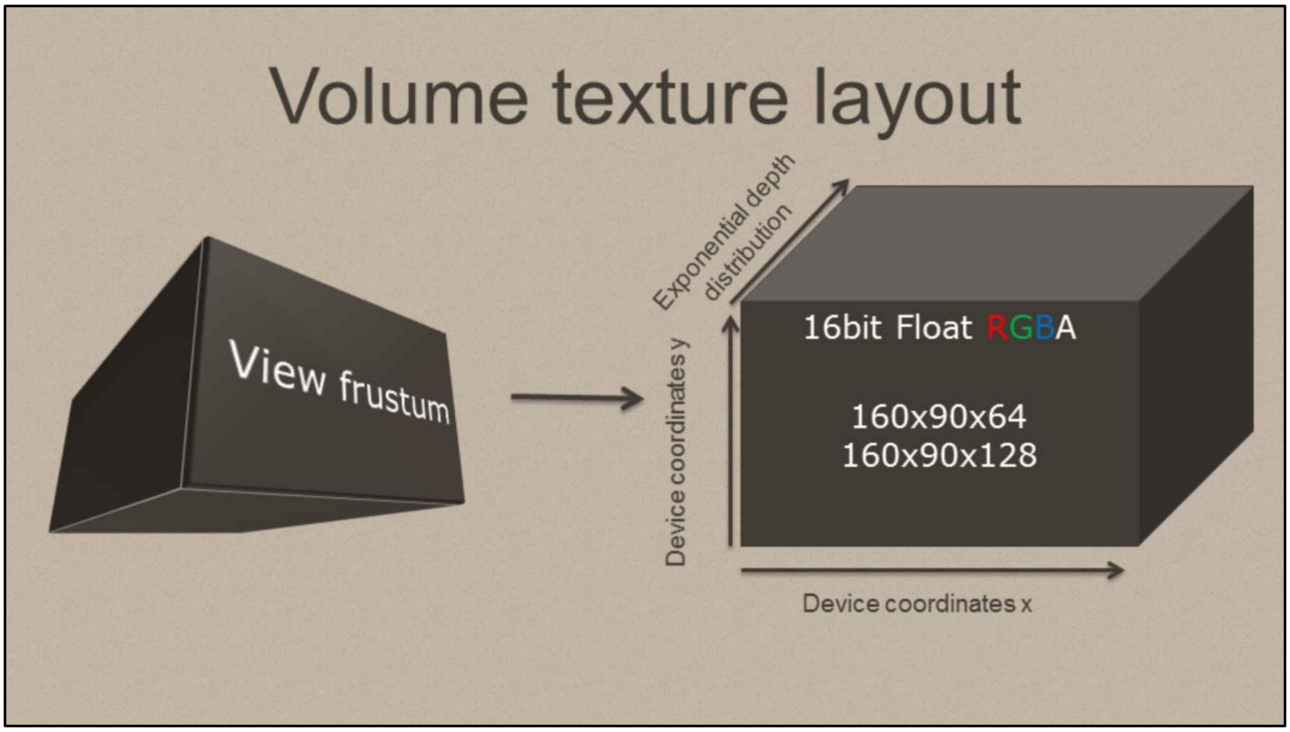
- Decouple typical scattering steps
  - Participating media density estimation
  - Calculating in-scattered lighting
  - Ray-marching
  - Applying effect

Key idea of our algorithm is to decouple and parallelize typical steps of raymarching algorithms and launch them separately.
This way we achieve parallelism and possibility of swapping and adjusting every element of the algorithm separately.


Algorithm overview

We can see on this diagram how we split our algorithm in multiple passes.

1. First and most important part is performing lighting and shadowing calculations for every volume cell.
2. In parallel (same or parallel pass) or serial, we estimated and animated the participating media density.
3. Then using this information stored in volume textures, we performed 2D raymarching through the volume and stored the result in volume slices.
4. And finally using pixel shaders, applied information on screen for forward or deferred shaded objects.

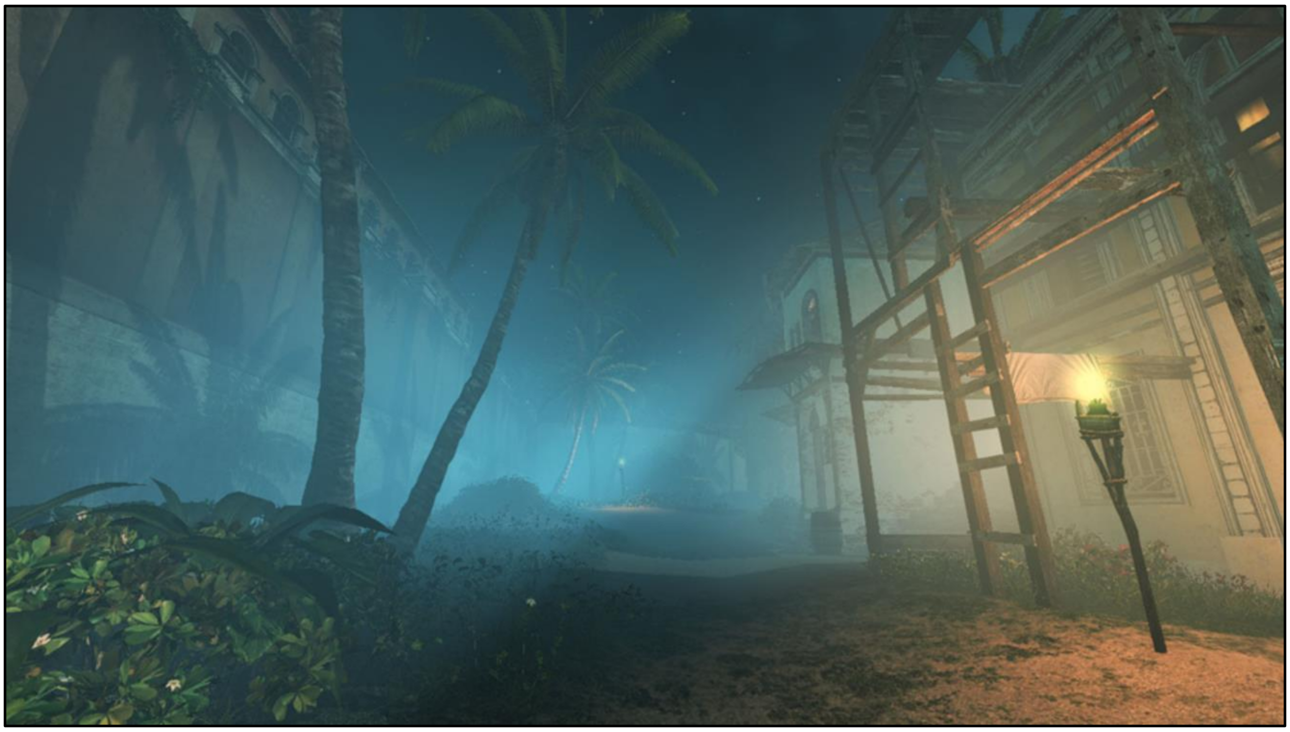


Volume texture layout

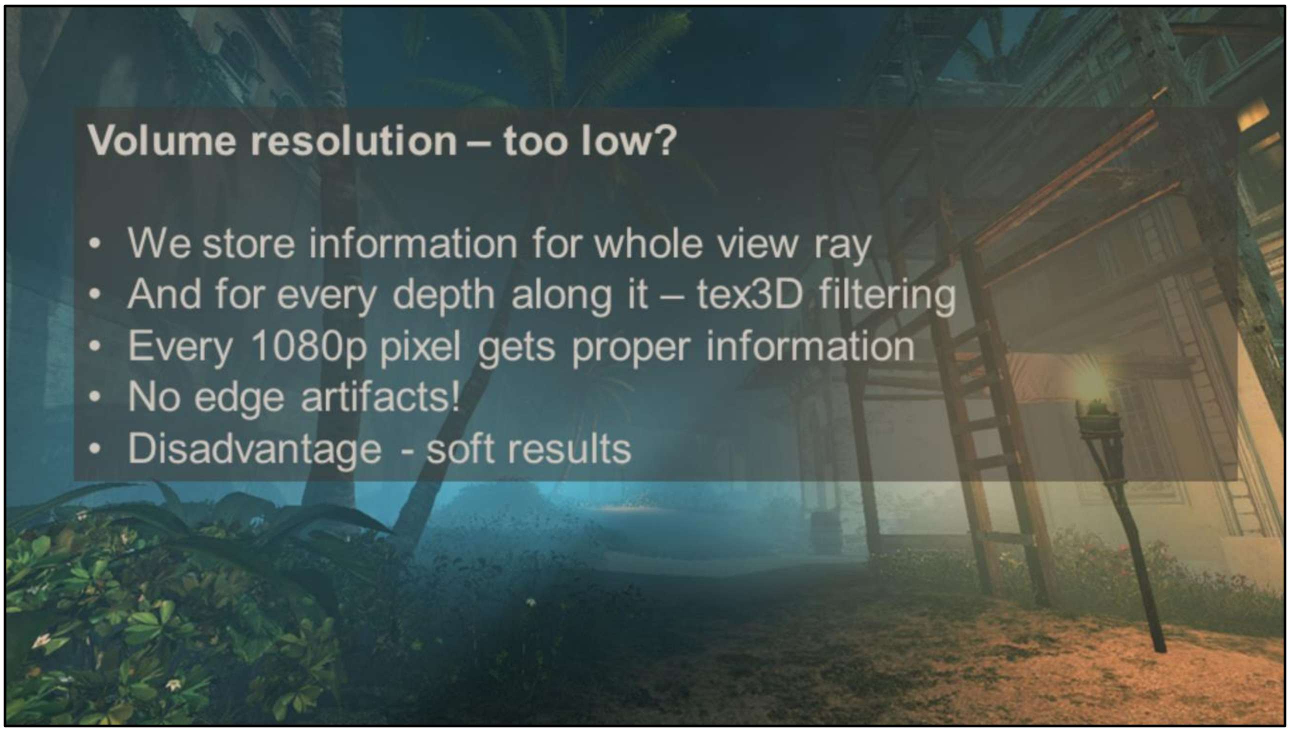
How does our volumetric fog intermediate storage texture look like?

At first, we tried simplest data layout –cuboid aligned to world space coordinates. It provided multiple benefits – for example very easy temporal filtering, but raymarching through such volume required multiple samples, was slow and produced aliasing artifacts.

Instead we decided to use layout aligned with the camera frustum. We directly map frustum to cuboid using normalized device coordinates in width x height axis and for depth slices we use exponential depth distribution.
We tried various depth distributions and ended up with one concentrated near the camera – this is where we need the most precision and aliasing artifacts show up easily.
Aligning our texture with the viewing frustum has several disadvantages such as being prone to temporal aliasing and flickering, but raymarching is just a parallel scan through depth slices.
We used volumes sized 160x90x64 or 160x90x128 depending on the platform. It provides fixed cost of almost all of the passes, not dependent on the screen resolution.
In 160x90x64 layout number of texels is equal to number of texels in 720p surface – but for every cell we perform lighting calculations just once.
Effect range depends on artist defined settings, but we distances between 50 and 128 meters – to keep long distance fog consistent with current gen art direction – but there are no reasons why it wouldn't be possible to do longer range fog (using exponential depth distribution or cascaded approach)



Is this volumetric texture resolution enough?



Volume resolution – too low?

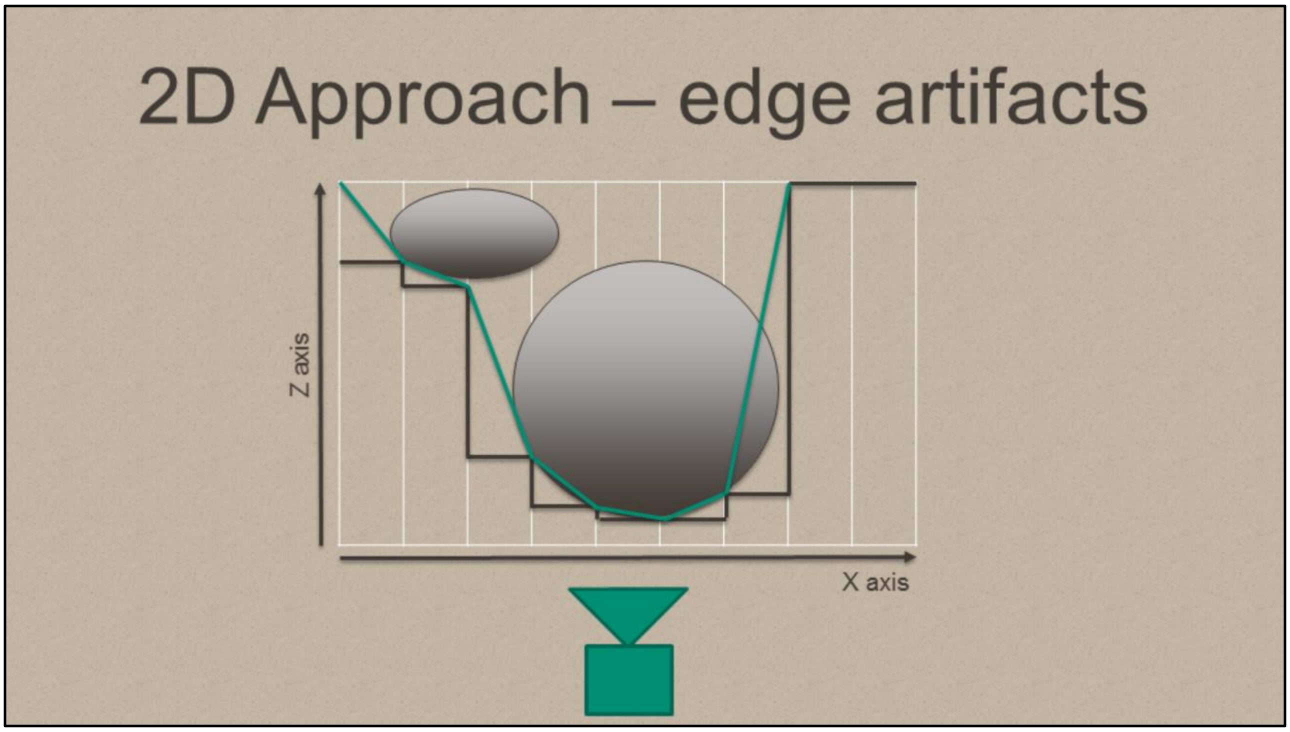
- We store information for whole view ray
- And for every depth along it – tex3D filtering
- Every 1080p pixel gets proper information
- No edge artifacts!
- Disadvantage - soft results

The resolution of our volume textures may seem extremely low, but it is sufficient:

1. We store low frequency information for every depth stored along the ray.
2. Due to the fact that when applying effect, we use quadrilinear filtering on volumetric data, we are unable to see single texels of volume texture.
3. Every target pixel receives information about proper and precise depth from its native resolution.
4. Perspective correction and volume shape makes sure that information is distributed correctly.
5. We don’t get the edge artifacts on depth discontinuities (described on the next slide).

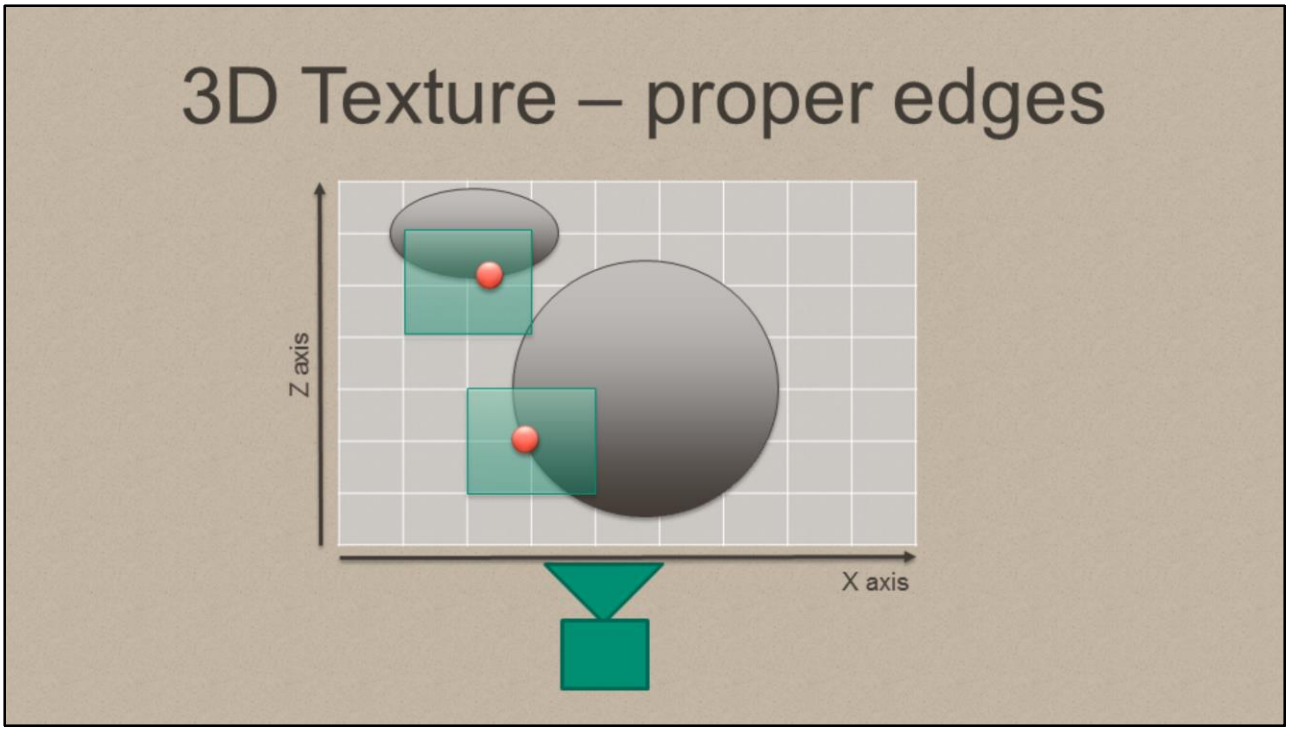
Obviously, produced effect is very soft and is missing high frequency geometric details, but it fits our art direction and is quite similar to reality (because in the real atmosphere multiple scattering effect takes place and softens the look of light shafts a lot).

Final advantage of not relying on scene depth is possibility of computing this pass in parallel with regular scene rendering as soon as shadow-maps are ready – for example using asynchronous compute on AMD hardware with console APIs or Mantle.

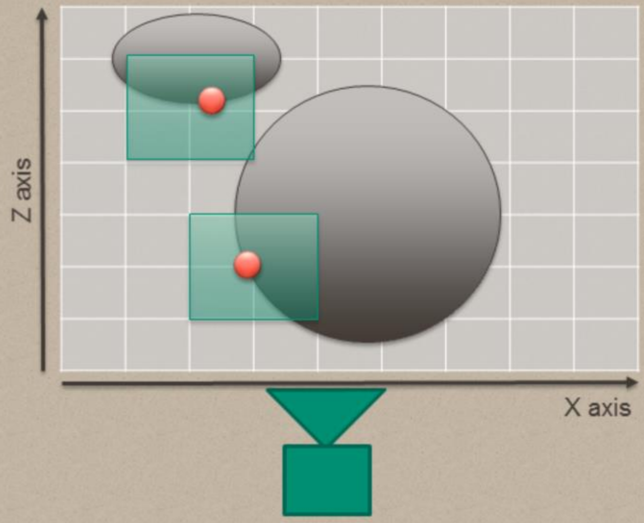


2D Approach – edge artifacts

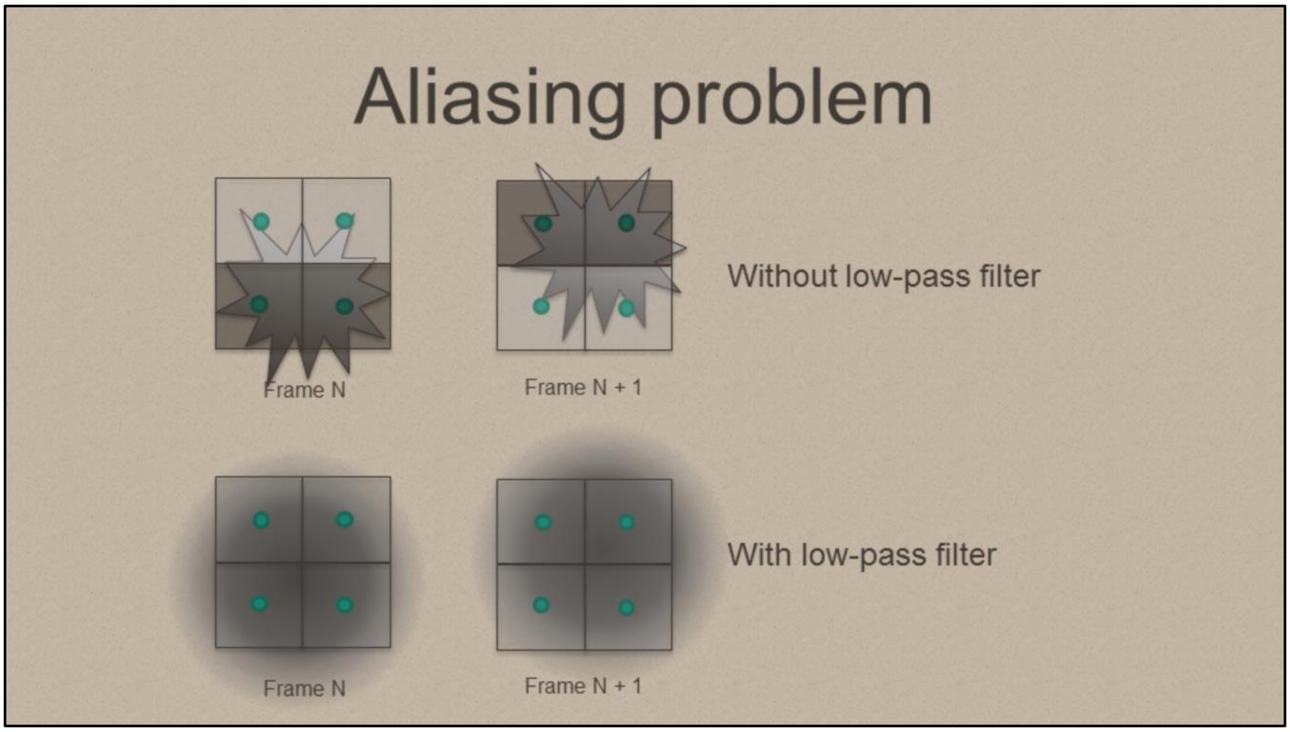
When doing regular 2D low resolution rendering, the main problem is behavior on edge discontinuities. For low resolution post-effect calculations we must pick one specific depth out of many potential depth fragments – this way some final shaded fragments will have improper information – either interpolated or selected from the neighbourhood (bilateral upsampling).



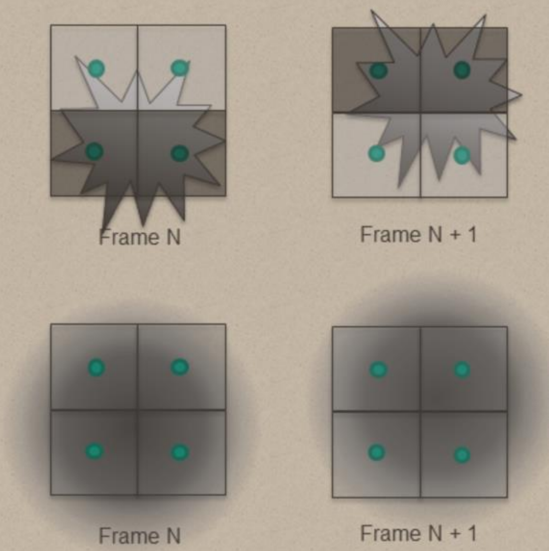
3D Texture – proper edges



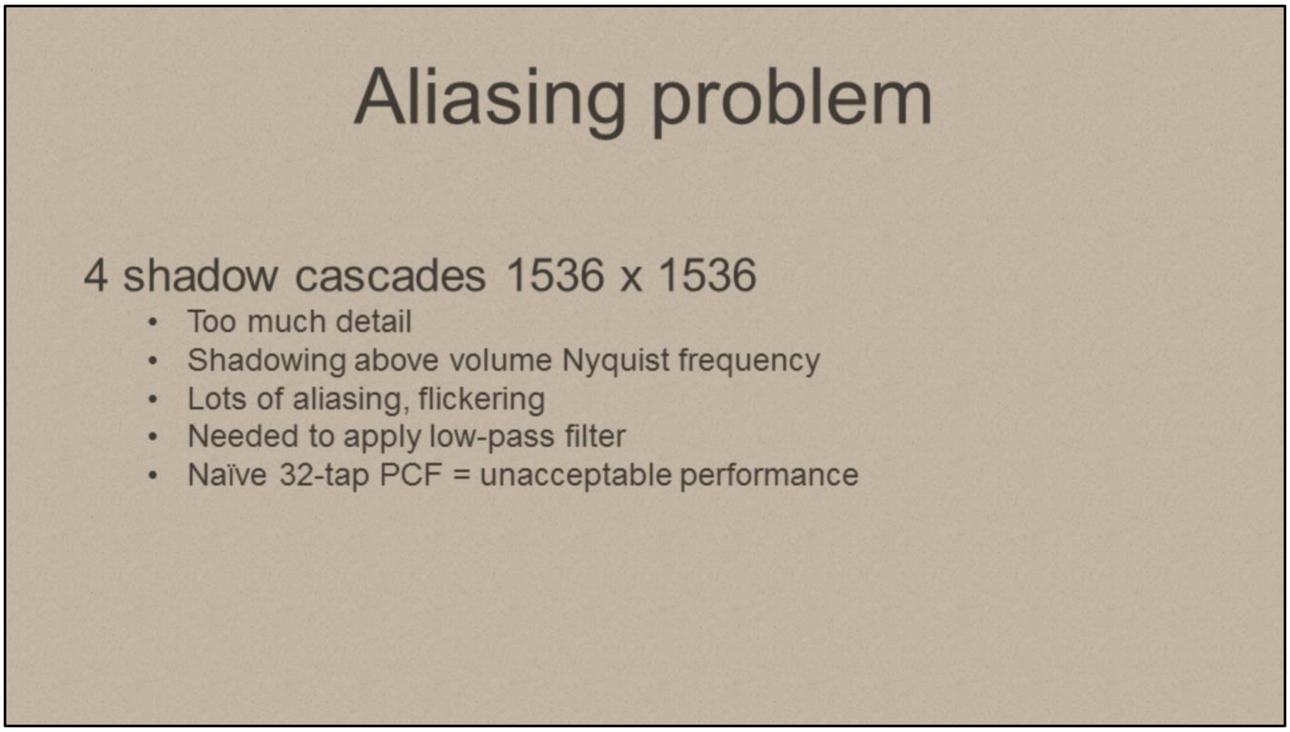
Fortunately, with 3D textures and 3D interpolation, we don't have such problems.
Every full resolution fragment and its depth gets proper, piecewise linear interpolation of calculated function.
While it is still in low resolution and can be "jagged", it doesn't have edge discontinuity artifacts.



Aliasing problem



We can see on this diagram how low-pass filtering removes temporal aliasing / flickering problem of high frequency source information.


Aliasing problem

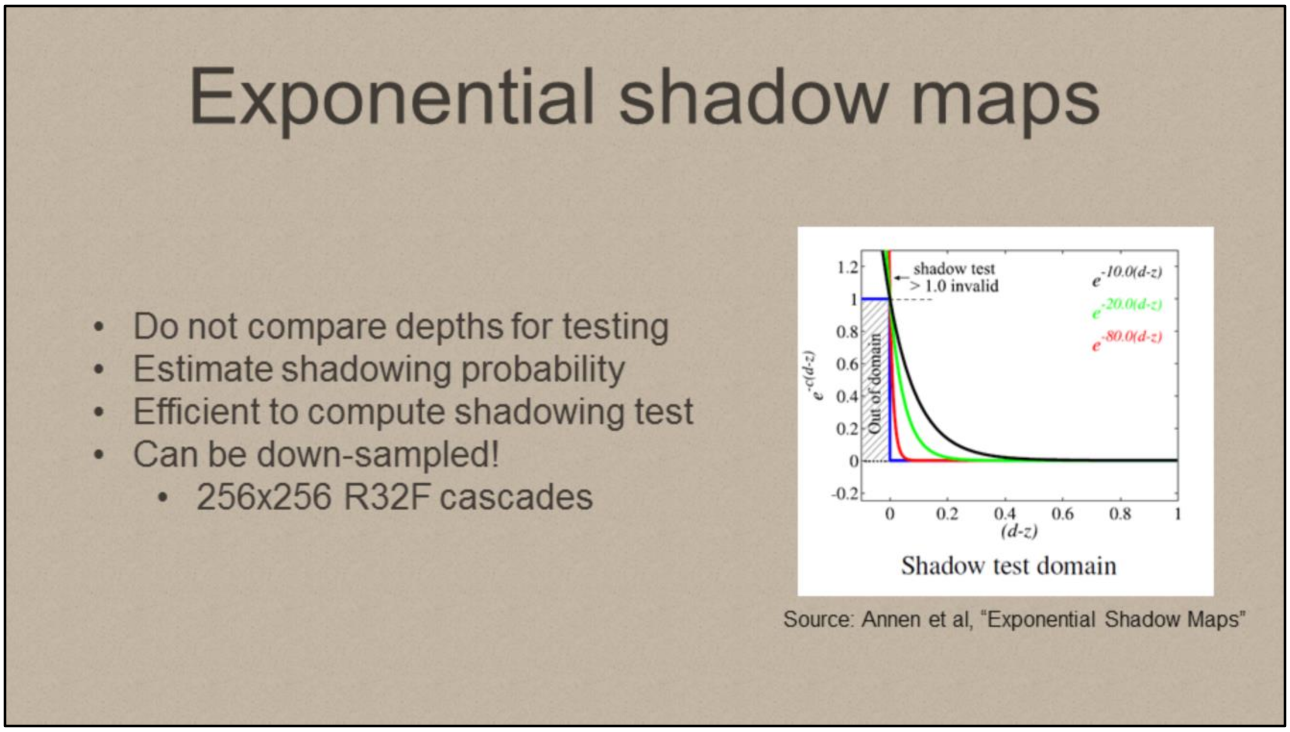
4 shadow cascades 1536 x 1536

- Too much detail
- Shadowing above volume Nyquist frequency
- Lots of aliasing, flickering
- Needed to apply low-pass filter
- Naïve 32-tap PCF = unacceptable performance

The first step of our algorithm is preparing the shadowmaps to be used fog calculating sun scattering shadowing.

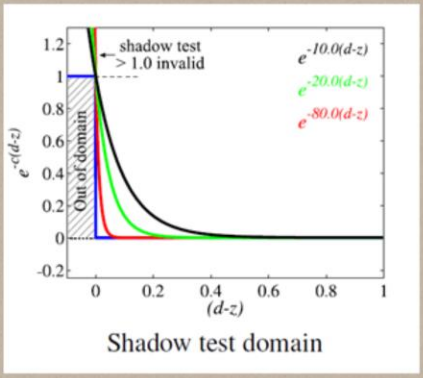
Why do we need it?

Our regular shadowing cascades were very high resolution (4 cascades, 1536x1536 or 1k x 1k depending on the platform) and had dense, high resolution information in them.
It was way too much detail for us, especially for close ranges and in first two cascades, condensed in first couple meters.
For smooth and approximate volumetric fog we needed something of much smaller resolution to reduce any flickering / aliasing artifacts from moving foliage etc.
First implementations using wide kernel PCF had very poor performance and still some flickering and aliasing artifacts.



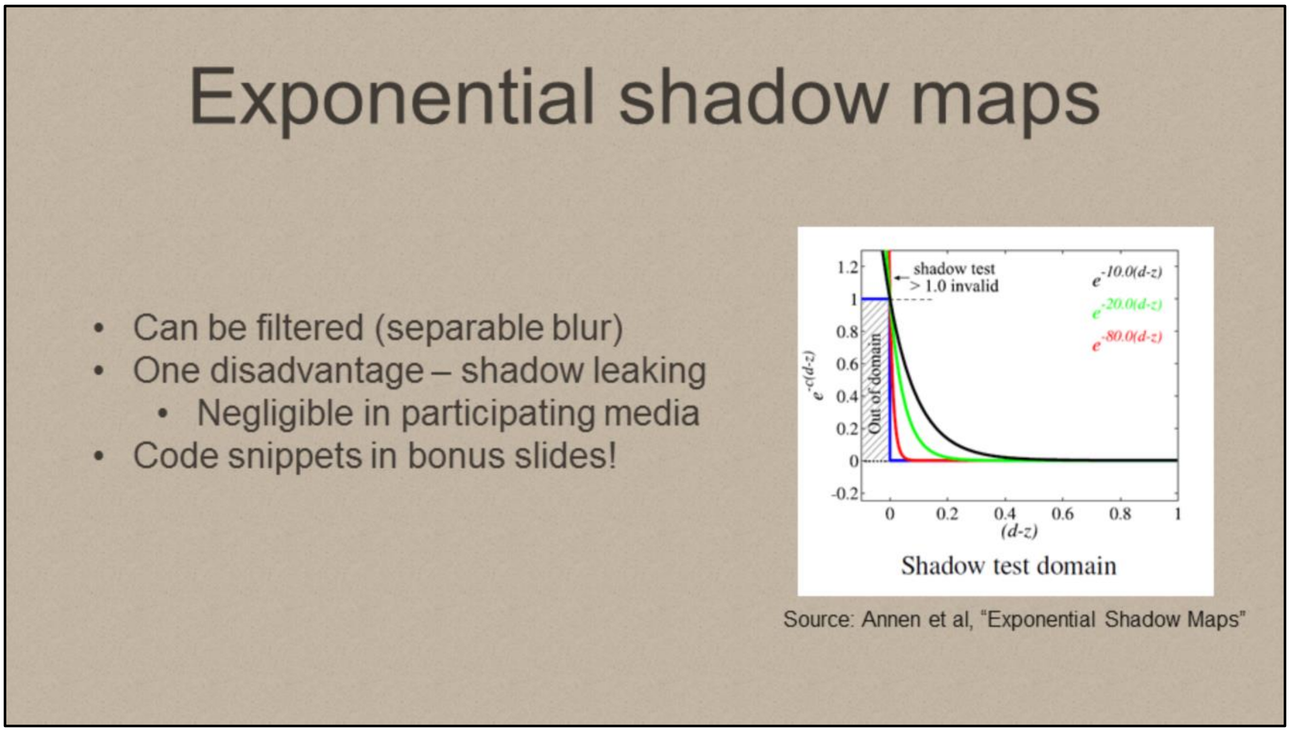
Exponential shadow maps

- Do not compare depths for testing
- Estimate shadowing probability
- Efficient to compute shadowing test
- Can be down-sampled!
  - 256x256 R32F cascades



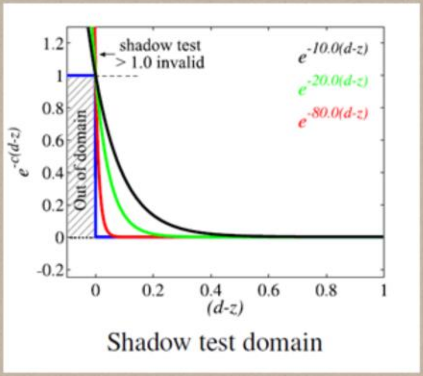
Source: Annen et al, "Exponential Shadow Maps"

The solution came from Exponential Shadow Maps algorithm.
It is simple algorithm that allows filtering of estimated shadowing probability and therefore downsample the shadowing function.
It is trivial and very efficient to compute shadowing test. Some simple code snippets of ESM are in bonus slides.



Exponential shadow maps

- Can be filtered (separable blur)
- One disadvantage – shadow leaking
  - Negligible in participating media
- Code snippets in bonus slides!

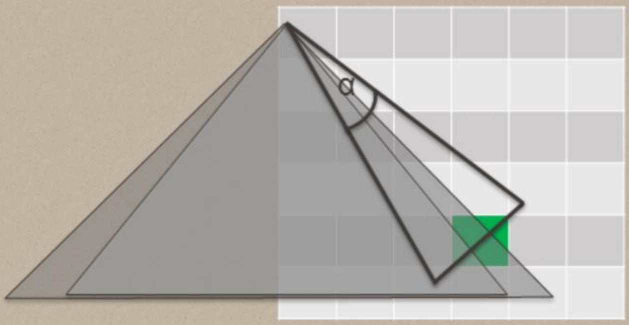


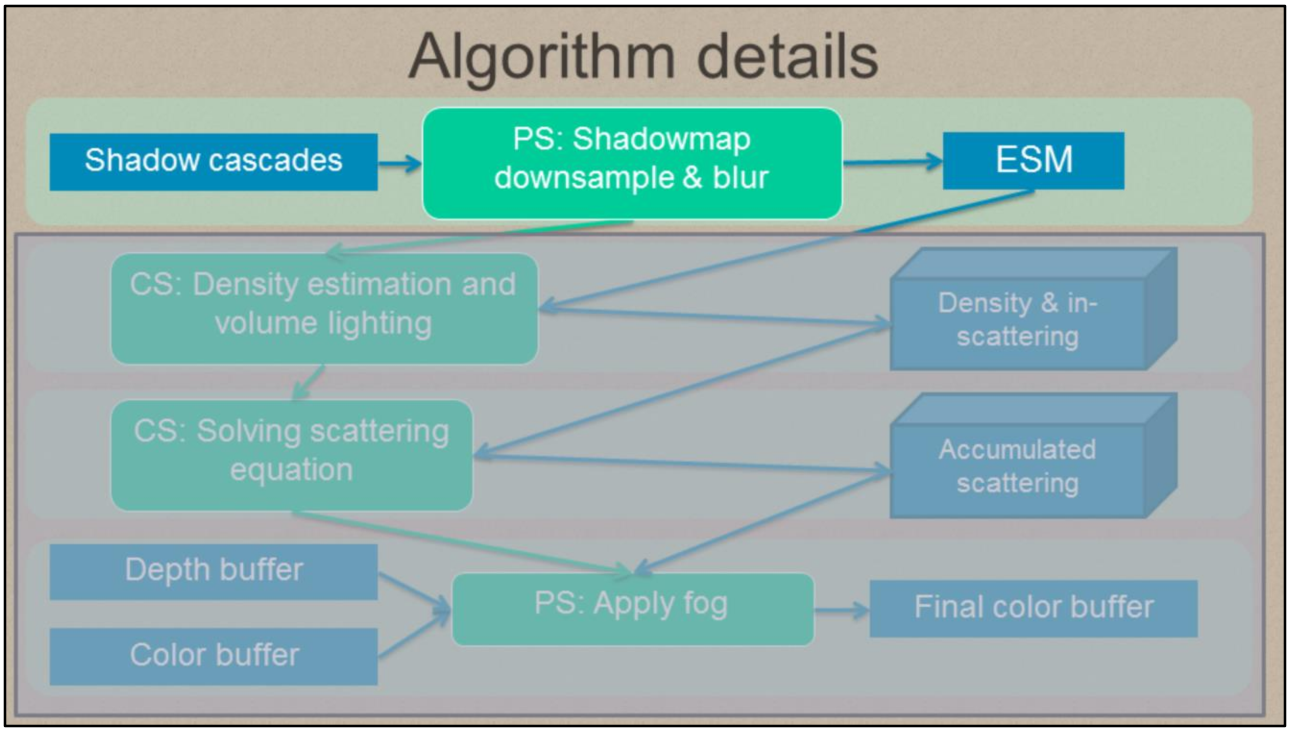
Source: Annen et al, "Exponential Shadow Maps"

First we downsample our cascaded shadowmaps four times (target R32F 1024 x 256 texture). During downsampling we calculate exponential shadowing function (see Exponential Shadow Mapping, an extension to Variance Shadow Mapping using exponential function instead of Chebyshevs inequality).

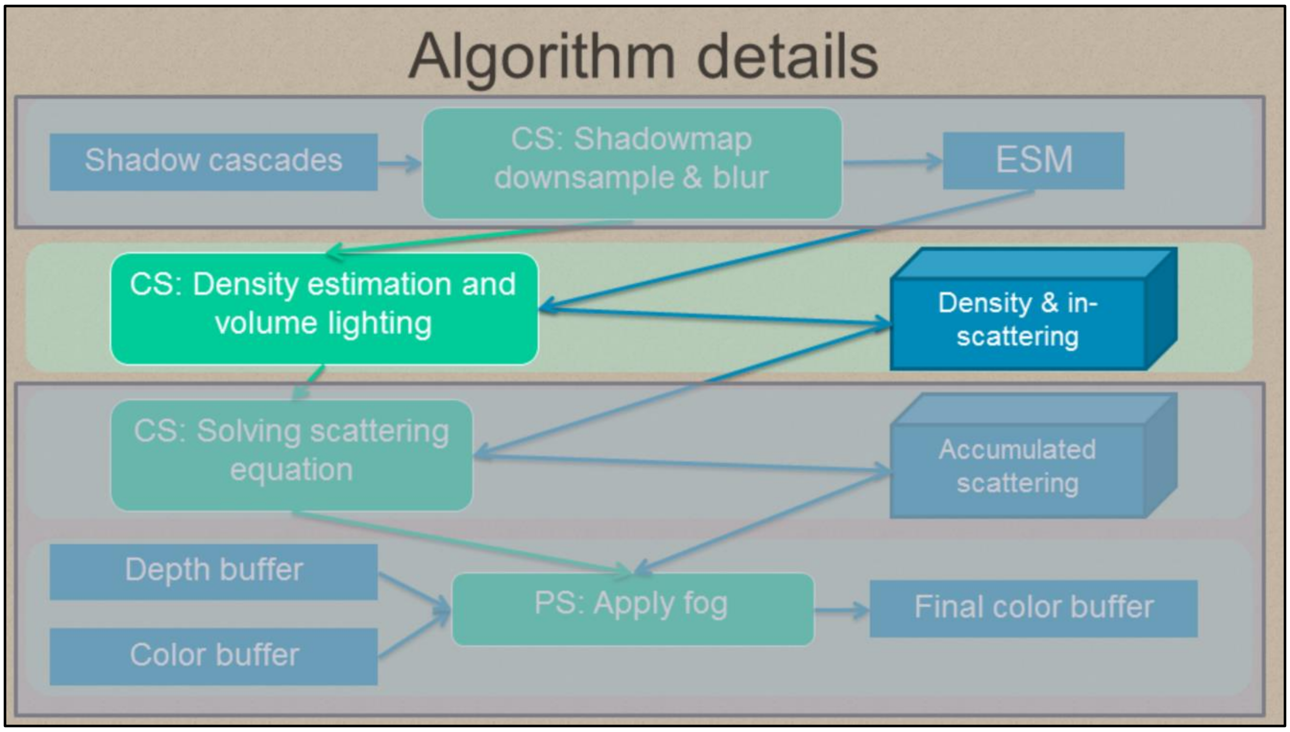
We also do additional separable box filter (as two separate steps) during this pass to make shadows softer and remove aliasing artifacts.
One disadvantage of ESM making it not practical for regular rendering – shadow leaking – wasn't noticeable in our case in participating media.

Spotlights antialiasing

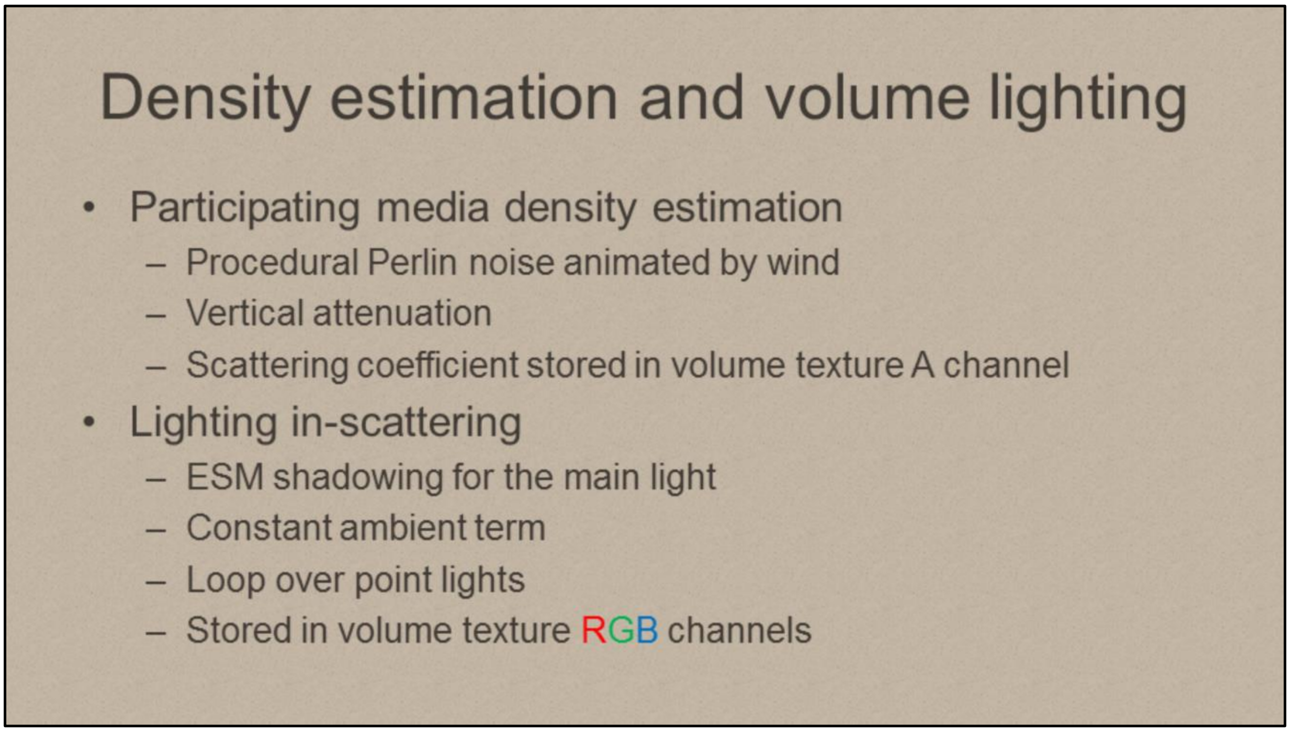



Algorithm details

Therefore the first step of our algorithm is downsampling the shadowing information.


Algorithm details

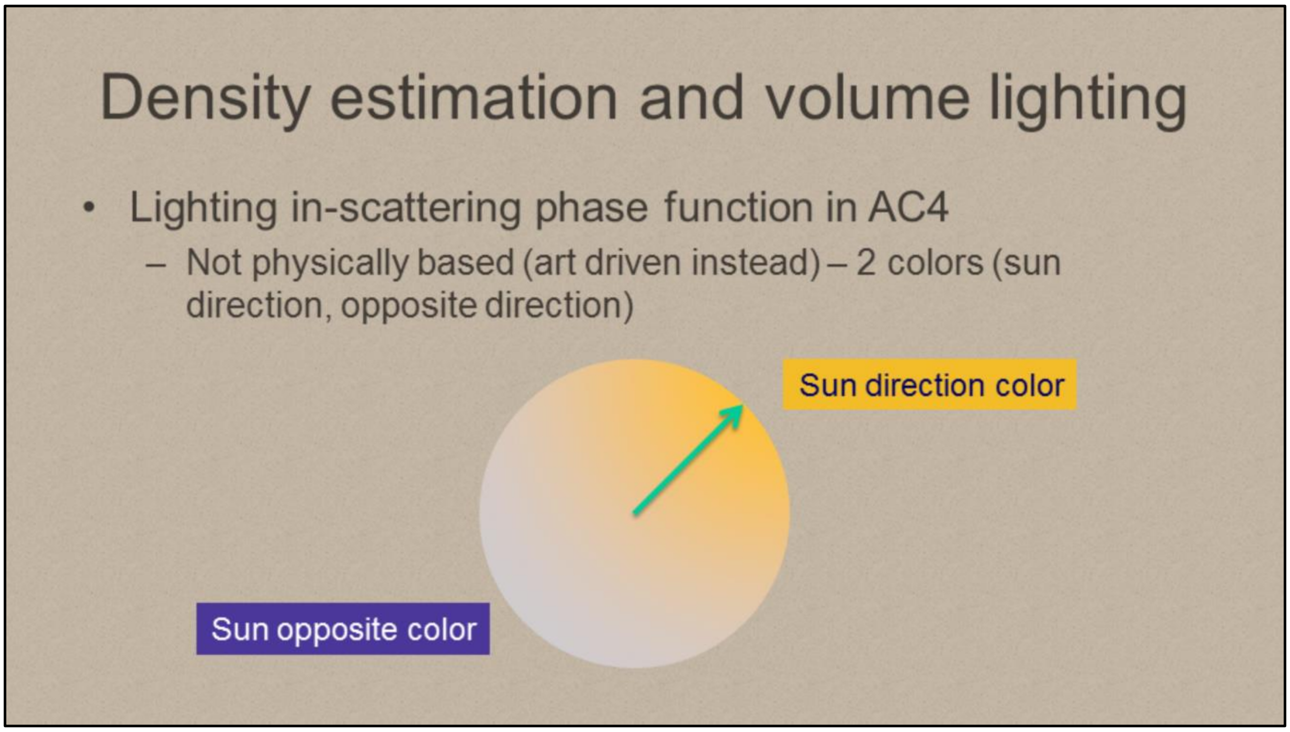
When we have downsampled shadow information ready, we can proceed with participating medium lighting calculations.


Density estimation and volume lighting

- Participating media density estimation
  - Procedural Perlin noise animated by wind
  - Vertical attenuation
  - Scattering coefficient stored in volume texture A channel
- Lighting in-scattering
  - ESM shadowing for the main light
  - Constant ambient term
  - Loop over point lights
  - Stored in volume texture RGB channels

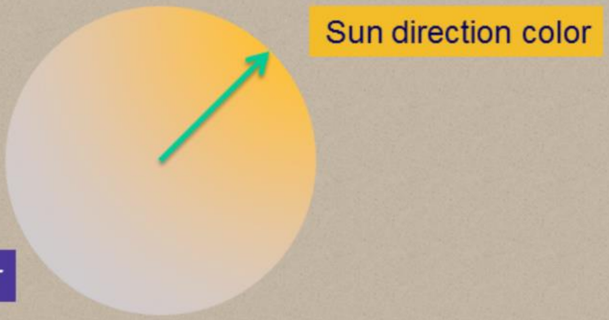
We combined density and lighting calculations due to a bit smaller bandwidth usage and same desired resolution, but they can be split and totally decoupled.
Density calculation is pretty straightforward, it is just one octave of Perlin noise that is animated by wind. We tried using multiple octaves, but in the end difference was rather subtle for added cost.
We also calculated vertical media density attenuation, as usually heavy particles such as steam water particles tend to gather around the ground level with exponential distribution.
The scattering coefficient got stored in the alpha channel of volume texture.

For lighting part we simply accumulated lighting from main light (sun/moonlight), constant ambient term and multiple dynamic point lights that were intersecting with view frustum and marked by artists as lights affecting the atmosphere.
We used mentioned Exponential Shadow Maps shadowing technique to get shadows from the main light.
We stored lighting information modulated by density in RGB channel of volumetric texture.

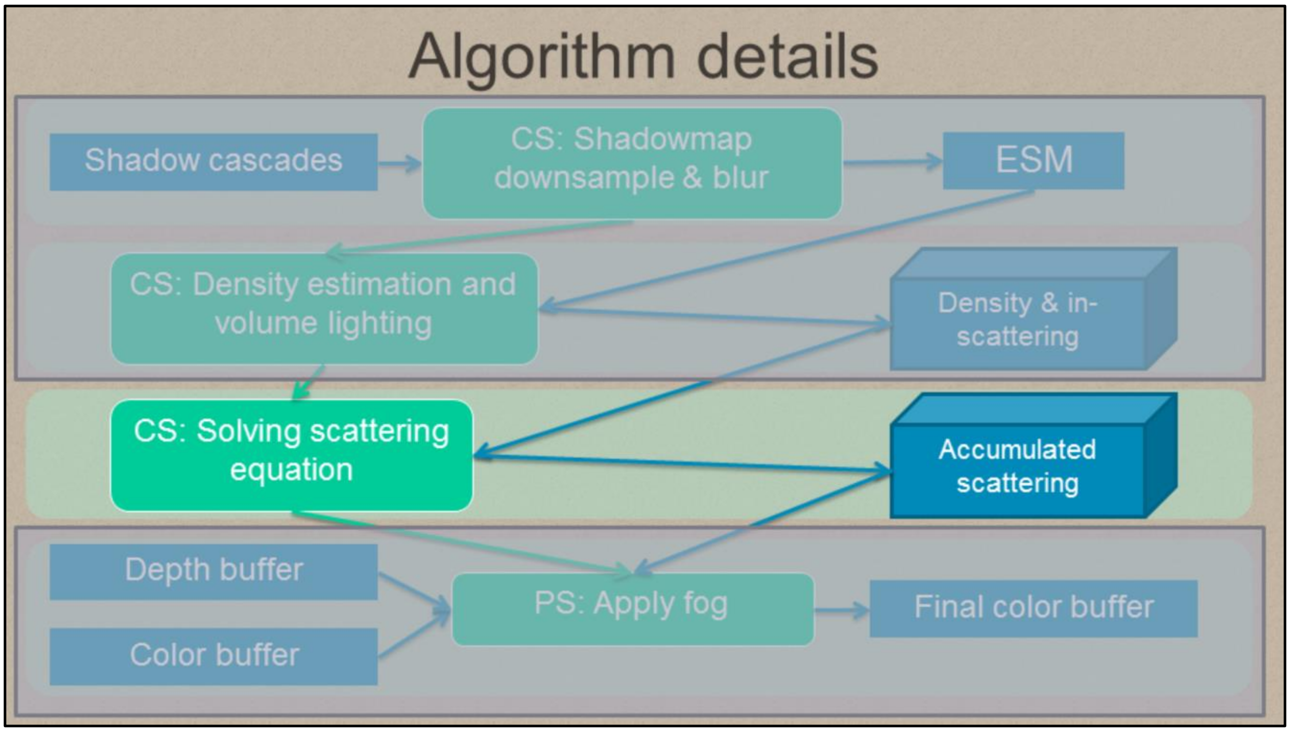


Density estimation and volume lighting

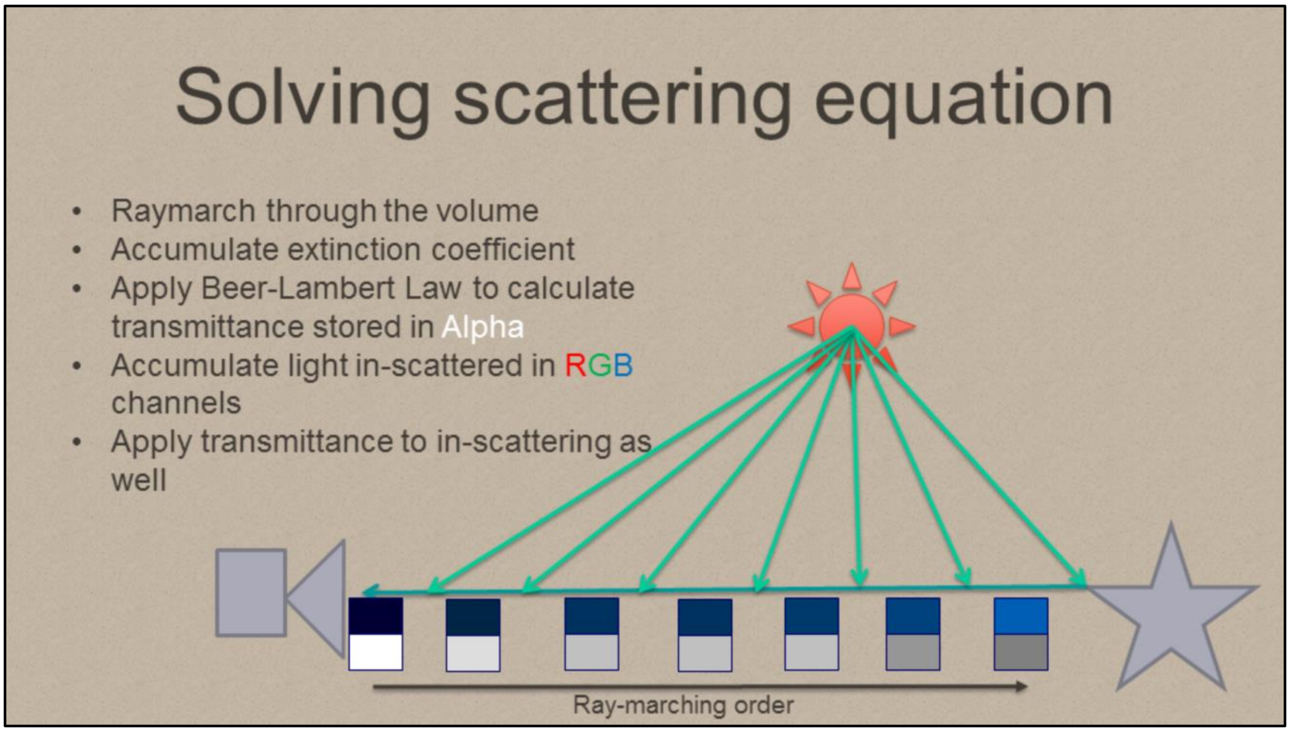
- Lighting in-scattering phase function in AC4
  - Not physically based (art driven instead) – 2 colors (sun direction, opposite direction)



On AC4 we didn't have any physically-based phase functions. Lighting color gradient (oriented towards direction of sun) was purely art-driven.
We had 2 colors of phase function – in direction of the sun and in opposite direction, apart from that with fully isotropic shape.
Obviously, any phase function can be applied in this pass to achieve more physically-correct effect. (described later)


Algorithm details

Third step is solving the scattering equation by linearly marching through the fog volume and performing a numerical calculus solution.



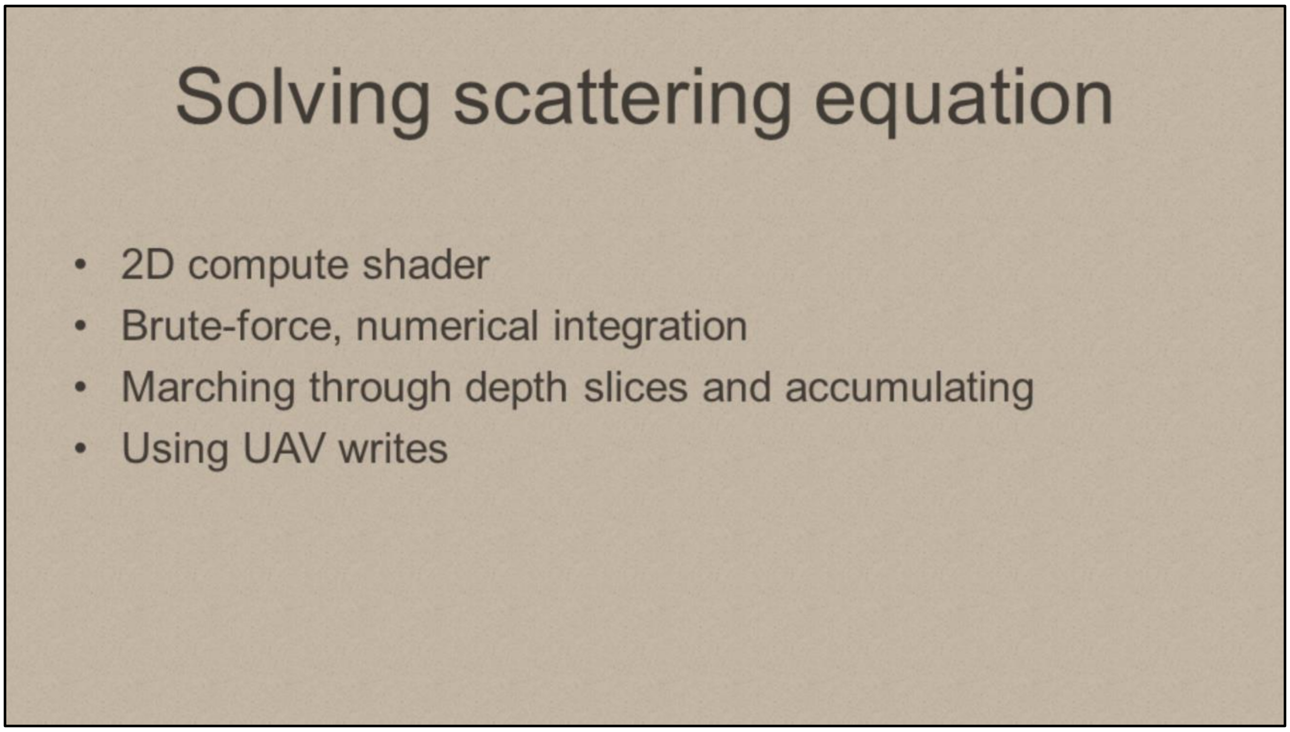
Solving scattering equation

- Raymarch through the volume
- Accumulate extinction coefficient
- Apply Beer-Lambert Law to calculate transmittance stored in Alpha
- Accumulate light in-scattered in RGB channels
- Apply transmittance to in-scattering as well

So how do we solve this scattering equation?

The out-scattering effect is described by Beer-Lambert law, exponential falloff function of density integral for given distance.
For the in-scattering it is simple sum of in scattered lighting so far (taking distance based out scattering into account).

We have already our in-scattering values calculated and accumulated in the volume texture.
Therefore, for every ray we can simply march through the volume starting from camera, calculate the density sum, accumulate in-scattering radiance and out-scattering falloff factor.



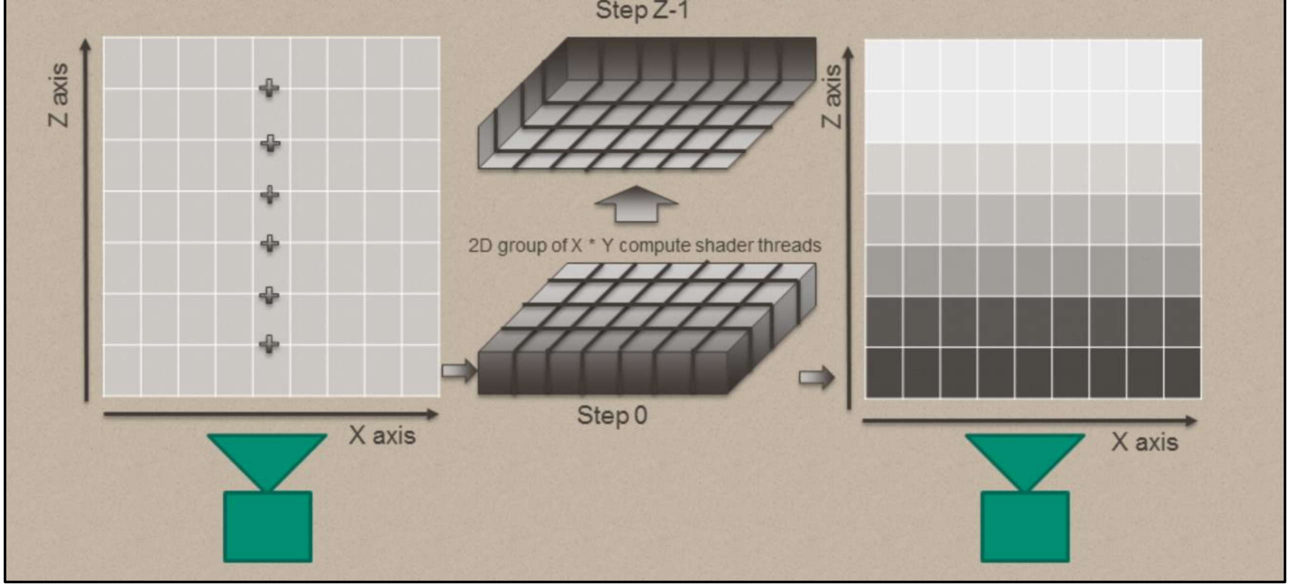
Solving scattering equation

- 2D compute shader
- Brute-force, numerical integration
- Marching through depth slices and accumulating
- Using UAV writes

Our compute shader accumulates in-scattered light and fog density at every step (numerical solution).
This way we end up with a volumetric texture with information at every texel how much light got in-scattered from camera to given 3D point and how much density of participating media we accumulated (which describes amount of out-scattered incoming light).

This data is ready to be applied in either forward (directly when drawing objects) or deferred (as fullscreen quad pass) manner.

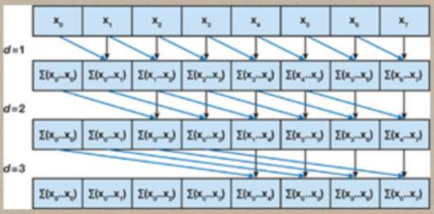
Solving scattering equation



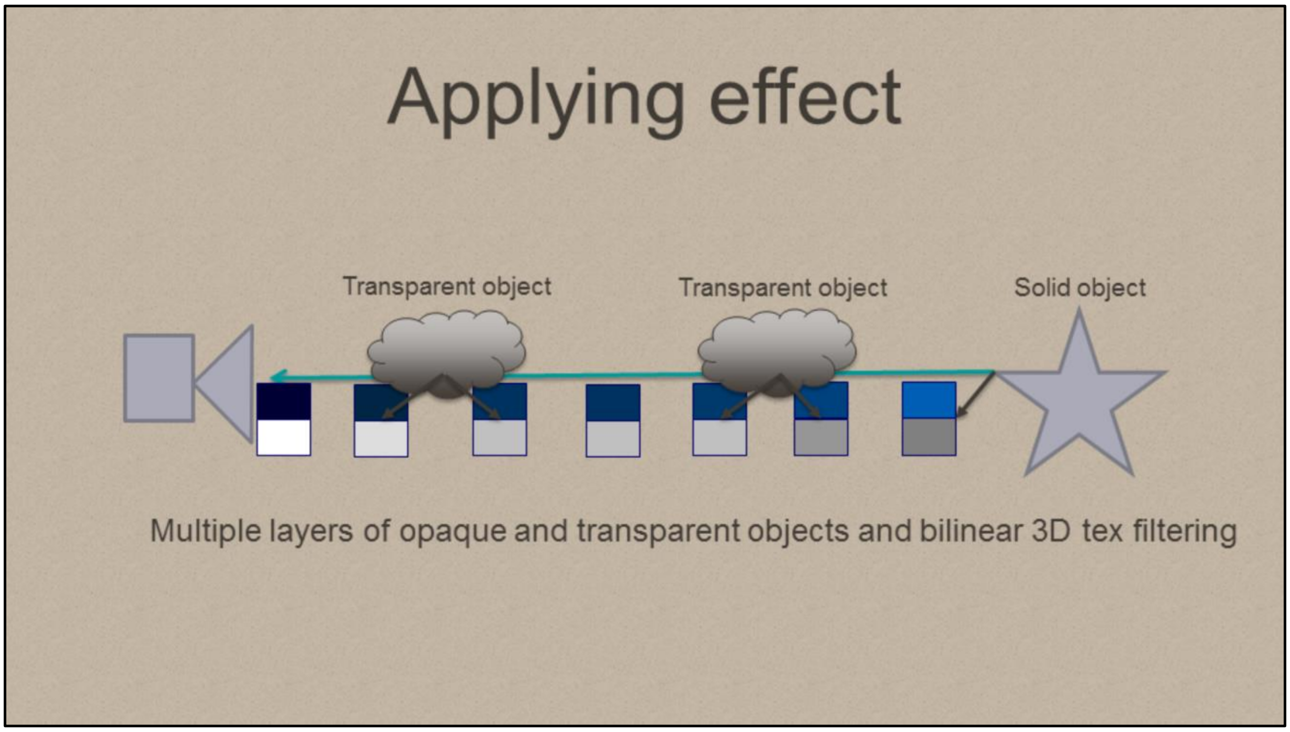
On this diagram we see how 2D compute shader thread group marches through 3D volume, accumulating in-scattered lighting and extinction coefficients.

Why not parallel sum?

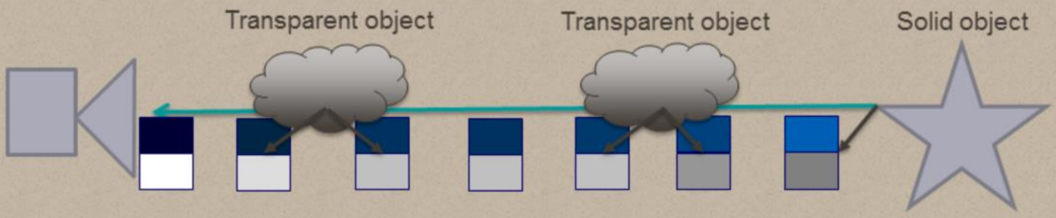
- Tried this approach!
- “Parallel Prefix Sum (Scan)”
- …but was 20-30% slower than brute-force!
  - LDS bank conflicts
  - Worse shader occupancy
  - Cache trashing / poor locality?
- Still, it may be better for 2D in some cases



Harris et al, “Parallel Prefix Sum (Scan) with CUDA”, GPU Gems 3

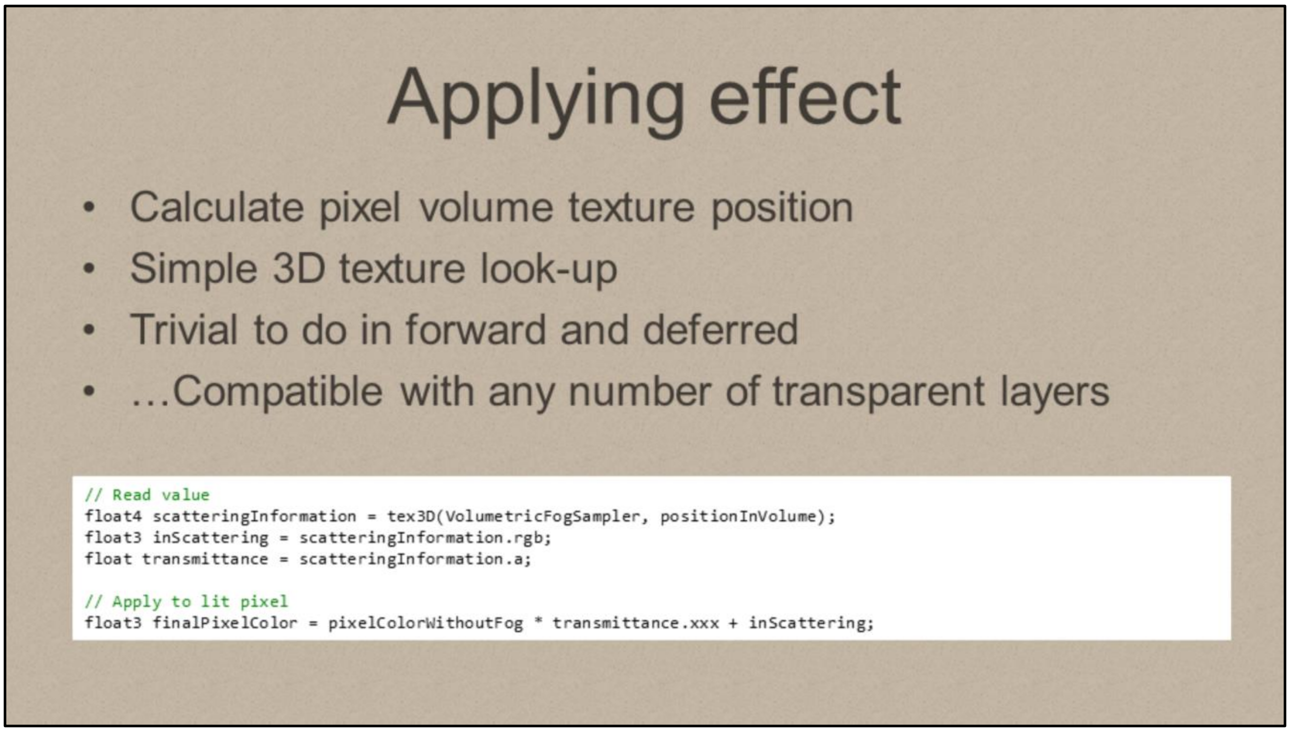


Applying effect



Multiple layers of opaque and transparent objects and bilinear 3D tex filtering

We can see on this diagram how due to storing whole view ray information we can apply volumetric fog information on both solid (1) and multiple transparent objects (2) and (3). They all have proper and filtered transmittance and in-scattering information.



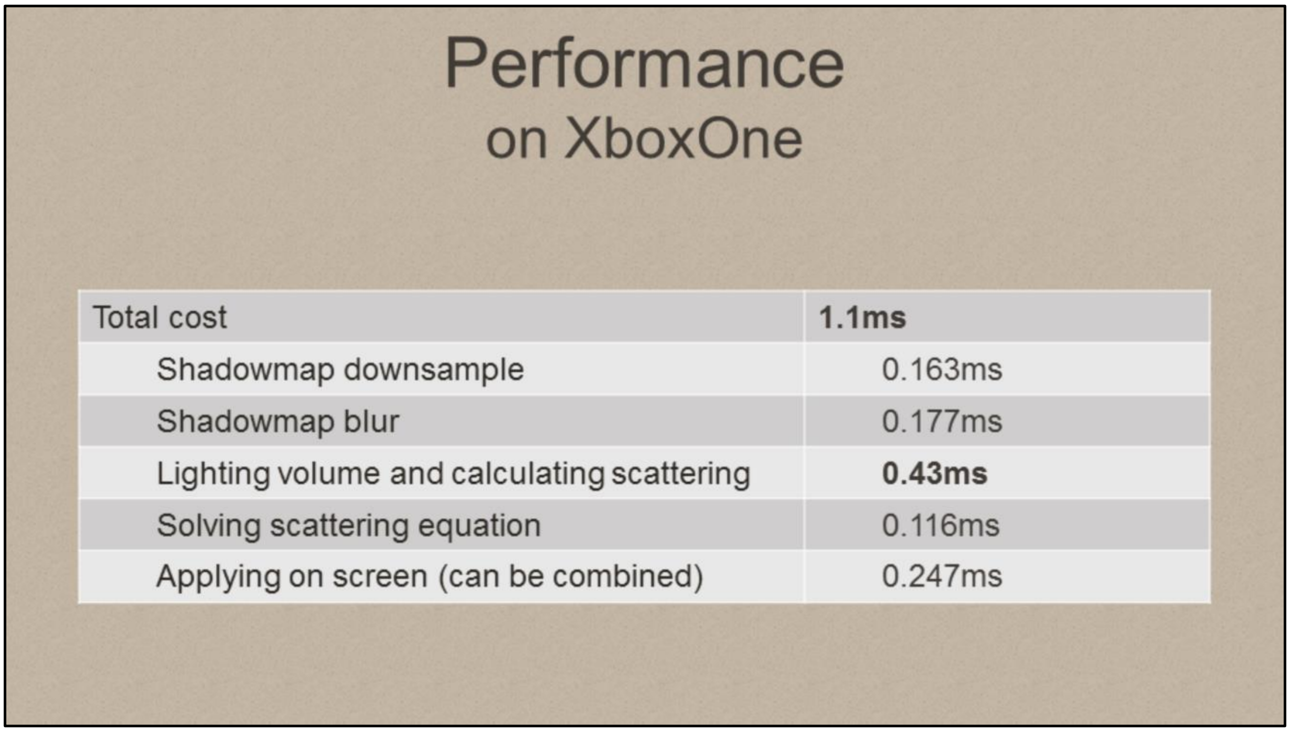
Applying effect

- Calculate pixel volume texture position
- Simple 3D texture look-up
- Trivial to do in forward and deferred
- …Compatible with any number of transparent layers

```
// Read value
float4 scatteringInformation = tex3D(VolumetricFogSampler, positionInVolume);
float3 inScattering = scatteringInformation.rgb;
float transmittance = scatteringInformation.a;

// Apply to lit pixel
float3 finalPixelColor = pixelColorWithoutFog * transmittance.xxx + inScattering;
```

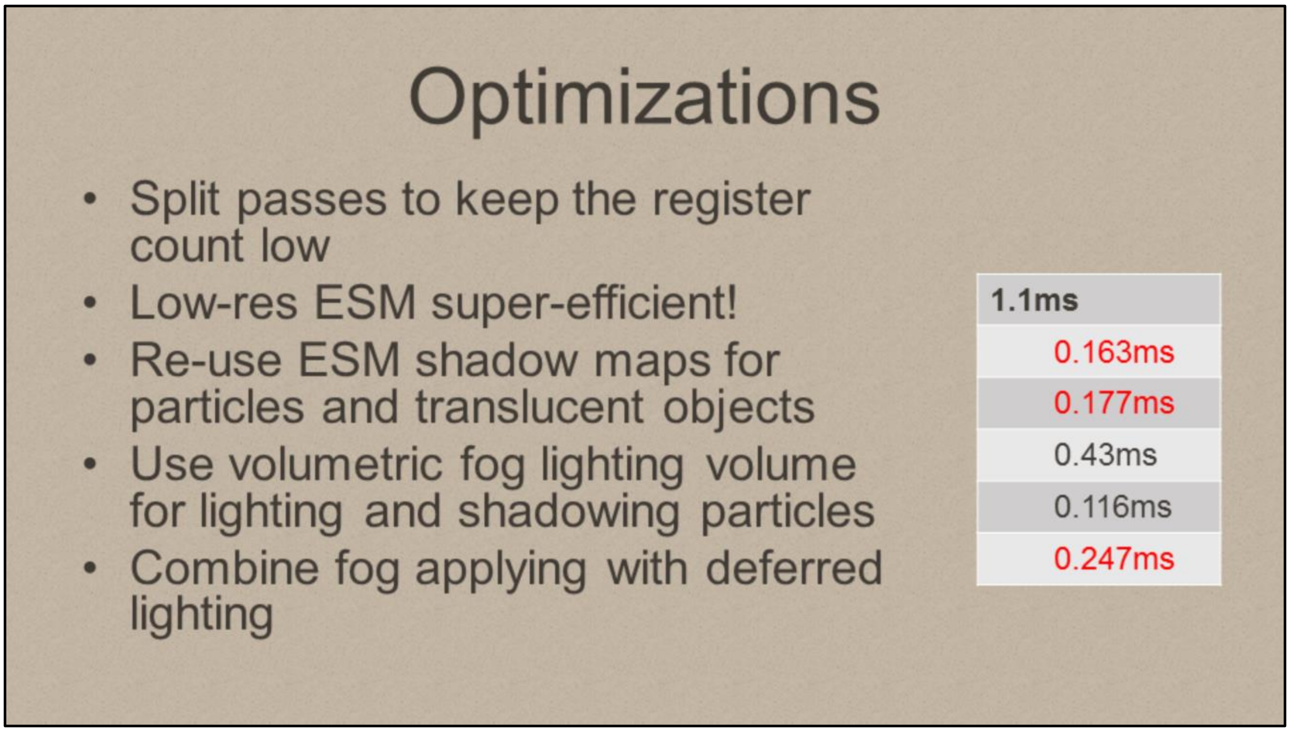
Final application of effect on screen is trivial in both deferred and forward – no special tricks, just a simple bilinear look-up from a 3D texture and a fused multiply-add operation.
Thing worth noting is that the effect is compatible with any number of transparent objects layers.



Performance
on XboxOne

Total cost	**1.1ms**
Shadowmap downsample	0.163ms
Shadowmap blur	0.177ms
Lighting volume and calculating scattering	**0.43ms**
Solving scattering equation	0.116ms
Applying on screen (can be combined)	0.247ms

Total cost was surprisingly small, around 1.1ms. Calculating it in double resolution had a cost of 1.6ms.
Most costly part was building density and lighting the volume, around 0.43ms.
Remaining passes were all under 0.2ms, except for “applying” pass that could be combined with lighting and would become “free”.

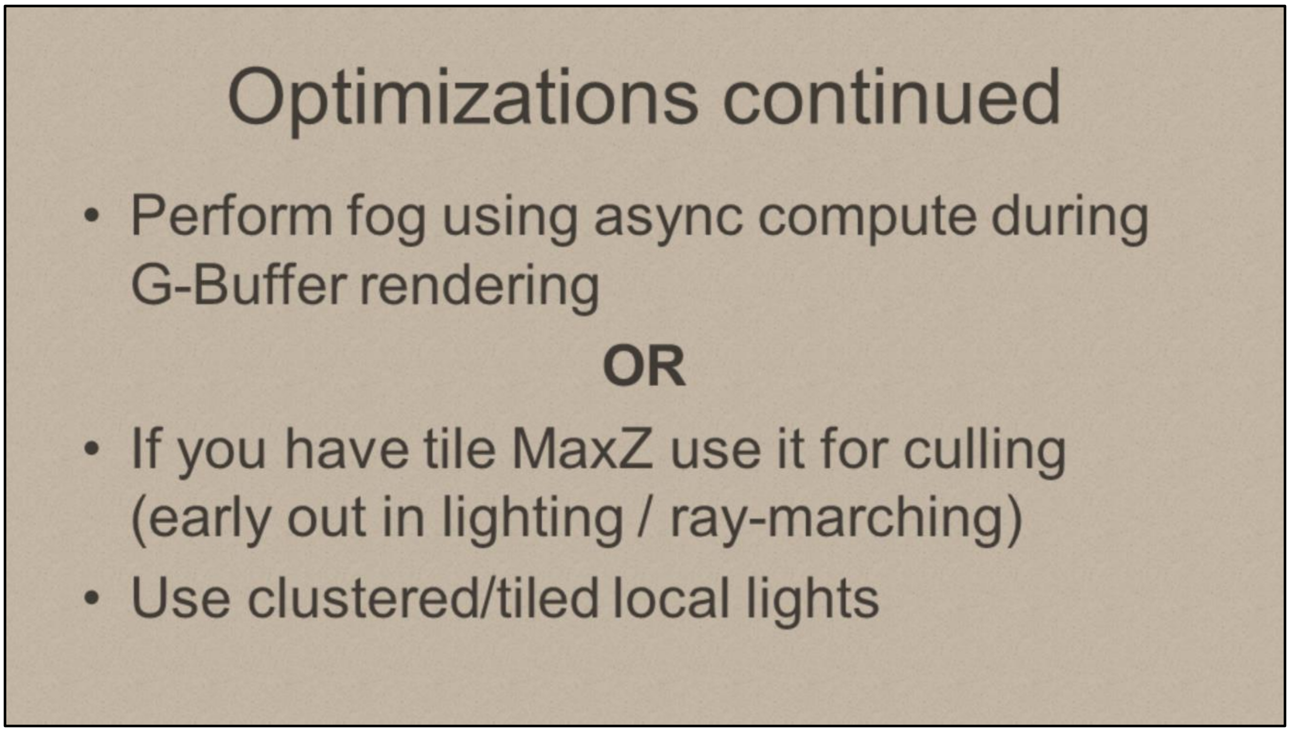


Optimizations

- Split passes to keep the register count low
- Low-res ESM super-efficient!
- Re-use ESM shadow maps for particles and translucent objects
- Use volumetric fog lighting volume for lighting and shadowing particles
- Combine fog applying with deferred lighting

1.1ms
0.163ms
0.177ms
0.43ms
0.116ms
0.247ms

First of all, volumetric fog was one of our effects that benefited the most from high wave occupancy and low VGPR counts – it was useful to split various passes.
Some of cost of volumetric fog is actually cost of shadow-map setup – it makes lots of sense to re-use it for other low frequency lighting – like particles, translucent and transparent objects (ocean etc).
Also the cost of calculating the lighting in texture and looping over multiple lights could be re-used by such render passes – by simple volumetric texture look-up.
Finally, there is no need to calculate full lighting with very high fog distances – after some distance shadowing and lighting could be LODed away.



Optimizations continued

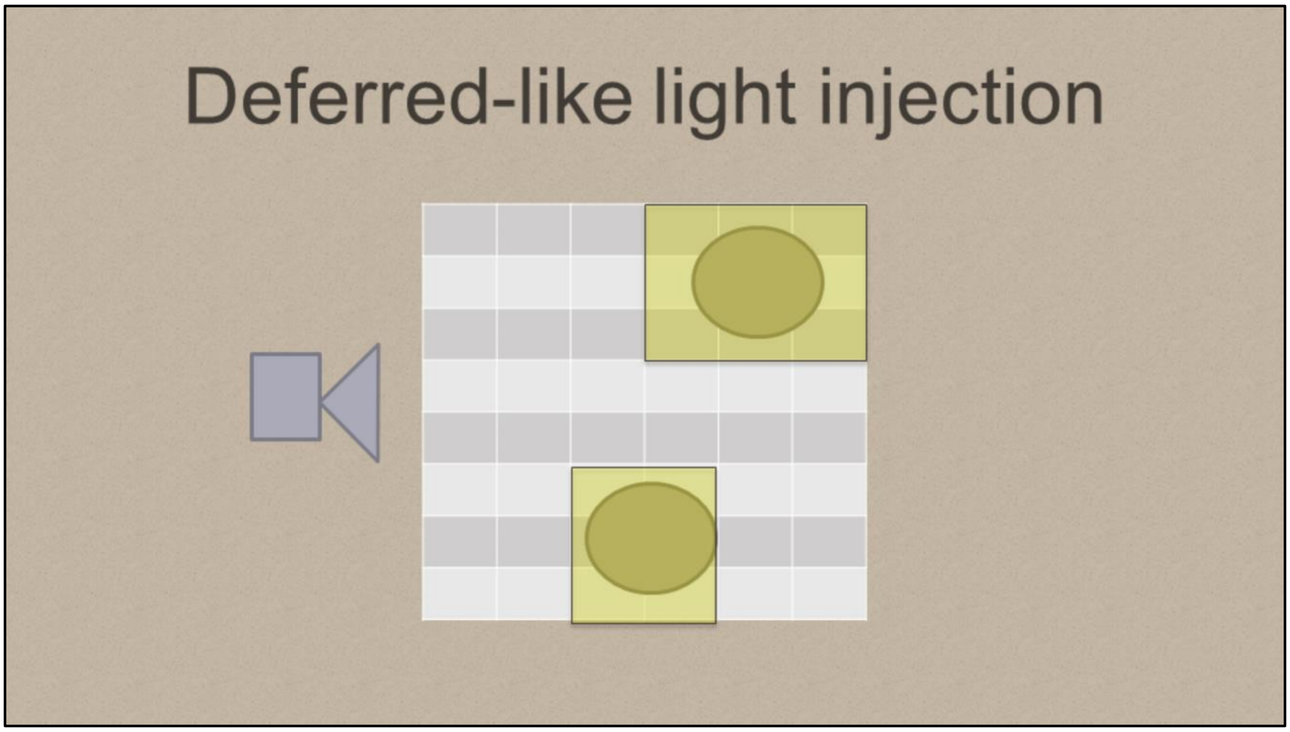
- Perform fog using async compute during G-Buffer rendering

OR

- If you have tile MaxZ use it for culling (early out in lighting / ray-marching)
- Use clustered/tiled local lights

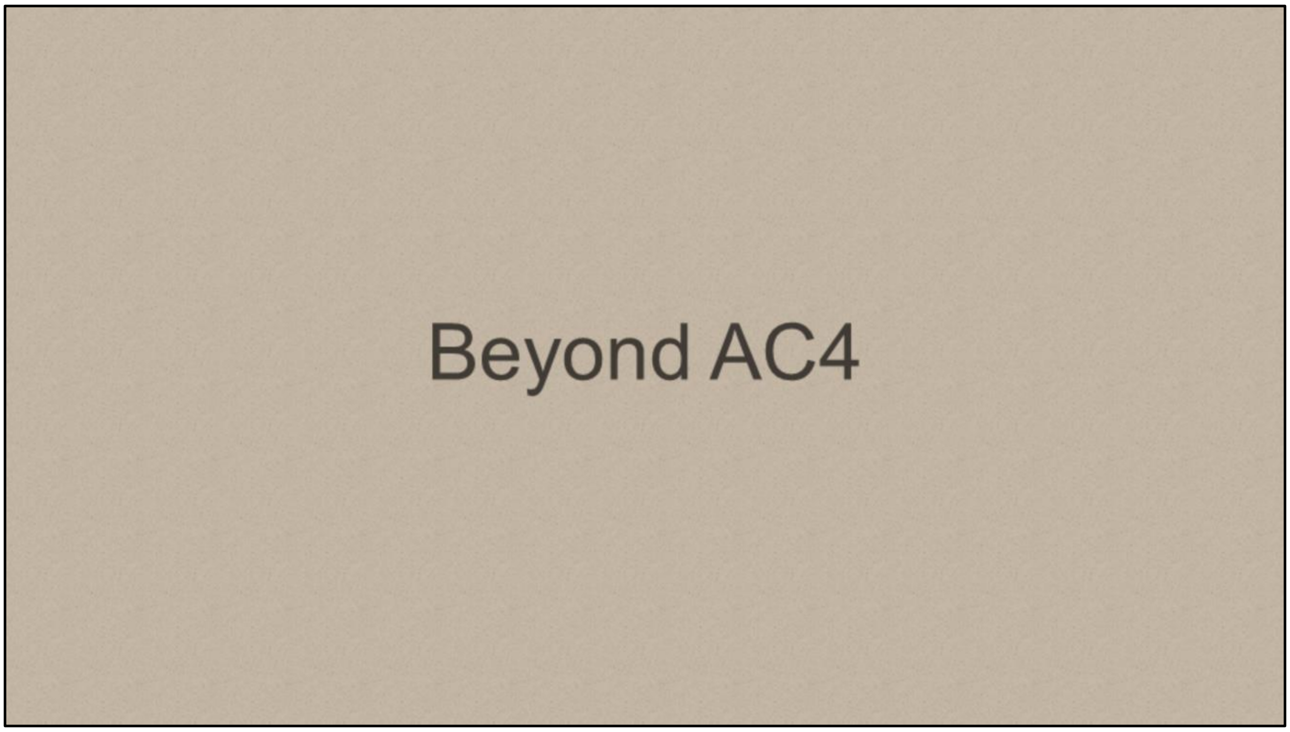
Some other optimizations include doing some early out on parts of the volumetric texture that is behind any actual rendered geometry. If engine has some form of hierarchical Z buffer available, such values can be read by volumetric fog and used for early out.
Similarly, there is no need to loop over all the visible lights – solutions like Forward+ or Clustered shading can be used for fog light culling.

If such information is not available in the engine or projected performance benefits wouldn't be big enough (for example lots of outdoor long-range scenes) then one could use next-gen consoles async compute capabilities for volumetric fog calculations. As the algorithm without culling doesn't rely on scene geometry it is possible to calculate it as soon as shadowmaps are ready. Volumetric fog can be quite heavy on both bandwidth and ALU, so it makes sense to put it in parallel with some vertex heavy pass like filling G-Buffer.



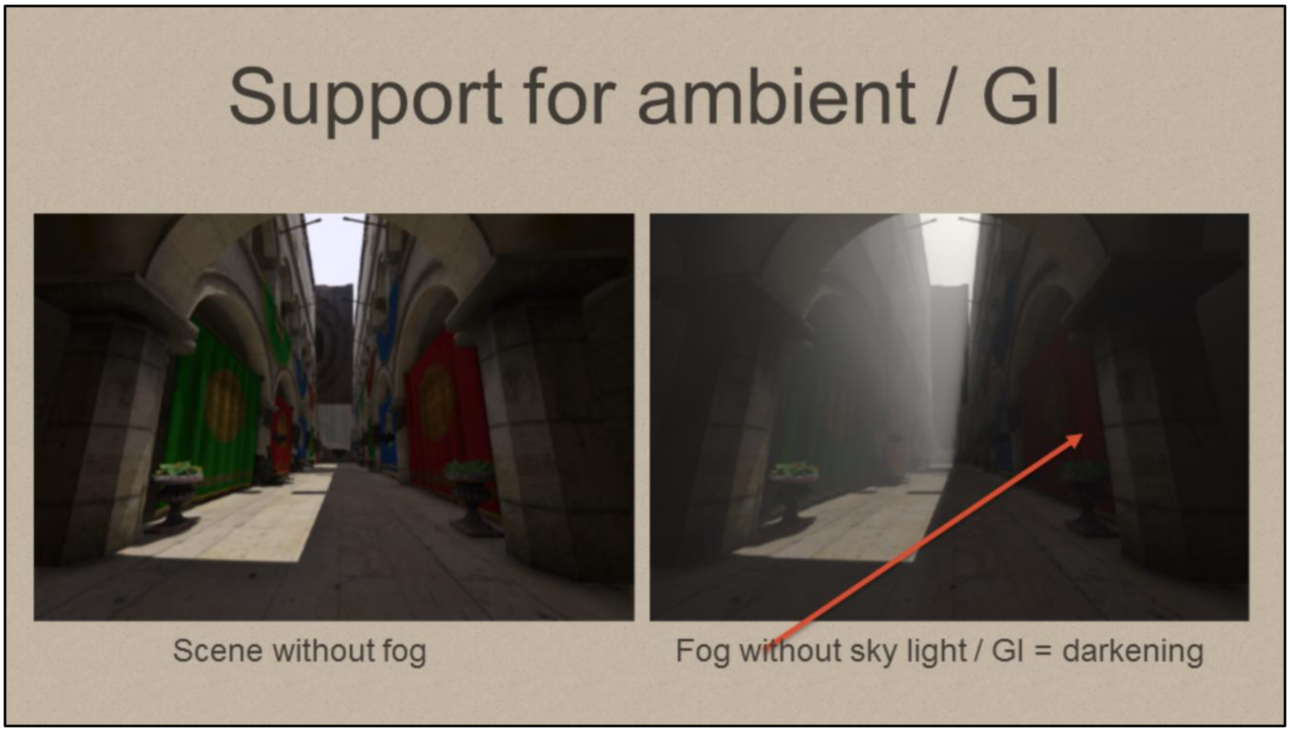
Deferred-like light injection

In similar manner, lighting could be also injected into volume as alternative to forward-based lighting techniques.
One could calculate volume bounding box of lights and perform lighting only on cells intersecting with it. It could be very easy and done using indirect dispatch functionality on DX11 / next gen consoles.
This optimization could be useful for very complex lights that use many VGPRs for computation – like area lights.

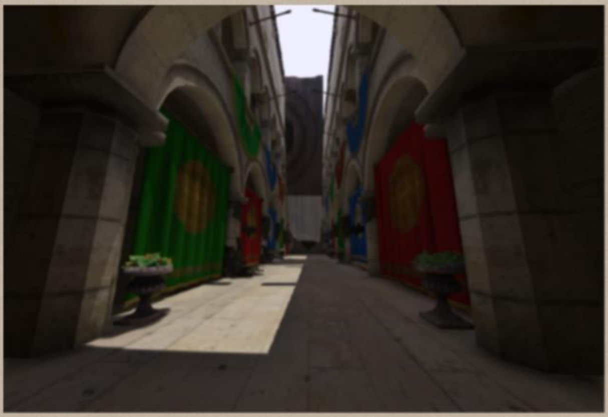

Beyond AC4

In the final presentation section, I will describe extensions to volumetric fog developed after AC4.

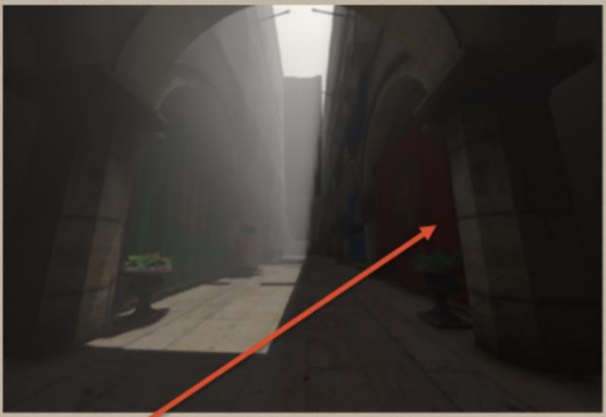
1. How to make effect much more physically based – missing pieces like support for physically based phase functions for analytical light sources and SH based light sources.
2. Making the effect even more stable by introducing temporal reprojection and jittering.



Support for ambient / GI

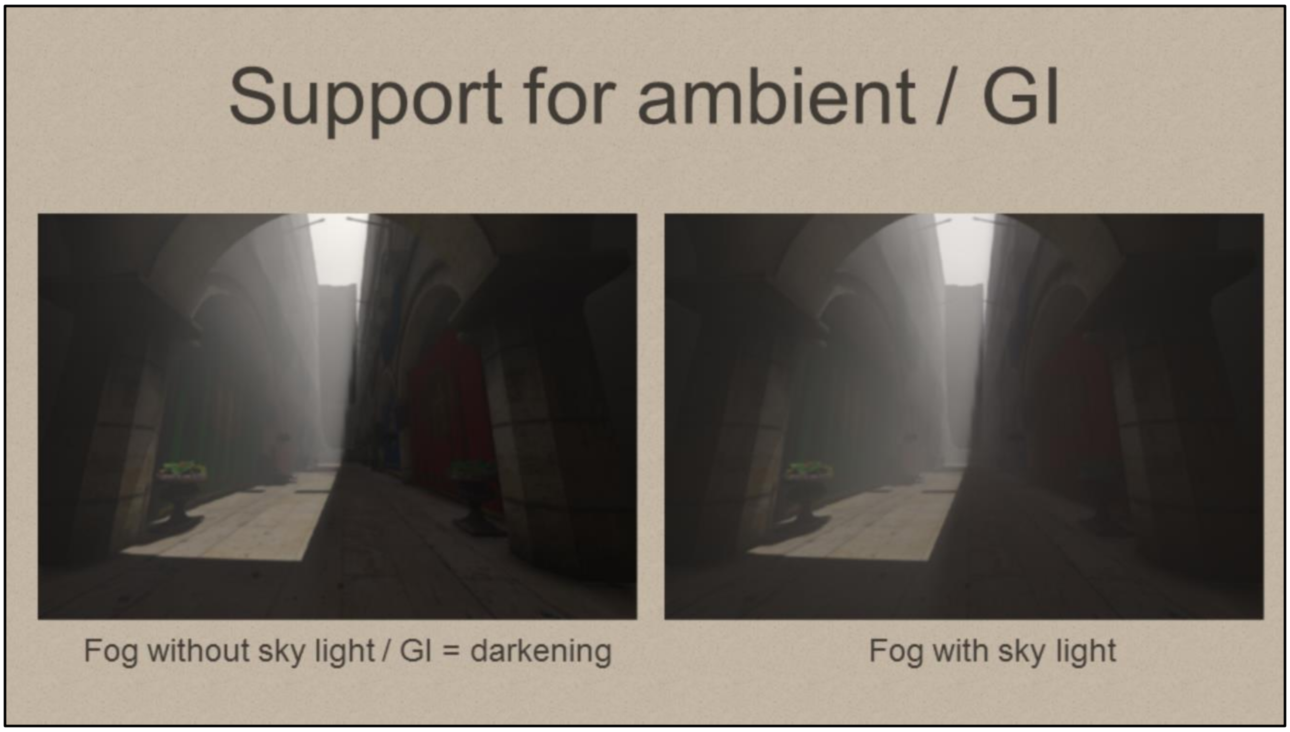


Scene without fog

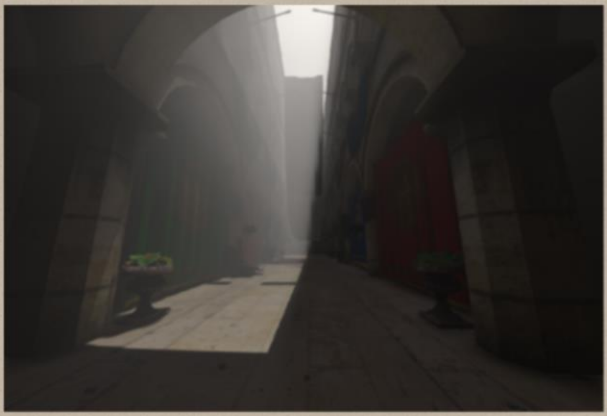


Fog without sky light / GI = darkening

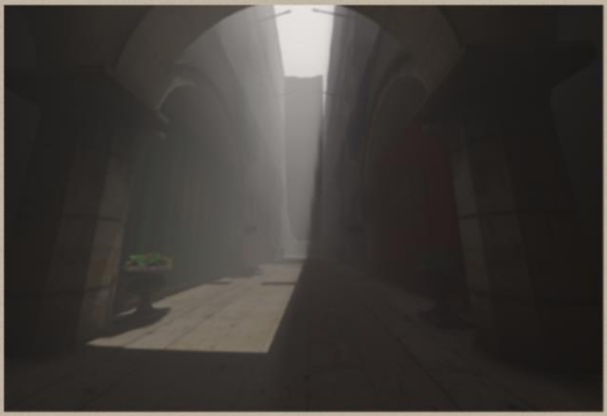
Lack of support of fog ambient/sky lighting results in too much scene darkening in the shadowed areas.
Some light gets out-scattered, but none is in-scattered due to lack of lighting and it produces unrealistic result. Therefore we need to add some GI.



Support for ambient / GI



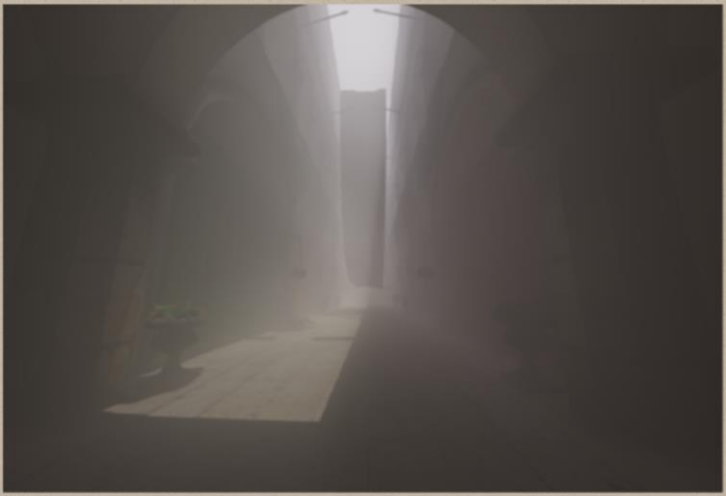
Fog without sky light / GI = darkening



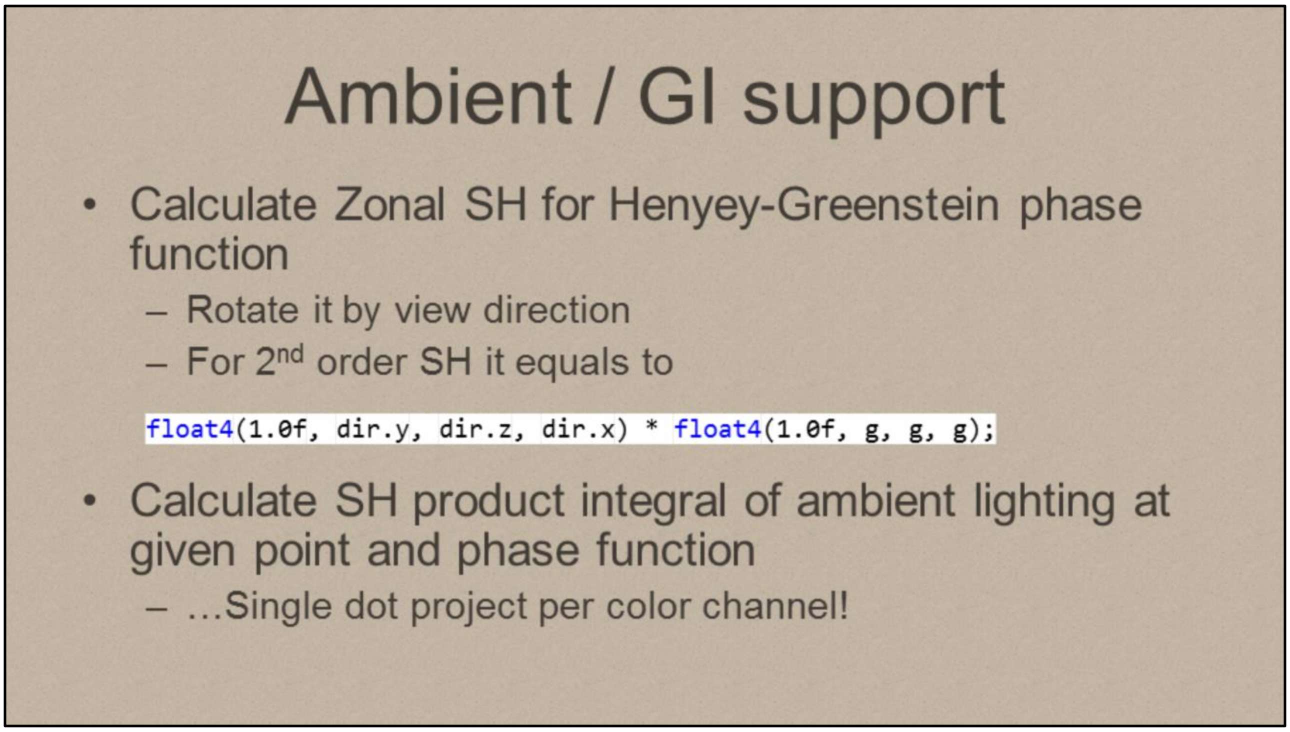
Fog with sky light

On this comparison we see how added sky lighting improves the result and adds proper fog in shadowed areas.

Exaggerated GI



Finally, we can see on this screenshot (exaggerated fog and GI) that the GI color contributes to the final fog color, adding proper color spatial variation.



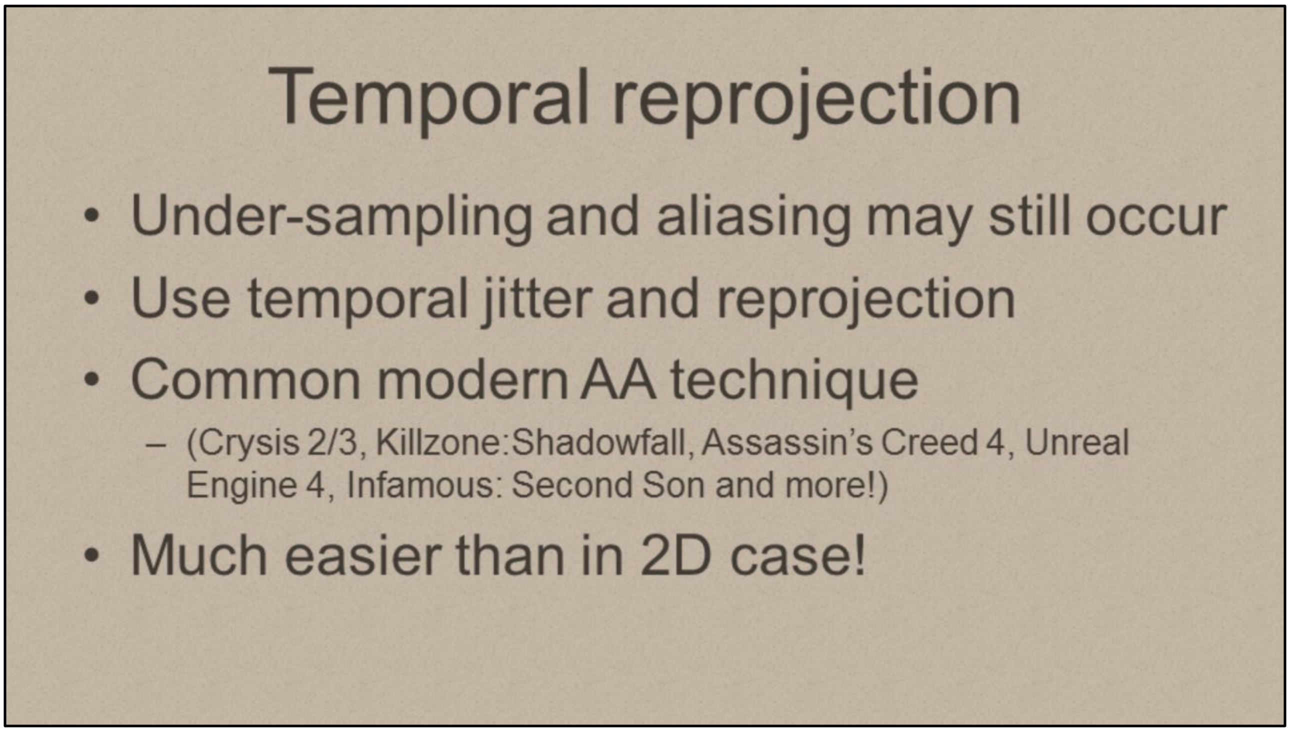
Ambient / GI support

- Calculate Zonal SH for Henyey-Greenstein phase function
  - Rotate it by view direction
  - For 2nd order SH it equals to

```
float4(1.0f, dir.y, dir.z, dir.x) * float4(1.0f, g, g, g);
```

- Calculate SH product integral of ambient lighting at given point and phase function
  - …Single dot project per color channel!

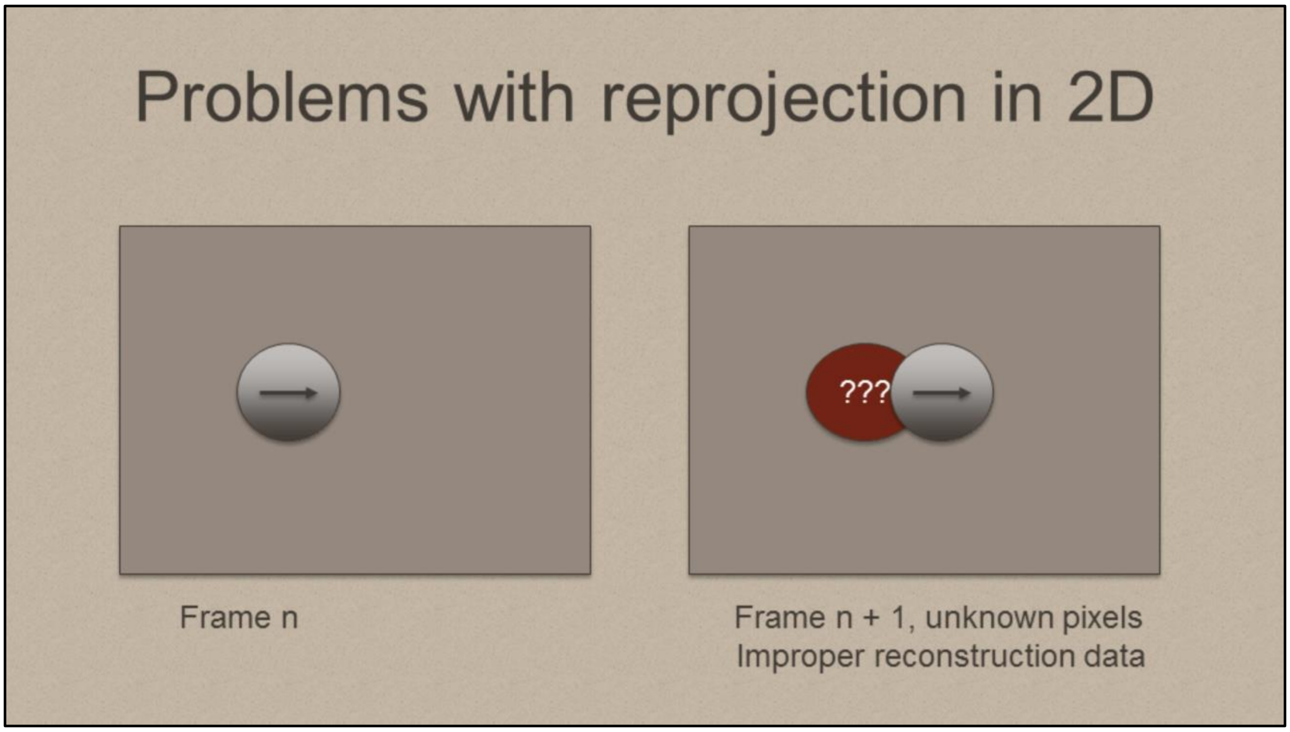
One of very common lighting storage basis for ambient lighting or GI are spherical harmonics.
They have multiple mathematical properties that make them so useful (orthonormality, rotation invariance etc.), but another one is the easiness of use of Zonal Spherical harmonics.
They can be trivially rotated (especially for lower order SH) and it is useful for volumetric fog as well.
HG phase function has trivial expansion to Zonal SH and for 2nd order it can be easily rotated (code on slide).
Then to calculate volumetric fog response, one has to simply calculate SH product integral of rotated phase function SH expansion by view vector and stored sky lighting / GI SH representation.



Temporal reprojection

- Under-sampling and aliasing may still occur
- Use temporal jitter and reprojection
- Common modern AA technique
  - (Crysis 2/3, Killzone:Shadowfall, Assassin's Creed 4, Unreal Engine 4, Infamous: Second Son and more!)
- Much easier than in 2D case!

While we spend considerable amount of time to remove aliasing and under-sampling problems on volumetric fog by using down-sampled SM representation, there still may be some under-sampling artifacts.
One of ideas to fix it is to use the temporal super-sampling by jittering the sample patterns every frame and using temporal smoothing and reprojection to combine such patterns from multiple frames.
It is used as a common AA technique in more and more games, but can be used for volumetric fog as well – as reprojection is much easier in 3D than in 2D case.

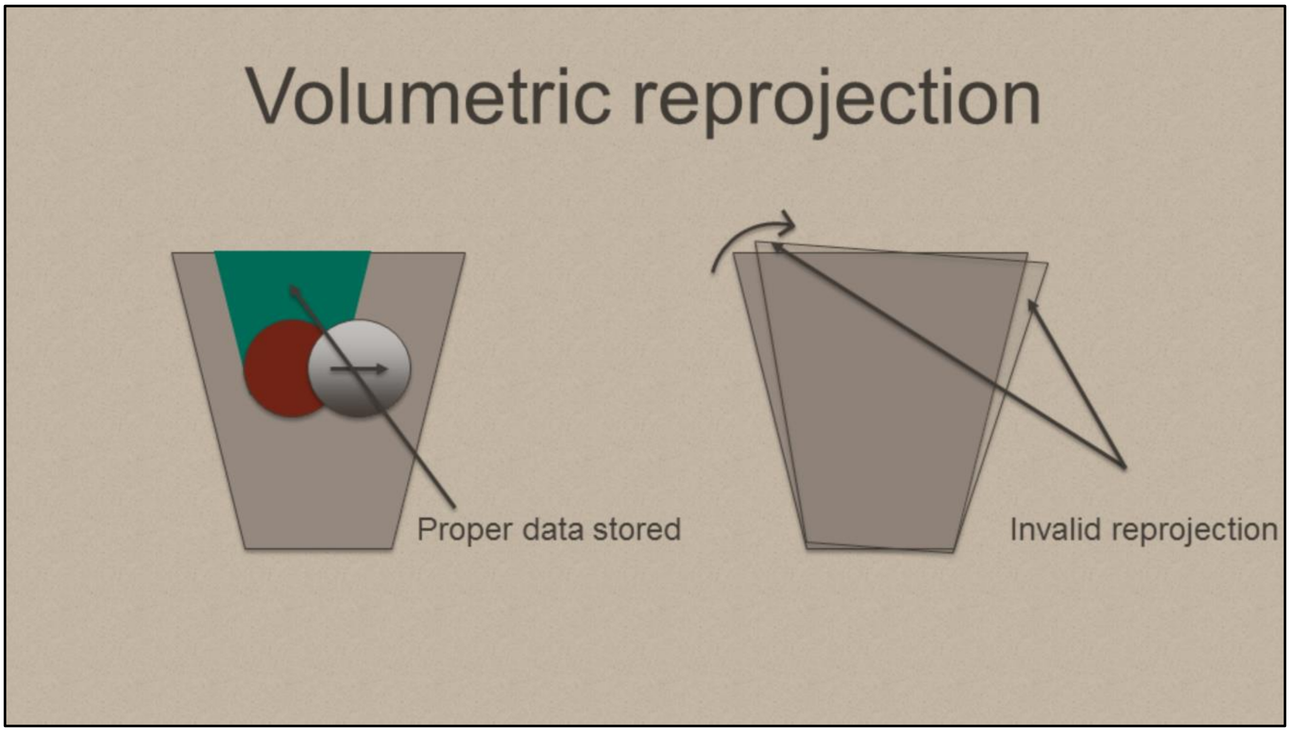


Problems with reprojection in 2D

Frame n

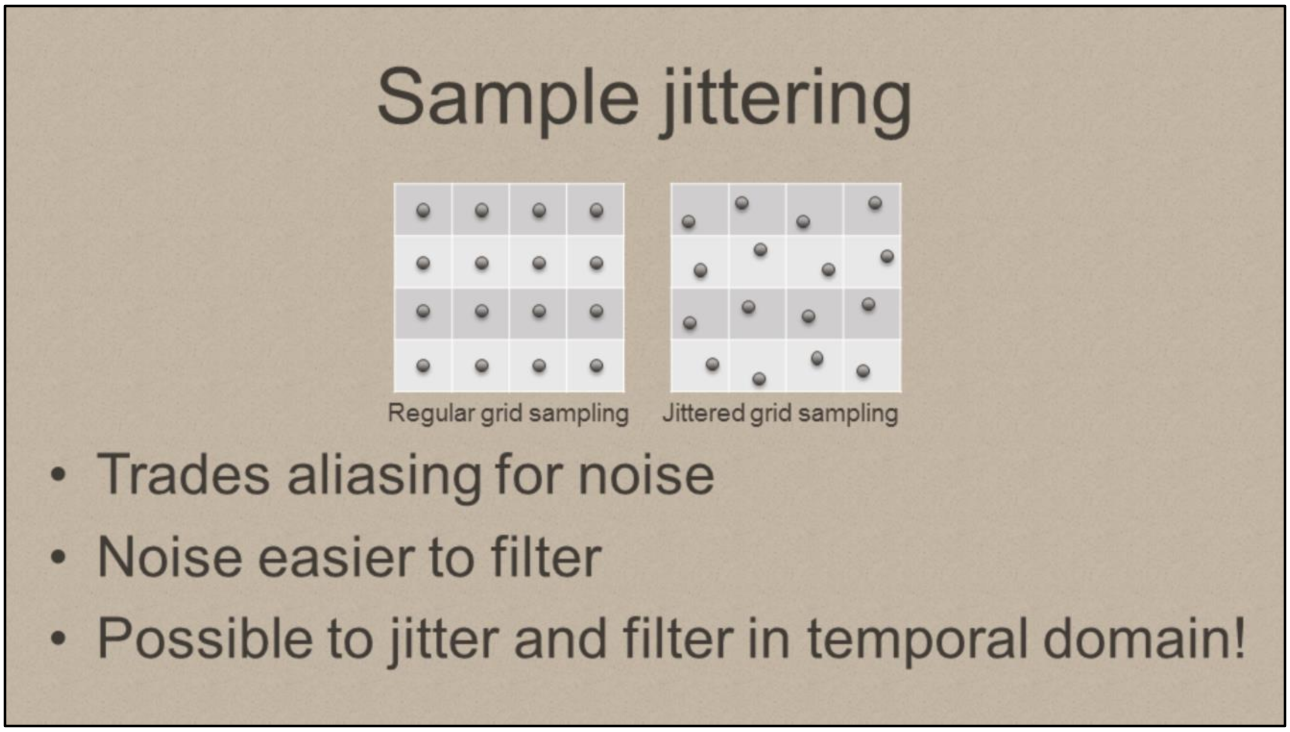
Frame n + 1, unknown pixels
Improper reconstruction data

Reprojection is usually problematic in 2D as there are occlusion / disocclusion problems when objects on the scene move. It is difficult to both detect such movement (especially when jittering) and to find proper data to fill resulting holes. Therefore usually algorithms struggle in finding a sweet-spot between potential shaking and blurring / tracing / ghosting of dynamic objects.

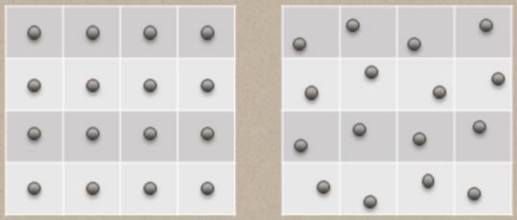


Volumetric reprojection

In 3D it is much easier. While space occupied by a dynamic object after movement is still invalid, all the data behind it is proper (if not using any early culling optimizations). Only data that was outside the volume aligned with frustum is invalid. (But still, obvious reprojection problems can happen with strongly view dependent and anisotropic phase functions).

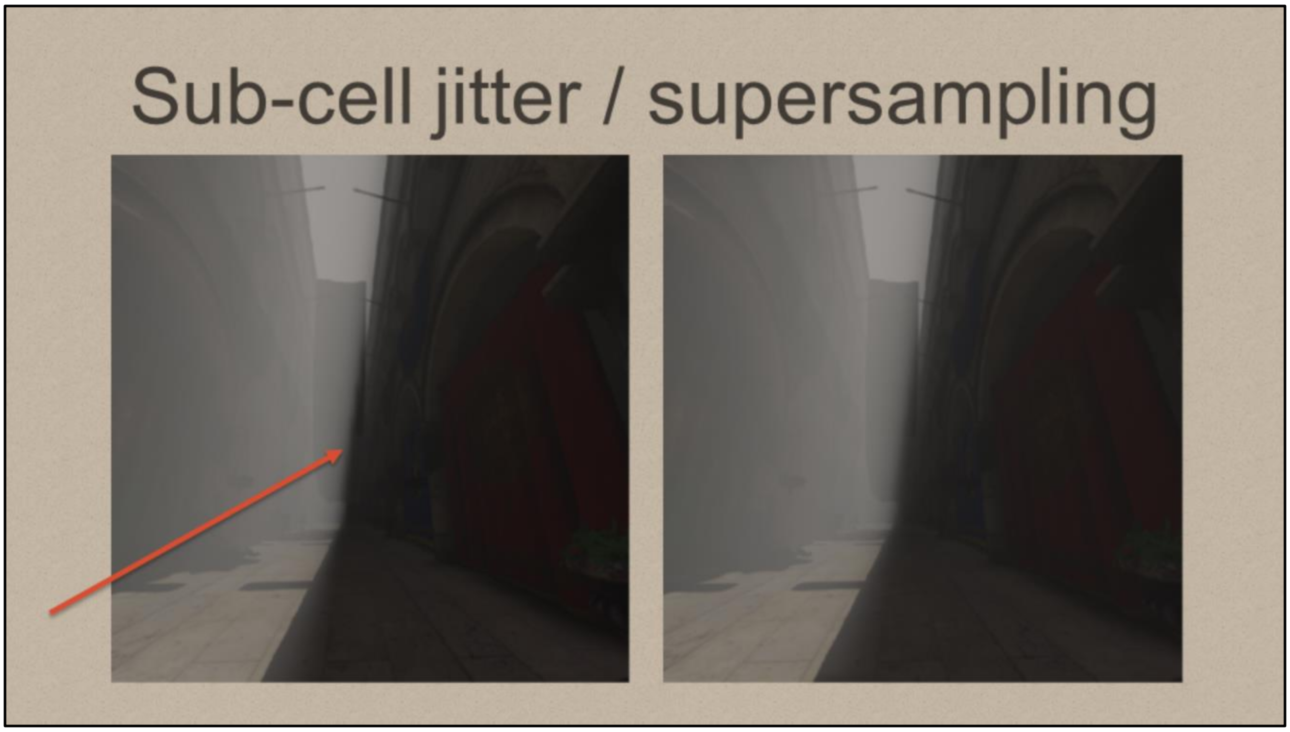


Sample jittering



- Trades aliasing for noise
- Noise easier to filter
- Possible to jitter and filter in temporal domain!

Another aliasing countering technique is sample jittering in spatial grid. While regular grid sampling produces aliasing that is hard to fight (as low frequency components not visible in the source signal appear), jittered grid sampling trades aliasing for higher frequency noise.
Noise and higher frequency signals can be easily filtered and blurred using lower resolution kernels.
Furthermore, it is possible to jitter and filter in both temporal and spatial domains.



Sub-cell jitter / supersampling

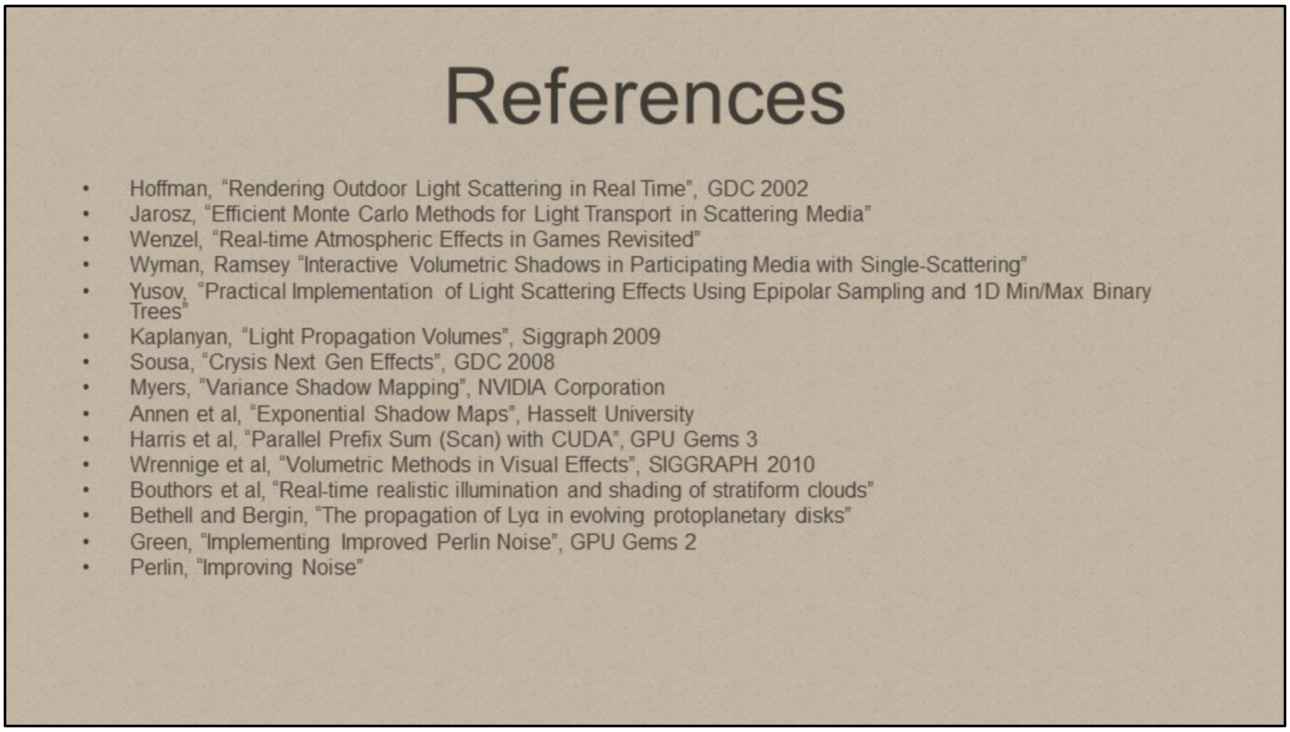
This two screenshots show how much better results are achieved by only simplest 1-sample temporal jitter and reprojection – almost all edge artifacts are gone!
We didn't check any complex 3D jittered grid sampling patterns, but it is worth investigating in the future as results could be even better.

Special thanks

- Natalya Tatarchuk
- People at Ubisoft Montreal
  - Ulrich Haar
  - Stephen Hill
  - Lionel Berenguier
  - Typhaine Le Gallo
  - Alexandre Lahaise

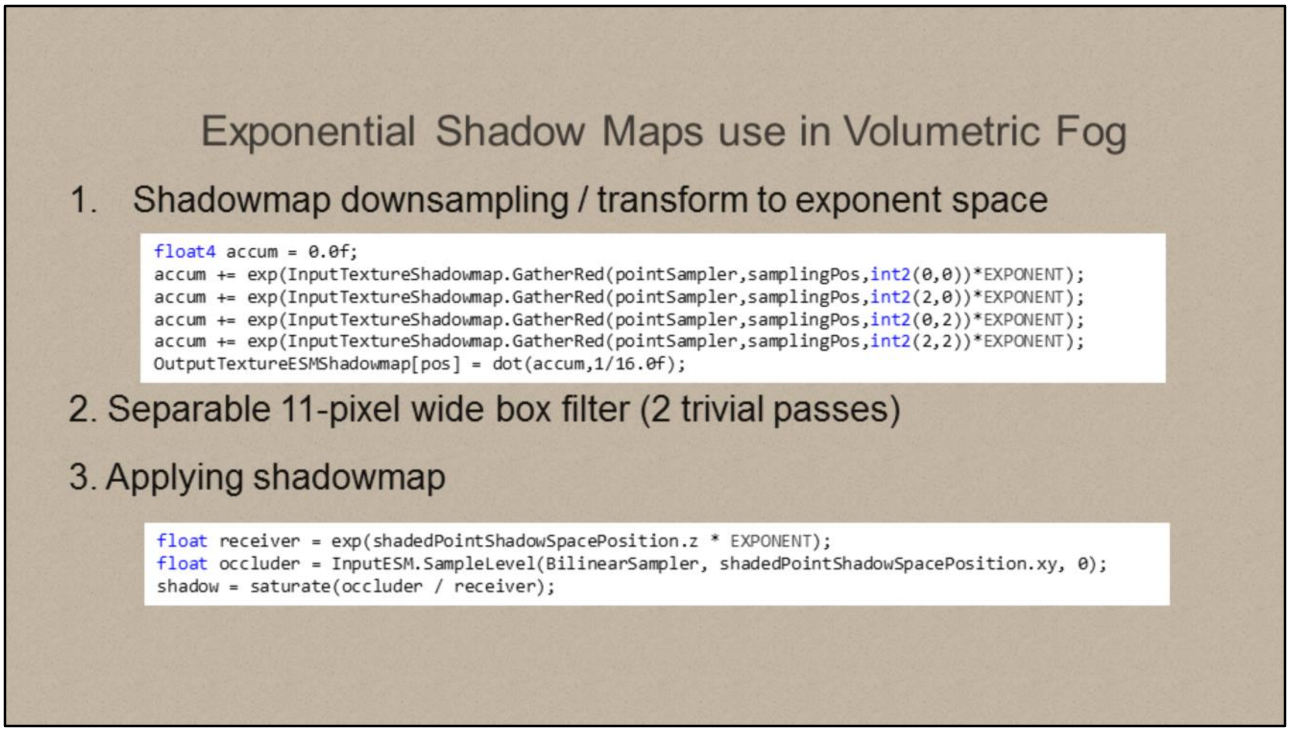
Questions?

- Contact me!
- Email: b_wronski@op.pl
- WWW / slides / code:
  http://www.bartwronski.com
- Twitter: @BartWronsk



References

- Hoffman, "Rendering Outdoor Light Scattering in Real Time", GDC 2002
- Jarosz, "Efficient Monte Carlo Methods for Light Transport in Scattering Media"
- Wenzel, "Real-time Atmospheric Effects in Games Revisited"
- Wyman, Ramsey "Interactive Volumetric Shadows in Participating Media with Single-Scattering"
- Yusov, "Practical Implementation of Light Scattering Effects Using Epipolar Sampling and 1D Min/Max Binary Trees"
- Kaplanyan, "Light Propagation Volumes", Siggraph 2009
- Sousa, "Crysis Next Gen Effects", GDC 2008
- Myers, "Variance Shadow Mapping", NVIDIA Corporation
- Annen et al, "Exponential Shadow Maps", Hasselt University
- Harris et al, "Parallel Prefix Sum (Scan) with CUDA", GPU Gems 3
- Wrennige et al, "Volumetric Methods in Visual Effects", SIGGRAPH 2010
- Bouthors et al, "Real-time realistic illumination and shading of stratiform clouds"
- Bethell and Bergin, "The propagation of Lyα in evolving protoplanetary disks"
- Green, "Implementing Improved Perlin Noise", GPU Gems 2
- Perlin, "Improving Noise"



Exponential Shadow Maps use in Volumetric Fog

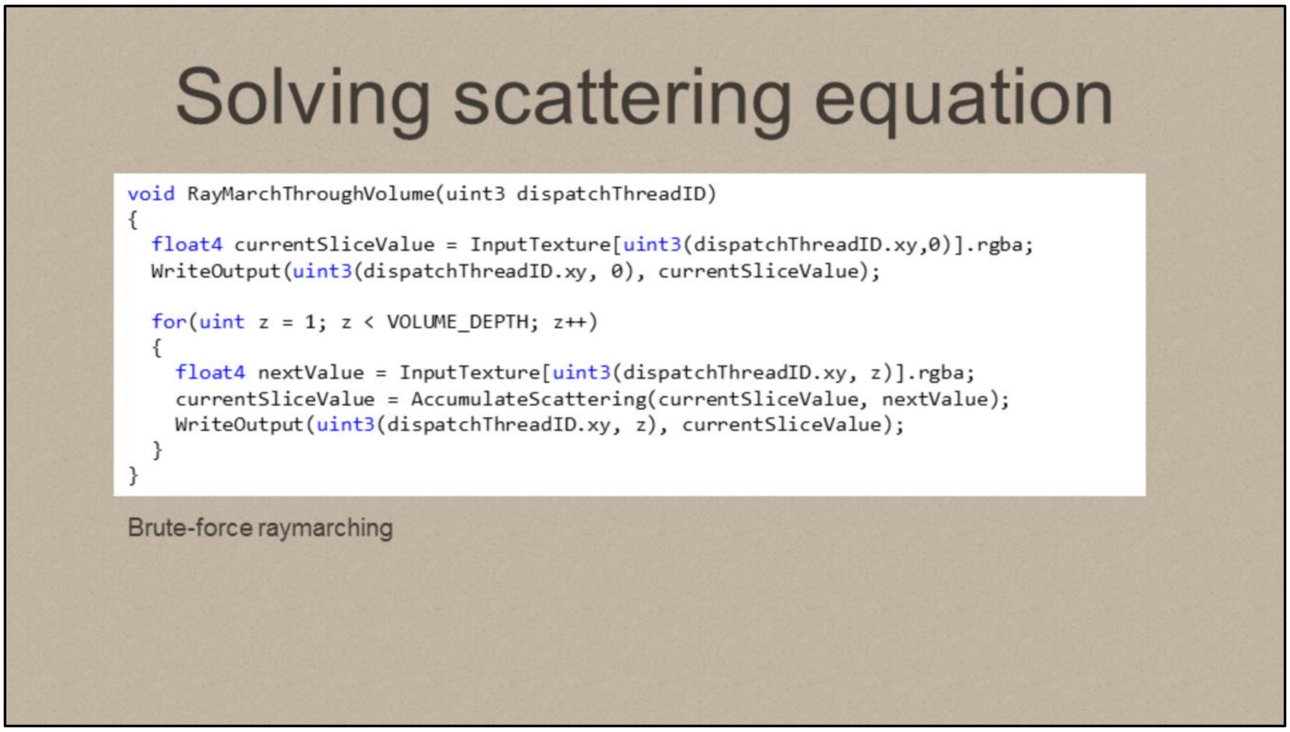
1. Shadowmap downsampling / transform to exponent space

```
float4 accum = 0.0f;
accum += exp(InputTextureShadowmap.GatherRed(pointSampler,samplingPos,int2(0,0))*EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler,samplingPos,int2(2,0))*EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler,samplingPos,int2(0,2))*EXPONENT);
accum += exp(InputTextureShadowmap.GatherRed(pointSampler,samplingPos,int2(2,2))*EXPONENT);
OutputTextureESMShadowmap[pos] = dot(accum,1/16.0f);
```

2. Separable 11-pixel wide box filter (2 trivial passes)

3. Applying shadowmap

```
float receiver = exp(shadedPointShadowSpacePosition.z * EXPONENT);
float occluder = InputESM.SampleLevel(BilinearSampler, shadedPointShadowSpacePosition.xy, 0);
shadow = saturate(occluder / receiver);
```

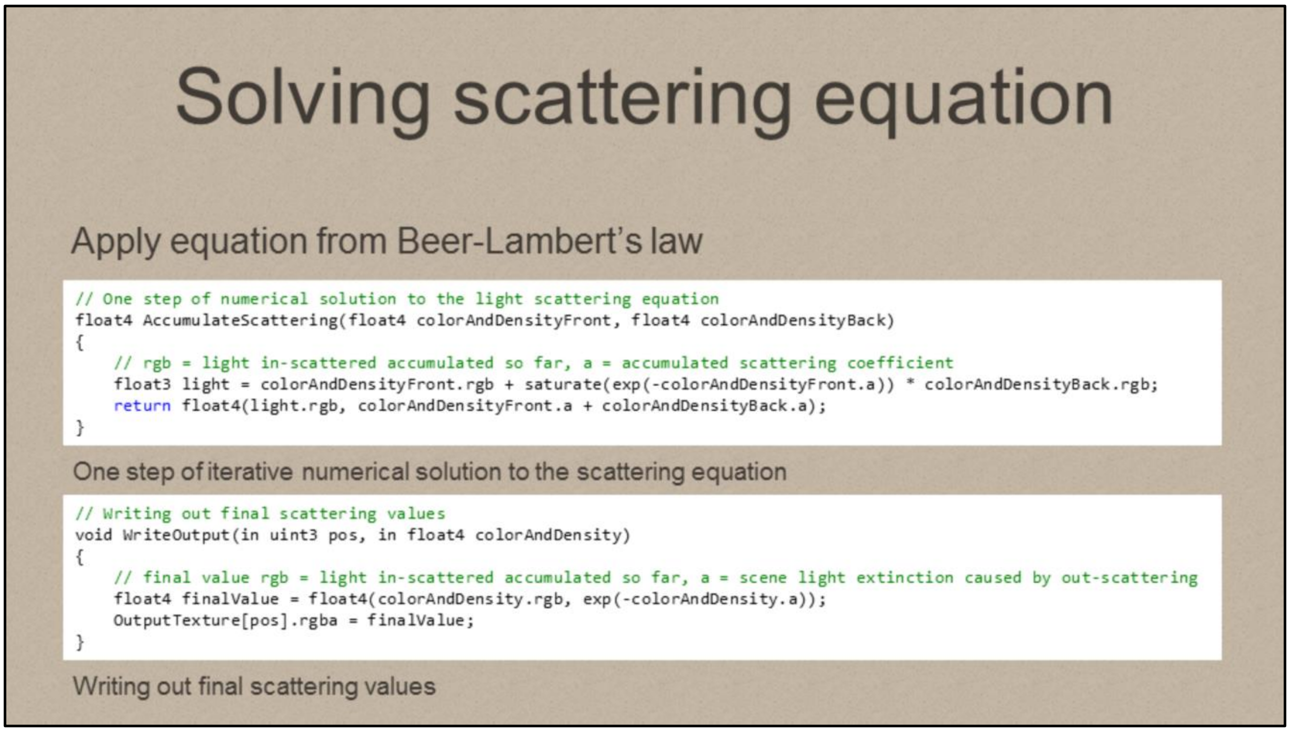


Solving scattering equation

```
void RayMarchThroughVolume(uint3 dispatchThreadID)
{
  float4 currentSliceValue = InputTexture[uint3(dispatchThreadID.xy,0)].rgba;
  WriteOutput(uint3(dispatchThreadID.xy, 0), currentSliceValue);

  for(uint z = 1; z < VOLUME_DEPTH; z++)
  {
    float4 nextValue = InputTexture[uint3(dispatchThreadID.xy, z)].rgba;
    currentSliceValue = AccumulateScattering(currentSliceValue, nextValue);
    WriteOutput(uint3(dispatchThreadID.xy, z), currentSliceValue);
  }
}
```

Brute-force raymarching



Solving scattering equation

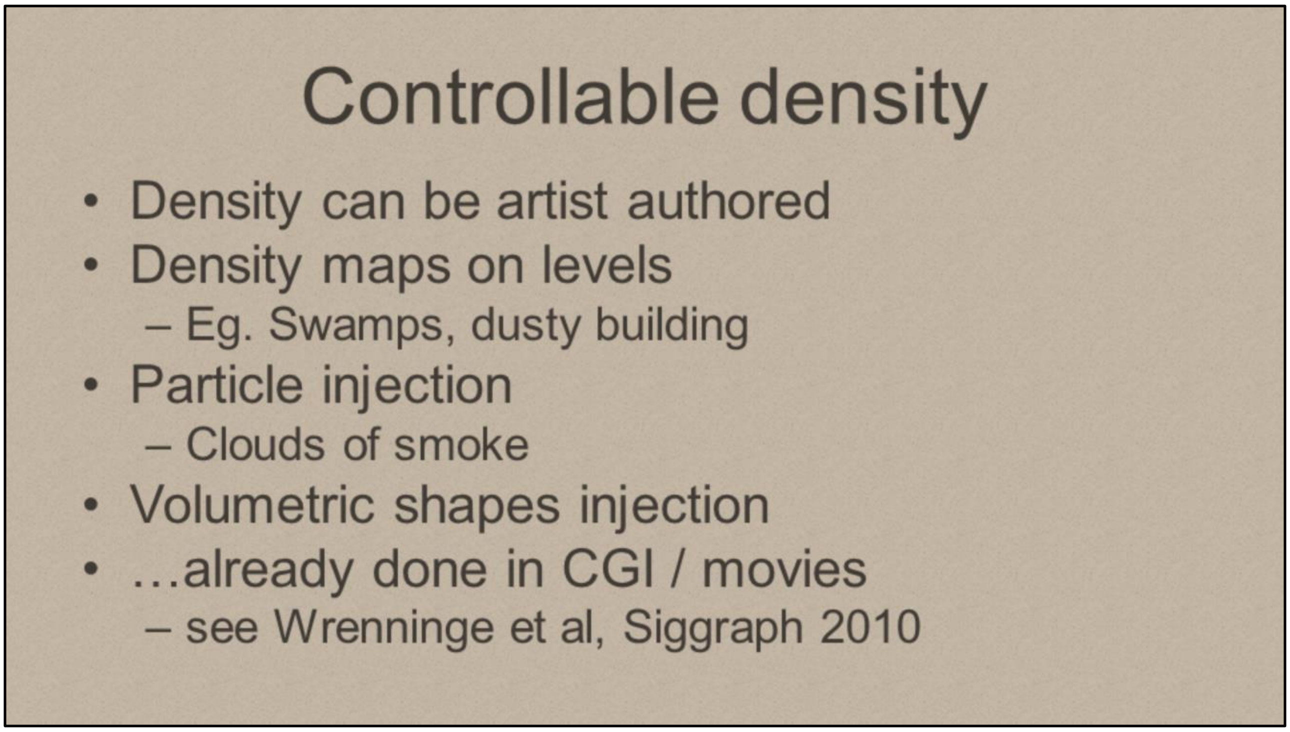
Apply equation from Beer-Lambert's law

```
// One step of numerical solution to the light scattering equation
float4 AccumulateScattering(float4 colorAndDensityFront, float4 colorAndDensityBack)
{
    // rgb = light in-scattered accumulated so far, a = accumulated scattering coefficient
    float3 light = colorAndDensityFront.rgb + saturate(exp(-colorAndDensityFront.a)) * colorAndDensityBack.rgb;
    return float4(light.rgb, colorAndDensityFront.a + colorAndDensityBack.a);
}
```

One step of iterative numerical solution to the scattering equation

```
// Writing out final scattering values
void WriteOutput(in uint3 pos, in float4 colorAndDensity)
{
    // final value rgb = light in-scattered accumulated so far, a = scene light extinction caused by out-scattering
    float4 finalValue = float4(colorAndDensity.rgb, exp(-colorAndDensity.a));
    OutputTexture[pos].rgba = finalValue;
}
```

Writing out final scattering values



Controllable density

- Density can be artist authored
- Density maps on levels
  - Eg. Swamps, dusty building
- Particle injection
  - Clouds of smoke
- Volumetric shapes injection
- …already done in CGI / movies
  - see Wrenninge et al, Siggraph 2010

While in our case density was a simple animated Perlin noise function, it could be controlled by artists in any way.
Density could be authored and painted on levels for some very objects full of haze and scattering particles like swamps (mist) or dusty buildings.
It could be injected into volume dynamically by particle systems or some analytical volumetric shapes / force fields.
See Wrenninge et al at Siggraph 2010 to check how it was done long time ago in movie / CGI industry! ☺